EDAN40/EDAN95: Functional Programming — Reference Sheet
Lund University

Karl Hallsby

Last Edited: April 27, 2020

Contents

List of Theorems

List of Listings

1 Introduction

This section is dedicated to giving a small introduction to functional programming. Functional Programming is a style of programming, nothing else. In this style, the basic method of computation is the evaluation of expressions as arguments to functions, which themselves return expressions.

> "Functional programming is so called because a program consists entirely of functions. [...] These functions are much like ordinary mathematical functions [...] defined by ordinary equations" (John Hughes)

If you want to view all possible language categories, visit Wikipedia's Programming Paradigms.

Defn 1 (Imperative Programming Language)**.** *Imperative programming languages* have a programming paradigm that uses statements that change a program's state. An imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates.

Defn 2 (Functional Programming Language)**.** *Functional programming languages* treat computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm in that programming is done with expressions or declarations instead of statements. In functional code, the output value of a function depends only on its arguments, so calling a function with the same value for an argument always produces the same result.

This is in contrast to Imperative Programming Languages where, in addition to a function's arguments, global program state can affect a function's resulting value. Eliminating side effects, that is, changes in state that do not depend on the function inputs, can make understanding a program easier, which is one of the key motivations for the development of functional programming.

Because of its close relationship to mathematics, it is much easier to develop mathematical techniques for reasoning about and proving the behavior of programs developed in functional languages. These techniques are important tools for helping us to ensure that programs work properly without having to resort to tedious testing and debugging which can only show the presence of errors, never their absence. Moreover, they provide important tools for documenting the reasoning that went into the formulation of a program, making the code easier to understand and maintain.

Remark 2.1 (Course Language). The languages of use in this course is Haskell. It is a purely functional language that supports impure actions with Monads.

Functional programming is very nice because it allows us to perform certain actions that are quite natural quite easily. For example,

- Higher-Order Functions
  - Functions that take functions as arguments and return functions as expressions
  - Used frequently
  - Currying
  - How to use effectively?
- Infinite Data Structures
  - Nice idea that is easily proven in functional languages
- Lazy evaluation (This is a function unique to Haskell)
  - Only evaluate expressions **ONLY WHEN NEEDED**
  - This also allow us to deal with idea of infinite data structures

1.1 Rewrite Semantics

One of the key strengths of Functional Programming Languages is the fact we can easily perform Rewrite Semantics on any given Expression.

Defn 3 (Rewrite Semantics)**.** *Rewrite semantics* is the process of rewriting and deconstructing an Expression into its constiuent parts. Rewrite semantics answers the question "How do we extract values from functions?"

Defn 4 (Expression)**.** An *expression* is a combination of one or more Operands and operators that the programming language interprets (according to its particular rules of precedence and of association) and computes to produce another value.

Remark 4.1 (Overloading). An expression can be overloaded if there is more than one definition for an operator.

Defn 5 (Operand)**.** An *operand* is a:

- Constant
- Variable
- Another Expression
- Result from function calls

```haskell
factorial 0 = 1 -- For argument 0, return 1
factorial n = n * factorial (n-1) -- For any other argument, return n * ((n-1) * ((n-1-1) * ... *
↪  (n-n)))

-- If we call factorial 3, what happens?
-- We can show what happens with REWRITE SEMANTICS
-- f 3 = 3 * f 2
--     = 3 * 2 * f 1
--     = 6 * 1 * f 0
--     = 6 * 1
--     = 6
```

Listing 1: Rewrite Semantics of a Factorial Function

1.2 Paradigm Differences

Functional programming is a completely different paradigm of programming than traditional imperative programming. One of the biggest differences is that **side effects are NOT allowed**.

1.2.1 Side Effects

Side effects are typically defined as being function-local. So, we can assign variables, make lists, etc. **so long as the effects are destroyed upon leaving the function**. Additionally, nothing globally usable can/should be changed.

```java
public int f(int x) {
        int t1 = g(x) + g(x);
        int t2 = 2 * g(x);
        return t1-t2;
}
// We should probably get 0 back.
// f(x) = t1-t2 = g(x) + g(x) - 2*g(x) = 0

// But, if g(x) is defined like so,
public int g(int x) {
        int y = input.nextInt();
        return y;
}
// The two instances of g(x) (g(x) + g(x)) can be different values,
// This invalidates the result we reached made earlier.
```

Listing 2: C-Like Code with Side Effects

1.2.2 Syntactic Differences

The = symbol has different meanings in Functional Programming Languages. In functional languages, =, is the mathematical definition of equivalence. Whereas in Imperative Programming Languages, = is the assignment of values to memory locations.

Typically, Functional Programming Languages do not have a way to directly access memory, since that is an inherently stateful change, breaking the rules of "side-effect free". However, "variables" **do** exist, but they are different.

- Variables are **NAMED** expressions, not locations in memory
- When "reassigning" a variable, the old value that name pointed to is discarded, and a new one created.

1.2.3 Tendency Towards Recursion

Most Functional Programming Languages use recursion more than they use iteration. This is possible because recursion can express all solutions that iteration can, but that does not hold true the other way around. Recursion is also intimately tied to the computability of an Expression.

Take the code snippet below as an example. It sums all values from a list of arbitrary size by taking the front element of the provided list (`x`) and adding that to the results of adding the rest of the list (`xs`) together.

```
sum1 [] = 0
sum1(x:xs) = x + (sum1 xs)
```

Listing 3: Basic List Summation

1.2.4 Higher-Order Functions

Similarly to what we defined in Listing 3, say we want to define the operations:

- Multiplying all elements together
- Finding if any elements are `True`.
- Finding if all the elements are `True`.

It would look like the code shown below. The code from Listing 3 will be included.

```
mySum [] = 0
mySum(x:xs) = x + (mySum xs)

myProd [] = 1
myProd (x:xs) = x * (myProd xs)

anyTrue [] = False
anyTrue (x:xs) = x || (anyTrue xs)

allTrue [] = True
allTrue (x:xs) = x && (allTrue xs)
```

Listing 4: List Comprehension Functions, No Higher-Order Functions Used

If you look at each of the functions, you will notice something in common between all of them.

- There is a default value, depending on the operation, for when the list is empty.
- There is an operation applied between the current element and,
- The rest of the list is recursively operated upon.

If we instead used a higher-order function, we can define all of those functions with just one higher-order function.

1.2.5 Infinite Data Structures

One of the benefits of lazy evaluation, and allowing higher-order functions, is that infinite data structures can be created. So, we could have a list of **all** integers, but we will not run out of memory (probably). Because of lazy evaluation, the values from these infinite data structures are computed **on when needed**.

For example, we find all prime numbers, starting with 2, using the Eratosthenes Sieve method (Listing 6). This method states we take **ALL** integers, starting from 2

1. Make a list out of them.
2. Take the first element out.
3. Remove all multiples of that number.
4. Put that number into a list of primes.
5. Repeat from step 2, until you find all the prime numbers you want.

In Haskell, this looks like:

```haskell
-- allElementsListFunction :: (t1 -> t2 -> t2) -> t3 -> [t1] -> t2
-- Takes a function that takes 2 things and spits out a third (t1 -> t2 -> t2)
-- Also takes a thing (t3)
-- Lastly, takes a list of thing t1 ([t1])
-- Returns a value of the same type as t2 (t2)
allElementsListFunction func initVal [] = initVal
allElementsListFunction func initVal (x:xs) = func x (allElementsListFunction func initVal xs)

-- After writing this function, when it is applied in the way below, each of the EXPRESSIONS,
-- my...2 is ALSO a function, which can be called.
mySum2 = allElementsListFunction (+) 0 -- Written this way, calling mySum2 is identical to calling
↪  sum1
myProd2 = allElementsListFunction (*) 1
anyTrue2 = allElementsListFunction (||) False
allTrue2 = allElementsListFunction (&&) True
-- Each operator provided (+, *, ||, &&) is a function in Haskell
-- I provided a function, and the initial value, so now each of these expressions is also a function.

-- Two lists for showing below
testIntList = [1, 2, 3, 4]
testBoolList = [True, True, False]

mySumTotal = mySum2 testIntList -- Returns 10
myProdTotal = myProd2 testIntList -- Returns 24
anyTrueTotal = anyTrue2 testBoolList -- Returns True
allTrueTotal = allTrue2 testBoolList -- Returns False
```

Listing 5: List Comprehension Functions, Higher-Order Functions Used

```haskell
-- The expression primes will NOT be computed until we ask the system to.
-- As soon as we do, it will be stuck in an "infinite loop", finding all primes
primes = sieve [2..] -- Infinite list of natural numbers, starting from 2
         where
           sieve (n:ns) =
               n : sieve [x | x <- ns, (x `mod` n) > 0]
```

Listing 6: Infinite Data Structure, All Primes by Eratosthenes Sieve

1.3 Language Basics

All of the functions and operations presented below come from **The Standard Prelude**. The library file *Prelude.hs* is loaded first by the REPL (Read, Evaluate, Print, Loop) environment that we will use. It defines:

- Mathematical Operations
- List Operations
- And other conveniences for writing Haskell.

1.3.1 Mathematical Operations

Prelude.hs defines the basic mathematical **integer** functions of:

- Addition
- Subtraction
- Multiplication
- Division
- Exponentiation

```
1  > 2+3
2  5
3  > 2-3
4  -1
5  > 2*3
6  6
7  >7 `div` 2
8  3
9  > 2^3
10 8
```

Listing 7: Integer Mathematical Operations

1.3.1.1 Precedences Just like in normal mathematics, there exists a precedence to disambiguate mathematical expressions containing multiple, different operations. In order of highest-to-lowest precedence:

1. Negation
2. Exponentiation
3. Multiplication and Division
4. Addition and Subtraction

1.3.1.2 Associativity Just like in normal mathematics, there are rules associativity rules to disambiguate mathematical expressions containing multiple of the same operations. There are only 2 types of associativity, left and right.

1. Left Associative:
   - Everything else.
   - Addition. $2 + 3 + 4 = (2 + 3) + 4$
   - Subtraction. $2 - 3 - 4 = (2 - 3) - 4$
   - Multiplication. $2 * 3 * 4 = (2 * 3) * 4$
   - Division. $2 \div 3 \div 4 = (2 \div 3) \div 4$
2. Right Associative:
   - Exponentiation. $2^{3^4} = 2^{(3^4)}$
   - Negation. $--2 = -(-2) = 2$

Remark (Types of Associativity). Technically, there are 3 types of associativity.

1. Left-Associative
2. Right-Associative
3. Non-Associative

Non-associativity means that it does not have an implicit associativity rule associated with it. It could also mean it is neither left-, nor right-associative.

1.3.2 List Operations

Prelude.hs also defines the basic list operations that we will need. To denote a list in Haskell, the elements are comma-delimited inside of square braces. For example, the mathematical list (set) of integers 1 to 3 $\{1, 2, 3\}$ is written in Haskell like so `[1, 2, 3]` .

Lists are a homogenous data structure. It stores several **elements of the same type**. So, we can have a list of integers or a list of characters but we can't have a list with both integers and characters. The most common list operations are shown below.

1.3.2.1 `head` Get the *head* of a list. Return the first element of a non-empty list. Remove all elements other than the first element. If the list **is** empty, then an Exception is returned. See Listing 8.

```
> head [1, 2, 3, 4, 5]
1
```

Listing 8: Haskell `head` Function

1.3.2.2 `tail` Get the *tail* of a list. Return the second through nth elements of a non-empty list. Remove the first element. If the list **is** empty, then an Exception is returned. See Listing 9.

```
> tail [1, 2, 3, 4, 5]
[2, 3, 4, 5]
```

Listing 9: Haskell `tail` Function

1.3.2.3 `last` Get the *last* element in a list. See Listing 10.

```
> last [1, 2, 3, 4, 5]
5
```

Listing 10: Haskell `last` Function

1.3.2.4 `init` Get the *init*ial portion of the list, namely all elements except the last one. See Listing 11.

```
> init [1, 2, 3, 4, 5]
[1, 2, 3, 4]
```

Listing 11: Haskell `init` Function

1.3.2.5 Selection, `!!` Select the nth element of a list. Lists in Haskell are zero-indexed. See Listing 12.

```
> [1, 2, 3, 4, 5] !! 2
3
```

Listing 12: Haskell `!!` Function

```
> take 3 [1, 2, 3, 4, 5]
[1, 2, 3]
```

Listing 13: Haskell `take` Function

1.3.2.6 `take` *Take* the first n elements of a list. See Listing 13.

1.3.2.7 `drop` *Drop* the first n elements of a list. See Listing 14.

```
> drop 3 [1, 2, 3, 4, 5]
[4, 5]
```

Listing 14: Haskell `drop` Function

1.3.2.8 Appending Lists to Lists, `++` Append the second list to the end of the first list. See Listing 15.

```
> [1, 2, 3, 4] ++ [9, 10, 11, 12]
[1, 2, 3, 4, 9, 10, 11, 12]
```

Listing 15: Haskell `++` Function

Remark. Be careful of this function. It runs in $O(n_1)$-like time, where n_1 is the length of the first list.

1.3.2.9 Constructing Lists, `:` To construct lists, they need to be composed from single expressions. This is done with the `cons` function. See Listing 16.

```
> 8:[1, 2, 3, 4]
[8, 1, 2, 3, 4]
```

Listing 16: Haskell `:` Function

The `cons` function is right-associative.

1.3.2.10 `length` To get the *length* of a list, use Listing 17.

```
> length [1, 2, 3, 4, 5]
5
```

Listing 17: Haskell `length` Function

1.3.2.11 `sum` The *sum* function is used to find the sum of all elements in a list. See Listing 18.

1.3.2.12 `product` The *product* function is used to find the product of all elements in a list. See Listing 19.

1.3.2.13 `reverse` The *reverse* function is used to reverse the order of the elements in a list. See Listing 20.

```
> sum [1, 2, 3, 4, 5]
15
```

Listing 18: Haskell `sum` Function

```
> product [1, 2, 3, 4, 5]
120
```

Listing 19: Haskell `product` Function

```
> reverse [1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
```

Listing 20: Haskell `reverse` Function

1.3.3 Function Application

Like in mathematics, functions can be used in expressions, and are treated as first-class objects. This means they have the same properties as regular variables, for almost all intents and purposes. For example, the equation

$$f(a, b) + cd$$

would be translated to Haskell like so

```
f a b + c * d
```

To ensure that functions are handled in Haskell like they are in mathematics, they have the highest precedence in an expression. This means that

```
f a + b
```

means

$$f(a) + b$$

in mathematics.

Table 1.1 illustrates the use of parentheses to ensure Haskell functions are interpretted like their mathematical counterparts.

Mathematics	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f\big(g(x)\big)$	`f (g x)`
$f\big(x, g(y)\big)$	`f x (g x)`
$f(x)g(y)$	`f x * g y`

Table 1.1: Parentheses Used with Functions

Note that parentheses are still required in the Haskell expression `f (g x)` above, because `f g x` on its own would be interpreted as the application of the function `f` to two arguments `g` and `x`, whereas the intention is that `f` is applied to one argument, namely the result of applying the function `g` to an argument `x`.

1.3.4 Haskell Files/Scripts

New functions can be defined within a script, a text file comprising a sequence of definitions. By convention, Haskell scripts usually have a `.hs` file extension on their filename.

If you load a script into a REPL environment, the *Prelude.hs* library is already loaded for you, so you can work with that directly. To load a file, you use the `:load` command at the REPL. Once loaded, you can call all the functions in the script at the REPL line.

If you edit the script, save it, and want your changes to be reflected in the REPL, you must `:reload` the REPL.

Some basic REPL commands are shown in Table 1.2

Command	Meaning
`:load` *name* or `:l` *name*	Load script *name*
`:reload` or `:r`	Reload the current scripts
`:type` *expr* or `:t` *expr*	Show the type of *expr*
`:?`	Show all possible commands
`:quit` or `:q`	Quit the REPL

Table 1.2: Basic REPL Commands

1.3.4.1 Naming Conventions There are some conventions and requirements when it comes to naming expressions in Haskell.

Function Naming Conventions Functions ***MUST***:

- Start with a **LOWER**-case letter
- Every subsequent character in the name can be upper-, lower-case, a number, underscores, or single quotes (`'`).

In addition, when naming your arguments to your functions:

- Numbers should get `n`.
- Characters should get `c`.
- Arbitrary values should get `x`.
- Lists should get `?s`, where `?` is the type of the list.
- Lists of lists should get `?ss`.

Type Naming Conventions When defining a type, there are rules similar to functions. However types ***MUST***,

- Start with an **UPPER**-case letter
- Every subsequent character in the name can be upper-, lower-case, a number, underscores, or single quotes (`'`).

List Naming Conventions By convention, list arguments in Haskell usually have the suffix s on their name to indicate that they may contain multiple values. For example, a list of numbers might be named `ns`, a list of arbitrary values might be named `xs`, and a list of list of characters might be named `css`.

1.3.5 Language Keywords

The following list of words have a special meaning in the Haskell language and cannot be used as names of functions or their arguments.

`case`	`class`	`data`	`default`	`deriving`	`do`	`else`
`if`	`import`	`in`	`infix`	`infixl`	`infixr`	`instance`
`let`	`module`	`newtype`	`of`	`then`	`type`	`where`

Table 1.3: Haskell Language Keywords

```haskell
a = b + c
    where
      b = 1
      c = 2

d = a * 2
```

1.3.6 The Layout Rule

The Layout Rule states that each definition must begin in precisely the same column. This layout rule makes it possible to determine the grouping of definitions from just their indentation.

For example,

It is clear from the indentation that `b` and `c` are local definitions for use within the body of `a`.

1.3.7 Comments

Haskell has 2 types of comments, like C-like languages.

1. From that point to the end of the line. Denoted with `--`.
2. Nested/multiline comments exist between the curly braces, `{- This is in the comment. -}` .

2 Constructing Functions

The most straight-forward way of constructing functions is to use functions that are already provided. For example, to define reciprocation of an integer or rational number, we would write:

```haskell
recip n = 1 / n
-- The / symbol is a function that is allowed to use infix notation.
```

Listing 21: Define Function From Others

2.1 Conditional Expressions

Defn 6 (Conditional Expression)**.** A *conditional expression* is one that chooses a path of execution based on some predicate/condition. In most languages, this is shown with the `if-then-else` structures.

Because in Haskell, we have Conditional Expressions, rather than a conditional statement, there must be a type for the expression. To ensure that these conditional expressions can be typechecked:

> **All possible options MUST have the same type.**

So, the first function in Listing 22 would **NOT** be compilable, because of a type error. The second would compile.

```haskell
failCompile n = if n < 0 then n else True
-- THIS WILL FAIL TO COMPILE
-- BOTH branches need to have the same type

willCompile n = if n < 0 then n else -n
```

Listing 22: Example Conditional Expression

In addition, **every `if` MUST** have a paired `else`.

2.2 Guarded Equations

Defn 7 (Guarded Equation)**.** A *guarded equation* is one in which a series of Guards are used to choose between a sequence of results that all have the same type.

Defn 8 (Guard)**.** A *guard* is a conditional predicate that is used to construct Guarded Equations. The guarding predicate is denoted with a `|` and is read as "such that".

Using Guarded Equations to define functions is an alternative to the use of Conditional Expressions. The benefit of using Guarded Equations is that functions with multiple Guards are easier to read.

In Guarded Equations, just like in Conditional Expressions, all possible options **MUST** have the same type.

An example of the same absolute value function from Listing 22 is shown in Listing 23.

```haskell
abs n | n >= 0 = n
      | otherwise = -n
-- otherwise is a special guard that is always true.
-- It can be thought of as the "default" option.
```

Listing 23: A Guarded Equation in Haskell

The use of `otherwise` in Listing 23 denotes a "default" case. If none of the previous Guards apply to the argument given to the function, then the `otherwise` option is chosen. One thing to note is that the Guards are checked in the order they are written. So, if `otherwise` appears before the end of the function, it will be matched early.

```haskell
abs2 n | otherwise = -n
       | n >= 0 = n
```

Listing 24: A Guarded Equation with Early Matching

2.3 Pattern Matching

By using pattern matching, many sequences of results can be chosen quickly and easily.

Like the others, Conditional Expressions, and Guarded Equations, each option **MUST** have the same type.

Patterns are matched in the order they are written. So if what was given matches the first pattern, the first option is taken. If what was given matches the second pattern, that option is taken, etc.

To define a pattern matching operation, there is **NO** special symbol required. All you have to do is give the function name again, the next pattern to match against, and the action to take (resulting in the same type).

This allows us to define functions in a third way.

To make pattern matching even easier, we are given acess to a wildcard pattern, `_`, which matches any value. By using this, you are also giving up the ability to reference that value in your function. The use of pattern matching on more than one parameter and using wildcards to simplify the function is shown in Listing 26.

The same name may not be used for more than one argument in a single pattern. Thus, in that third example, we could not use `b` for both parameters. However, a way around that is to use 2 different arguments, and then use a Guard to ensure the arguments are the same.

```haskell
myNot :: Bool -> [Char]
myNot False = "True"
myNot True = "False"
```

Listing 25: Basic Pattern Matching in Haskell

```haskell
newAnd :: Bool -> Bool -> [Char]
newAnd True True = "True"
newAnd True False = "False"
newAnd False True = "False"
newAnd False False = "False"

-- We can make the definition of newAnd even better.
newAnd' :: Bool -> Bool -> [Char]
newAnd' True True = "True"
newAnd' _ _ = "False"

-- Both of these definitions are functionally equivalent.

-- The logical and operator is actually implemented like so, shown below.
realAnd :: Bool -> Bool -> Bool
realAnd True b = b -- We can use b throughout this pattern to reference the second argument.
realAnd False _ = False
```

Listing 26: Pattern Matching with Multiple Parameters and Wildcards

2.3.1 Tuple Patterns

A tuple of patterns is itself a pattern, which will match any tuple of the same arity, whose elements all match the corresponding patterns, in order.

The code to select the first, second, and third elements of a triple tuple can be seen in Listing 27.

```haskell
first (x, _, _) = x
second (_, y, _) = y
third (_, _, z) = z
```

Listing 27: Tuple Pattern Matching

2.3.2 List Patterns

Similarly to tuple pattern matching, a list of patterns is itself a pattern, which matches any list of the same length whose elements all match the corresponding patterns in order. For example, a function test that decides if a list contains precisely two characters beginning with 'z' can be defined as follows:

2.3.3 Integer Patterns

As a special case that is sometimes useful, Haskell also allows integer patterns of the form $n+k$, where n is an integer variable and $k > 0$ is an integer constant. There are two points to note about $n + k$ patterns.

1. They only match integers $\geq k$.
2. For same reason as `cons`/list patterns, integer patterns must be parenthesised.

```haskell
-- The 'z':_ MUST be written in parentheses because function application has the
-- highest precedence. If they weren't there, then the function would be interpreted
-- as (test 'z'):_ which makes no sense.
test ('z': _) = True
test _ = False -- ANYTHING else must be False, by our definition of the function
```

Listing 28: List Pattern Matching

2.4 Lambda Expressions

Defn 9 (Lambda Expression)**.** *Lambda expressions* are an alternative to defining functions using equations. Lambda expressions are made using:

- A pattern for each of the arguments.
- A body that specifies how the result can be calculated in terms of the arguments.
- But do not give a name for the function itself.

In other words, lambda expressions are nameless functions.

The use of Lambda Expressions comes from the invention of Lambda Calculus. These are typically represented with the lower-case Greek letter λ on paper.

In Haskell, Lambda Expressions are written as seen in Listing 29.

```haskell
-- Lambda expression for adding 1 to a provided argument.
\ x -> x + 1

-- As these are also technically functions
(\ x -> x + 1) 1 -- Evaluates to 2.
```

Listing 29: Lambda Expressions in Haskell

Lambda Expressions are useful for several reasons.

1. They can be used to formalise the meaning of curried function definitions.
2. They are useful when defining functions that return functions as results by their very nature, rather than as a consequence of currying.
3. Can be used to avoid having to name a function that is only referenced once.

2.5 Sections

Defn 10 (Section)**.** A *section* is a way of writing expressions as infix or prefix with some number of pre-provided arguments. In general, if $\oplus$ is an operator, then expressions of the form $(\oplus)$, $(x\oplus)$, and $(\oplus y)$ for arguments x and y are called sections, whose meaning as functions can be formalised using lambda expressions as follows:

$$(\oplus) = \lambda x \to (\lambda y \to x \oplus y) \tag{2.1a}$$

$$(x\oplus) = \lambda y \to x \oplus y \tag{2.1b}$$

$$(\oplus y) = \lambda x \to x \oplus y \tag{2.1c}$$

Sections have 3 main applications:

1. They can be used to construct a number of simple but useful functions in a particularly compact way, as shown in the following examples:
   - $(+)$ is the addition function $\lambda x \to (\lambda y \to x + y)$
   - $(1+)$ is the successor function $\lambda y \to 1 + y$
   - $(1/)$ is the reciprocation function $\lambda y \to 1/y$
   - $(*2)$ is the doubling function $\lambda x \to x * 2$

- $(/2)$ is the halving function $\lambda x \to x/2$

2. Sections are necessary when stating the type of operators, because an operator itself is not a valid expression in Haskell
3. Sections are also necessary when using operators as arguments to other functions.

3 List Comprehensions

List comprehensions allow for functions on lists to be defined in a simple, relatively natural manner. The syntax for these is largely drawn from mathematics and set operations.

3.1 Generators

In mathematics, the comprehension notation can be used to construct new sets from existing sets. For example, to get the set squares from one to five is written like this in mathematics.

$$\{1, 4, 9, 16, 25\} = \{x^2 | x \in \{1 \dots 5\{\} \tag{3.1}$$

Equation (3.1) would be said to contain all numbers x^2 where each x is an element of the set $\{1 \dots 5\}$. In Haskell, this same thing would be written as

```
> [x^2 | x <- [1..5]]
[1, 4, 9, 16, 25]
```

Listing 30: Haskell List Comprehensions

Defn 11 (Generator)**.** A *generator* is an expression that generates all possible values in a set. In the cases above, `x <- [1..5]` is a generator.

A generator will "drop down one list level", so a generator like `xs <- xss` where `xss :: [[T]]` will make `xs :: [T]` . So, you can apply a generator an arbitrary number of times to "go down list levels".

In addition to the usual usage of Generators to get values out of a list/set, the wildcard symbol `_` can be used too.

```
listLength :: [a] -> Int
listLength xs = sum [1 | _ <- xs] -- _ here means we don't care what is in xs
{- Rather, all we care about is being able to GO THROUGH the list itself, and
   create another list of ALL 1s, then sum that list up, getting a single int
   that represents the length of the input list.
 -}
```

Listing 31: Wildcard Generator

3.1.1 Multiple Generators

Multiple generators can be defined together using a comma between them. See Listing 32 below.

```
[(x, y) | x <- [1, 2, 3], y <- [1, 2, 3]]
-- [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]

[(x, y) | y <- [1, 2, 3], x <- [1, 2, 3]]
-- [(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3, 3)]
```

Listing 32: Multiple Generators

3.1.2 Dependent Generators

Lastly, more deeply nested (Generators that are further to the right of the |) Generators can depend on earlier ones. In Listing 33, we generate the same lists as in Listing 32, but having the second Generator being dependent on the first.

```haskell
[(x, y) | x <- [1..3], y <- [x..3]]
-- [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]

[(x, y) | y <- [1..3], x <- [y..3]]
-- [(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3, 3)]
```

Listing 33: Dependent Generators

3.2 Guards

If you recall from Section 2.2, we used Guards to create functional equations that were carried out based on the argument provided to the function. We can do the same thing here in list comprehensions. In a list comprehension, if a guard is `True`, then the current values are retained and put in the new list; if it is `False`, then the value is discarded.

> A guard can restrict values made by earlier Generators.

In addition, because of the lazy evaluation of Haskell, many operations that would normally be very expensive are not too costly. Take for example, generating all the primes up to some large value, as seen in Listing 34.

```haskell
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
{- Returns the list of all numbers from 1 to n that when divided
   by x, have a modulo of 0, i.e. they are factors of n.
   They CANNOT be multiples of n, because that would require us to go past
   the n provided.
 -}

prime :: Int -> Bool
prime n = factors n == [1,n] -- [1,n] is a list equality of 2 elements, 1 and n
{- Because of lazy evaluation, the False result is returned as soon as ANY
    factor other than one or the number itself is produced.
   In addition, if factors n produces more than 2 elements, then the
    evaluation is automatically False
 -}

-- Using the 2 above functions, we can now generate all primes up to some value
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

Listing 34: Guarded List Comprehension for Prime Generation

3.3 The `zip` Function

The library function `zip` produces a new list by pairing successive elements from two existing lists until either or both are exhausted. This means that the list returned by `zip` will always be limited by the shortest list provided. The `zip` function

is often useful when programming with list comprehensions.

$$listOfXs = [x_1, x_2, x_3, \ldots]$$
$$listOfYs = [y_1, y_2, y_3, \ldots]$$
$$\texttt{zip}listOfXslistOfYs = [(x_1, y_1), (x_2, y_2), \ldots]$$

An example here is determining if the elements in a list are sorted, in ascending value.

```haskell
-- First, let's pair each of the elements up together
pairs :: [a] -> [(a, a)]
pairs xs = zip xs (tail xs)

-- Now that we can generate a list of pairs, we can now pairwise compare them
sorted :: Ord a=> [a] -> Bool -- Here, we require that the type a be part of the Ord typeclass
sorted xs = and[x <= y | (x, y) <- pairs xs]
-- A list of booleans are returned by the list comprehension
-- The and goes through and logically and's them together.
-- If any single element is false, the whole thing becomes false.
```

Listing 35: Using the `zip` Function

3.4 String Comprehensions

Strings in Haskell are nothing but lists of characters (there is no null terminator in the list).

For example, `"abc"::String` is just an abbreviation for `['a', 'b', 'c']::[Char]` . Because strings are just special kinds of lists, any polymorphic function on lists can also be used with strings.

```haskell
"abcde" !! 2
-- 'c'

take 3 "abcde"
-- "abc"

length "abcde"
-- 5

zip "abc" [1, 2, 3, 4]
-- [('a', 1), ('b', 2), ('c', 3)]
```

Listing 36: Polymorphic List Comprehensions Used on Strings

For the same reason, list comprehensions can also be used to define functions on strings.

4 Recursion

In Haskell, there is **no** concept of iteration. By extension, this also means that there is no `while`, `for`, and/or `do-while` looping structures. Instead of using iteration, Recursion is used instead.

Defn 12 (Recursion)**.** *Recursion* is the process of defining something in terms of itself, typically in a smaller amount/value. By "slightly" reducing the size of the problem through each recursive "iteration", the problem domain is smaller. Eventually, the problem domain becomes small enough that the solution is simple. This is called the *base case*. Once the simple solution has been found, then all recursions can start building from that solution.

```haskell
-- Counts the number of lowercase characters in a String
countLowers :: String -> Int
countLowers xs = length [x | x <- xs, isLower x]
-- isLower returns True/False if x is lowercase
countLowers "Haskell"
-- 6

-- Count the number of a chosen letter present in a given String
countLetter :: Char -> string -> Int
countLetter x xs = length [x' | x' <- xs, x==x']
countLetter 's' "Mississippi"
-- 4
```

Listing 37: List Comprehensions Used on `String` s

Remark 12.1 (Recursive). A function that uses recursion is said to be *recursive*.

The reason that Recursion is used instead of iteration is for multiple reasons.

- Recursion can compute everything iteration can.
- Recursion is inductively provable.
- Recursion is side-effect free.
- Recursion is a natural way to work with algebraic datatypes, lists, and the infinite structures of Functional Programming.

One of the first Recursive programs most people write is the factorial function.

```haskell
factorial 0 = 1 -- Base case
factorial n = n * factorial (n-1)
```

Listing 38: Factorial, Recursively Defined

Sometimes library functions simplify the way a function is written, but many function also have simple and natural definitions using Recursion.

Remark. Note that throughout this section, we are regularly writing functions with the same name as functions included in the `Prelude.hs` library file. To prevent possible namespace collisions, functions I am writing that have the same name as `Prelude.hs` library functions will be named slightly differently, typically preceeded with `my`.

Remark. Throughout this section, many Recursive functions will be created, many of them quite inefficient. However, they are written this way to emphasize clarity and convey what recursion is and how to write recursive structures.

4.1 List Recursion

Lists, being potentially infinite, and composed of a single element and the rest of the list (see Constructing Lists, `:` , Paragraph 1.3.2.9), make for natural targets of Recursion.

In every case of list recursion, you need to identify the base case. Typically, this is the empty list. If you wanted to find the product of a list of numbers, you could use the recursive definition shown in Listing 39.

```haskell
myProduct :: Num a => [a] -> a -- Num a makes the requirement that a be in the number typeclass
myProduct [] = 1 -- Because 1 is the identity value for multiplication, that should be the base value
myProduct (x:xs) = x * myProduct xs
```

Listing 39: Product of a List

In addition, you can use the _ wildcard in your list recursion as well, if you don't actually care what that particular element's value is.

```haskell
myLength :: [a] -> Int
myLength [] = 0 -- The empty list has length 0
myLength (_:xs) = 1 + myLength xs
-- As we don't care about the actual value of x here, we can use the wildcard "_"
```

Listing 40: Length of a List

4.2 Multiple Arguments

Just like normal functions, there is no reason why we need to limit ourselves to a single argument on recursive functions. This allows us to do things like, define our own `zip` function.

```haskell
myZip :: [a] -> [b] -> [(a, b)]
myZip [] _ = [] -- Base case 1: List a is empty -> Return empty list
myZip _ [] = [] -- Base case 2: List b is empty -> Return empty list
myZip (x:xs) (y:ys) = (x,y):(myZip xs ys)
-- Otherwise, make a tuple out of the current elements, and recurse
-- the rest of the way through the list.
```

Listing 41: Multi-Argument Recursion of Haskell's `zip` Function

Keep in mind, the more parameters you give the function, the more base cases you **may** need to add. In Listing 41, you need 2 base cases, because either of the 2 lists may be empty, indicating we should end our `zip` ping procedure. Another good example of multi-argument recursion that requires more than one base case is the `drop` library function (Paragraph 1.3.2.7).

```haskell
myDrop :: Int -> [a] -> [a]
myDrop 0 xs = xs -- Base case 1, we want to remove 0 elements, so return the unmodified list
myDrop n [] = [] -- Base case 2, we want to remove n more elements, but there are none left.
                 -- So, we stop recursing and return the empty list.
myDrop n (_:xs) = myDrop (n-1) xs
-- Otherwise, we want to keep dropping elements, and still have some list left.
```

Listing 42: Multi-Argument Recursion of Haskell's `drop` Function

4.3 Multiple Recursion

Defn 13 (Multiple Recursion)**.** *Multiple recursion* is when a functions is applied more than once in its own definition.

Remark 13.1 (Multiply Recursive). A function that uses Multiple Recursion is said to be *multiply recursive*.

The first multiply recursive function most people write is a function to calculate the Fibonacci sequence.

Multiply recursive functions may require more than one base case to terminate the recursion, depending on how the function works.

4.4 Mutual Recursion

Defn 14 (Mutual Recursion)**.** *Mutual recursion* is when a set of two or more functions are defined in terms of each other.

```haskell
-- Mathematical definition of Fibonacci's Sequence. This is not an efficient implementation.
fibonacci :: Int -> Int
fibonacci 0 = 0 -- Base case 1, since sometimes we subtract by 2
fibonacci 1 = 1 -- Base case 2
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Listing 43: Multiple Recursion in Fibonacci's Sequence

Remark 14.1 (Mutually Recursive). A set of functions that use Mutual Recursion are said to be *mutually recursive*.

The easiest mutually recursive functions to visualize are the `odd` and `even` functions. As a side note, these functions are not actually implemented this way, for efficiency's sake.

```haskell
myEven :: Int -> Bool
myEven 0 = True -- 0 IS even
myEven n = myOdd (n-1) -- If n is even, then taking 1 away will make it odd.

myOdd :: Int -> Bool
myOdd 0 = False
myOdd n = myEven (n-1) -- If n is odd, then taking 1 away will make it even.
```

Listing 44: Mutually Recursive `odd` and `even` Functions

4.5 Help With Recursion

Defining Recursive functions may seem easy, but it can be quite difficult. Here, I will include a list of steps that may make it easier to think about and write new recursive functions.

1. Define the type of the function. Section 4.5.1
2. Enumerate the cases. Section 4.5.2
3. Define the Simple cases, The Base Cases. Section 4.5.3
4. Define the other cases. Section 4.5.4
5. Generalize and Simplify. Section 4.5.5

4.5.1 Define Function Type

When first starting your function definition, think about what you want this function to do.

- What type do you want it to take in?
- What type should it output?
- What should be returned by the function?

Do not worry about generalizing the types of the function for now, that can come once you've written the thing.

4.5.2 Enumerate Cases

What are the cases you can have? You will need to have one or more base cases, to eventually terminate the recursion. You also need cases that slowly break your information down into smaller pieces that you can perform some action on.

For example, with lists, there are 2 general cases:

1. Empty Lists — Base case
2. Non-empty Lists — Other Case(s)

4.5.3 Define Simple Cases

In the simple cases, there is usually an obvious solution to the problem at hand. For example, if you are writing the `product` function for lists, the base case would be the empty list. In that case, you would simply return 1, because that is the multiplicative identity.

4.5.4 Define Other Cases

Here is where some of the most difficult work can happen. When you are **NOT** in the base case, what do you want to do? Additionally, how do you eventually get down to the base case?

4.5.5 Generalize/Simplify

Once you have finished writing the recursive function, and tested it to make sure it works, you can clean up your work. You can start by generalizing the type, if possible. For example, if a recursive function is acting on a list, but does not use the information within the list, then the type can be generalized. If the function started with a type signature

```
1 [Int] -> Int
```

but you never use the contents of the list, then you should be able to generalize the function to

```
1 [a] -> Int
```

Additionally, if you should also look at your function to see if a library function can achieve many of the same things as your custom recursion does. For example, if you want to apply a function to every element in a list, and you've written your own implementation, switch to the `map` function instead.

5 Types and Typeclasses

Defn 15 (Type)**.** A *type* is a collection of related values. For example, the `Bool` type contains 2 values, `True` and `False`. The definition of a type limits the types and amount of values that an expression of that type can take.

In Haskell, to denote the Type of an expression, you use the `::` symbol. Namely, `e :: T` meaning that the Expression `e` "has type" `T`, or `e` "is of type" `T`.

> **EVERY Expression MUST HAVE A Type!**

To ensure that every Expression has a Type before execution, Haskell (like most other functional languages) uses Type Inferencing to attempt to determine the type of every expression.

Defn 16 (Type Inferencing)**.** *Type inferencing* is the process of solving the system of type equations introduced by expressions and operators/functions used on them to determine the Type of every Expression in a program.

When an operator or function is used, it has a type signature. This indicates what Types it takes in and what type(s) it returns. If there are operators/functions with pre-determined type signatures, like addition, then the types of the expressions that are fed into the addition operator need to be something that can be added. After the language system has gone through the whole file/program, a system of equations for the type of every expression can be created. By solving this system of equations, the Type of every expression can be determined.

Remark 16.1 (Done Statically). It is important to note that Type Inferencing is done **statically**, meaning it is done before the program is even executed. When compiled, the Types are determined during compilation. When loaded into the REPL, the Types are determined when the file is parsed and interpreted.

Remark 16.2 (Doesn't Catch Everything). Type Inferencing does **NOT** catch all possible errors. Namely, it does not catch runtime errors, such as division-by-zero. However, it does remove a large category of errors that present themselves due to inappropriate typing.

Because Type Inferencing is done statically, Haskell programs are both *strongly typed* and *type safe*. Meaning all Type errors that could occur in a program are found before program execution. The use of Type Inferencing is the reason that **all execution paths must have the same type**.

Remark. The determination of Types in a program using Type Inferencing where selection statements have different Types is an undecidable problem. Thus, the language designers enforced programmers to write all Conditional Expression, Guarded Equations and pattern matching paths to have the same Type.

5.1 Basic Types

There are a variety of basic types built into the Haskell language.

5.1.1 `Bool`

The `Bool` Type contains the two logical values, `True` and `False`.

5.1.2 `Char`

The `Char` Type contains all the single characters that are available from a typical keyboard, including many control characters. Characters in Haskell must be enclosed beween forward quotes; for example, `'c'`.

5.1.3 `String`

The `String` Type is a Type Alias for `[Char]`. This means it also includes all the characters available from a typical keyboard, and the control characters, however more than one character can be present. Strings in Haskell are enclosed between double quotes; for example, `"string"`.

Remark. The statement about allowing more than one character be present is a little misleading. Since the `String` Type is actually `[Char]`, a list of single characters. Haskell just **shows** them to us nicely formatted.

5.1.4 `Int`

The `Int` Type represents fixed-precision integers, analogous to the signed integer system of traditional imperative languages. The number of bits (32 or 64) depends on your computer's architecture. On a 64-bit computer, 64-bit integers will be used, making the limits of the representable integers $[-2^{63}, 2^{63} - 1]$. Likewise, on a 32-bit computer, 32-bit integers will be used, setting the limits of the system in the interval $[-2^{31}, 2^{31} - 1]$.

5.1.5 `Integer`

The `Integer` Type represented arbitrary-precision integers. These are integers that can be **any possible value**, limited only by the memory capabilities of your computer. Thus, expressions of type `Integer` can be as big as your memory allows.

`Int` and `Integer` are also different in terms of their performance. Typically, `Int` will run faster because there is dedicated hardware to perform the computation, whereas `Integer`s will need to be handled in software as a sequence of digits.

5.1.6 `Float`

The `Float` Type represents single-precision floating-point numbers. Single-precision refers to the number of bits used to represent the number, in this case, 32. Floating-point refers to the ability to represent values like $\frac{1}{3}$, π, and any other value with a fractional portion.

5.2 List Types

Defn 17 (List)**.** In Haskell, a *list* is a sequence of elements that are **all the same Type**. The elements are enclosed within brackets, `[` and `]`, with each element separated by commas.

Listing 45 shows examples of lists and their synatax.

Below are some examples of lists.

```
1 [False, True, False] -- :: [Bool]
2 ['a', 'b', 'c', 'd'] -- :: [Char]
3 ["One", "Two", "Three"] -- :: [String]
4 [1, 2, 3] -- :: [Int]
```

Listing 45: Example of Lists in Haskell

The number of elements in a list is its length, where the empty list has length zero and does not contain any elements. The empty list is unique in that it can also be considered an element for other lists, allowing for the construction of lists using the `cons` operator `:`.

There are 3 major things to remember about List types:

1. The type of a list conveys no information about its length.
2. There are no restrictions on the type of the elements of a list, so long as they are all the same type.
3. There is no restriction that a list must have a finite length.

5.3 Tuple Types

Defn 18 (Tuple)**.** A *tuple* is a **finite** sequence of components/elements that may have different types. The elements are enclosed in parentheses and separated by commas.

Some examples of tuples are shown in Listing 46.

```haskell
(False, "True", False) -- :: (Bool, String, Bool)
('a', (2^8976), 'c', 3.14) -- :: (Char, Integer, Char, Float)
("One", "Two", "Three") -- :: (String, String, String)
(1, "2", '3') -- :: (Int, String, Char)
```

Listing 46: Example of Tuples in Haskell

The number of elements in a tuple is the *arity* of the tuple. The empty tuple has an arity of 0. A tuple of arity 2 is a pair, arity 3 is a triple, and so on.

Tuples of arity 1 are NOT allowed!!

There are 3 major things to remember about Tuple types:

1. The type of a tuple conveys its arity.
2. There are not restirctions on the type of the elements in a tuple, they do not even have to have the same type.
3. Tuples must **ALWAYS** have a finite arity, to ensure Type Inferencing works.

5.4 Function Types

Defn 19 (Function)**.** A *function* a mapping from arguments of one Type to another. The type signature of a function is written $\mathtt{T}_1 -> \mathtt{T}_2$.

Some examples of functions are shown in Listing 47.

Remark 19.1 (Pureness of Definition). This is the purest definition of a function, drawn right from mathematics. This interpretation forms the basis of Functional Programming Languages. This definition also prevents the introduction of side-effects, another basis for the developement of Functional Programming Languages.

```haskell
not :: Bool -> Bool
isDigit :: Char -> Bool

add :: (Int, Int) -> Int
add (x, y) = x + y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Listing 47: Example of Functions in Haskell

Type signatures can be provided for a function, although the language system will typically figure out the most generic Types possible for a function because of the Type Inferencing system. However, including them is good practice for documentation and for helping to write recursive functions.

The only way to handle **MULTIPLE** parameters passed into and returned from a single function is to put them into either a List or Tuple.

There is **NO RESTRICTION** for a function to be *total* on the arguments provided to a function, namely there may be some arguments for which the function is undefined.

5.5 Curried Function Types

Functions with multiple arguments can be handled differently, if we are allowed to Curry functions and use Higher-Order Functions. If we can use those, then we can have a function take a function as an argument as return a function as a result. So, instead of making `add` take 2 integers at once, we could instead give it just one, and that function will return another that adds the given value to the previously provided one. An example is shown in Listing 48

```haskell
curryAdd :: Int -> (Int -> Int)
curryAdd x = \y -> (x + y)

-- We can partially apply the curryAdd function now too.
add5 = curryAdd 5
-- add5 is now a function Int -> Int that adds 5 to any provided integer
val1 = add5 10 -- = 15
val2 = add5 (-5) -- = 0
```

Listing 48: Curried Version of Addition

There are some conventions regarding curried functions and their type signatures.

- The arrow in function types are *right associative*.
- The application of arguments to a function is done with spacing, and is assumed to associate to the left.
- Unless a function explicitly takes/returns a tuple as an argument/return value, functions are defined in their curried form.

5.5.1 The Power of Currying

All functions that could be expressed with tuple arguments can also be expressed by using functions that take functions as arguments and return functions as results. Some functions that return tuples as results cannot always be converted to return functions, as some tuples is actual data.

Currying functions is important to allow for Partial Application.

5.6 Polymorphic Types

In some cases, Functions can accept any Type as an argument. For example, the `length` function will find the length of any provided lists, regardless of the type of the elements in the list.

To represent this variability in the Type a function may have, we use a Type Variable.

Defn 20 (Type Variable)**.** A *type variable* is like a traditional variable, except instead of representing a variety of values, it represents a variety of Types. In Haskell, these are denoted with lowercase letters, typically just a single letter, in the **TYPE SIGNATURE**.

Remark 20.1 (Lowercase Letter Distinction)*.* The distinction between Type Variables and Expressions named with lowercase letters is important to make, because a lowercase letter is a type variable only when used in a type signature. If a lowercase letter is used anywhere else, it is considered an expression.

Some examples of functions that use type variables are shown in Listing 49.

These Type Variables are what make a function Polymorphic.

Defn 21 (Polymorphic)**.** A *polymorphic* "thing" is one that can be multiple Types. In Haskell's case, functions are the only item that can be polymorphic, thus creating polymorphic functions. These polymorphic functions are denoted by the Type Variable in their type signature.

Some examples of polymorphic functions are shown in Listing 49.

```haskell
length :: [a] -> Int
fst :: (a, b) -> a
snd :: (a, b) -> b
head :: [a] -> a
take :: Int -> [a] -> [a]
zip :: [a] -> [b] -> [(a, b)]
id :: a -> a
```

Listing 49: Polymorphic Functions Using Type Variables

5.7 Overloaded Types

Some Functions and operators are overloaded, allowing for multiple definitions of a function using the same symbol. The correct version of the function is chosen based on the Types of the arguments provided to the function. For example,

```haskell
> 1 + 2
3

> 1.1 + 2.2
3.3
```

To continue ensuring that only the correct types are still fed into the function, we use Typeclasses. These are also sometimes called Class Constraints. Some examples of Typeclasses/Class Constraints are shown in Listing 50.

```haskell
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a
negate :: Num a => a -> a
abs :: Num a => a -> a
signum :: Num a => a -> a
```

Listing 50: Overloaded Function Types Examples

In addition, the number literals are also overloaded, for example `3 :: Num a => a` .

5.8 Typeclasses

Defn 22 (Typeclass)**.** A *typeclass* is a way to group Types together and ensure that any Type in the same typeclass has the same operations available to it. This is a way to group together Types that support certain overloaded functions, called methods.

A single Type can belong to multiple typeclasses. Making typeclasses similar to interfaces from the OOP world.

Remark 22.1 (Class Constraint)*.* A *class constraint* is a way to constrain the Types that an expression can take. In this section, we discuss some of the basic Typeclasses that form these class constraints.

Remark 22.2 (Class vs. Typeclass)*.* In Haskell, Typeclasses are called *class*es. I deliberately call them a different name to make it clear that we are not using the term class like used in Object-Oriented Programming.

5.8.1 `Eq`

The `Eq` Typeclass contains Types that can be compared for equality and inequality. For a Type to be part of this class, they must provide the methods seen in Listing 51.

ALL the basic Types are part of this Typeclass, including lists and tuples (if their elements are instances of types in this typeclass). The only basic type that is **NOT** in this Typeclass are Functions, because there is no general way to compare two functions for equality.

```haskell
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Listing 51: `Eq` Typeclass Required Methods

5.8.2 `Ord`

The `Ord` Typeclass contains Types that can be compared for relative value. For a Type to be part of this class, they must provide the methods seen in Listing 52.

```haskell
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool

min :: a -> a -> a
max :: a -> a -> a
```

Listing 52: `Ord` Typeclass Required Methods

ALL the basic Types are part of this Typeclass, including lists and tuples (if their elements are instances of types in this typeclass). `String` s, lists, and tuples are ordered lexicographically, meaning they are sorted according to their first distinguishing element. The only basic type that is **NOT** in this Typeclass are Functions, because there is no general way to compare two functions for their values.

5.8.3 `Show`

The `Show` Typeclass contains Types that can be converted to a string of characters using the `show` function. For a Type to be part of this class, they must provide the methods seen in Listing 53.

```haskell
show :: a -> String
```

Listing 53: `Show` Typeclass Required Methods

ALL the basic Types are part of this Typeclass, including lists and tuples (if their elements are instances of types in this typeclass). The only basic type that is **NOT** in this Typeclass are Functions, because there is no general way to show the value of a function.

5.8.4 `Read`

The `Read` Typeclass contains Types whose values can be converted from strings of characters to values. For a Type to be part of this class, they must provide the methods seen in Listing 54.

```haskell
read :: String -> a
```

Listing 54: `Read` Typeclass Required Methods

ALL the basic Types are part of this Typeclass, including lists and tuples (if their elements are instances of types in this typeclass). The only basic type that is **NOT** in this Typeclass are Functions, because there is no general way to show the value of a function.

5.8.5 Num

The Num Typeclass contains Types whose values are numeric, and are part of the Eq and Show typeclasses. For a Type to be part of this class, they must provide the methods seen in Listing 55.

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
```

Listing 55: Num Typeclass Required Methods

Only Int , Integer , and Float are part of the Num Typeclass.

5.8.6 Integral

The Integral Typeclass contains Types whose values are numeric, and are part of the Eq and Show typeclasses **AND** are integral values. For a Type to be part of this class, they must provide the methods seen in Listing 56.

```
div :: a -> a -> a
mod :: a -> a -> a
```

Listing 56: Integral Typeclass Required Methods

Only Int and Integer are part of the Integral Typeclass. Also, the standard prelude list functions use Int instead of the Integral typeclass for performance reasons.

5.8.7 Fractional

The Fractional Typeclass contains Types whose values are numeric, and are part of the Eq and Show typeclasses **AND** are non-integral values. For a Type to be part of this class, they must provide the methods seen in Listing 57.

```
(/) :: a -> a -> a
recip :: a -> a
```

Listing 57: Fractional Typeclass Required Methods

Only Float are part of the Fractional Typeclass.

6 Higher-Order Functions

Higher-order functions allow common programming patters to be encapsulated as functions. This is why these are called functional programming languages, because their common abstraction/encapsulation pattern is done with functions.

Defn 23 (Higher-Order Function)**.** A function that takes a function as an argument **or** returns a function as a result is a *higher-order function.*

In Haskell, functions with multiple arguments are usually defined using the notion of currying. That is, the arguments are taken one at a time by exploiting the fact that functions can return functions as results. In addition, it is also permissible to define functions that take functions as arguments.

Remark. It was an important breakthrough to figure out that tradition functions that use tuples to pass in their arguments are equivalent to their curried counterparts.

Because of this unique property in Haskell (Haskell is not the only language that allows this), functions can be partially applied.

Defn 24 (Partial Application)**.** *Partial application* is when a curried function receives fewer parameters than it needs to fully compute a value. By only providing $m = n - x$ arguments (where $n > x$), the called function returns a function. This returned function can then have its last m parameters applied to compute something.

Remark 24.1 (Partially Applied)*.* Functions that have had Partial Application used on them are said to be *partially applied*.

The benefit of Higher-Order Functions are:

1. Common programming idioms from other languages can be encoded using functions in the language itself.
2. Domain-specific languages can be defined as collections of Higher-Order Functions.
3. Algebraic properties of Higher-Order Functions can be used to reason about them and programs as a whole.

6.1 List Processing

The standard `Prelude.hs` provides many Higher-Order Functions for working with lists in addition to the ones mentioned in Section 1.3.2.

Two of the most commonly used functions, `map` and `filter` are shown and described below. Other Higher-Order Functions are shown later in this section, namely `all`, `any`, `takeWhile`, `dropWhile`.

6.1.1 `map`

The `map` function applies a function to all elements in a given list. The list comprehension below shows one definition of the `map` function.

```
1 map :: (a -> b) -> [a] -> [b]
2 map f xs = [f x | x <- xs]
```

Listing 58: `map` Defined with a List Comprehension

With this definition of `map`, which may not be its actual implementation, we can use it like so.

```
1 > map (+1) [1, 3, 5, 7]
2 [2, 4, 6, 8]
3
4 > map isDigit ['a', '1', 'b', '2']
5 [False, True, False, True]
6
7 > map reverse ["abc", "def", "ghi"]
8 ["cba", "fed", "ihg"]
```

Listing 59: Simple Usages of the `map` Function

There are 3 more things to note about `map`:

1. It is a polymorphic function that can be applied to lists of any type, as are most higher-order functions on lists.
2. It can be applied to itself to process nested lists. For example, the function map (map (+1)) increments each number in a list of lists of numbers, as shown in Listing 60.
3. The `map` function can also be defined using recursion, as seen in Listing 61. This is the preferable definition of `map` because it is easier to reason about and extend.

```haskell
map (map (+1)) [[1, 2, 3], [4, 5]]
-- [[2, 3, 4], [5, 6]]
```

Listing 60: Nested Application of the `map` Function

```haskell
map f [] = []
map f (x:xs) = f x : map f xs
```

Listing 61: Recursive Definition of the `map` Function

6.1.2 `filter`

The `filter` function selects all elements of a list that satisfy a given predicate/property. This predicate is a function that returns a logical value, `True` / `False`. Like `map`, `filter` is another function with 2 definitions, one based on list comprehensions and one based on recursion.

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

Listing 62: `filter` Defined with List Comprehension

```haskell
filter p [] = []
filter p (x:xs) | p x = x:filter p xs -- Predicate is true. Keep current element, recurse
                | otherwise filter p xs -- The predicate is false. Drop current element.
```

Listing 63: `filter` Defined with Recursion

In words, Listing 63 when selecting elements, all elements that satisfy a predicate from the empty list gives the empty list, while for a non-empty list the result depends upon whether the head satisfies the predicate.

Some simple examples of using `filter` are shown in Listing 64.

```haskell
filter even [1..10]
-- [2, 4, 6, 8, 10]

filter (>5) [1..10]
-- [6, 7, 8, 9, 10]

filter (/= ' ') "abc def ghi"
-- "abcdefghi"
```

Listing 64: Simple Usage of the `filter` Function

6.1.3 `all`

`all` is a Higher-Order Function that decides of **ALL** the elements of a list satisfy a given predicate. This technically works

by taking a list of boolean values and performing the logical AND operation on all elements. If any single element is `False`, then the entire calculation becomes `False`.

```
all even [2, 4, 6, 8]
-- True
{- This function is actually the application of `even` to every element, which
   constructs a list of the same size of only Boolean values.
   That list is then logically ANDed together.
 -}
```

Listing 65: Simple Usage of the `all` Function

6.1.4 `any`

`any` is a Higher-Order Function that decides of **ANY** the elements of a list satisfy a given predicate. This technically works by taking a list of boolean values and performing the logical OR operation on all elements. If any single element is `True`, then the entire calculation becomes `True`.

```
any odd [2, 4, 6, 8]
-- False
{- This function is actually the application of `odd` to every element, which
   constructs a list of the same size of only Boolean values.
   That list is then logically ORed together.
 -}
```

Listing 66: Simple Usage of the `any` Function

6.1.5 `takeWhile`

`takeWhile` compares each element in a list against a given predicate. The function will continue to take elements from the given list and construct another list **so long as that predicate remains** `True`.

```
takeWhile isLower "abc def"
-- "abc"
{- Technically, the space character ' ' is NOT a lowercase character, so we stop
   taking values from the String/Char list once we reach that.
 -}
```

Listing 67: Simple Usage of the `takeWhile` Function

6.1.6 `dropWhile`

`dropWhile` compares each element in a list against a given predicate. The function will continue to drop elements from the given list **so long as that predicate remains** `True` and will return whatever is left of the original list after the predicate becomes `False`.

```haskell
dropWhile isLower "abc def"
-- " def"
{- Technically, the space character ' ' is NOT a lowercase character, so we drop
   all the lowercase values from the String/Char list UNTIL we reach that.
 -}
```

Listing 68: Simple Usage of the `dropWhile` Function

```haskell
f [] = v
f (x:xs) = x ? f x
-- ? is SOME operator that is applied to the ehad of the list and the result of
-- Recursively processing the tail of the list.
```

Listing 69: The Basis for Defining the `foldr` Function

6.2 The `foldr` Function

Many functions that have a list as an argument can be defined by the recursion pattern in Listing 70.

`foldr` is an abbreviation for "fold right". What this really means is that the operator being applied is assumed to be right-associating. In practice, however, it is best to think of the behaviour of `foldr f v` in a non-recursive manner. Instead, `foldr` replaces every `cons` in a list by the function `f`, and the empty list at the end by the value `v`. So, for example `foldr (+) 0 [1, 2, 3]` becomes $(1 + (2 + (3 + 0)))$ (The 0 is for the empty list).

The definition of `foldr` , like many other Higher-Order Functions, is recursive.

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
{- foldr takes:
   1) A function, which itself takes 2 parameters and returns one value. (+, for example)
   2) An initial value, for the base case of the list being empty
   3) The list to operate on.
 -}
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Listing 70: The Basis for Defining the `foldr` Function

The general operation of the `foldr` function is shown in Equation (6.1).

$$foldr(\oplus)v[x_0, x_1, x_2, \ldots, x_n] = x_0 \oplus (x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus v) \cdots))) \tag{6.1}$$

The benefits of using the `foldr` function are:

1. Some recursive functions on lists are simpler to define using `foldr` .
2. Properties of functions defined using `foldr` can be proved using its algebraic properties, such as *fusion* and *the banana split rule*.
3. Advanced program optimizations can be made simpler if `foldr` is used instead of explicit recursion.

6.3 The `foldl` Function

In addition to being able to define operations that are assumed to be right-associative, we can write functions that are assumed to be left-associative with the `foldl` function. The general basis for which the `foldl` function was written is shown in Listing 71.

The function maps the empty list to the accumulator value v. Any non-empty list is mapped to the result of recursively processing the tail using a new accumulator value obtained by applying an operator `?` to the current value and the head of the list.

```haskell
f v [] = v
f v (x:xs) = f (v ? x) xs
-- Where ? is some operator
```

Listing 71: The Basis of Defining the `foldl` Function

Many of the operations that The `foldr` Function defines can also be defined using `foldl` . When this is the case, the only way to choose which to use is based on the efficiency of the function itself. This is dependent on Haskell's underlying mechanisms, which is discussed later. For now, both of them work equally well.

In practice, just as with `foldr` it is best to think of the behaviour of `foldl` in a non-recursive manner. It is better to think of `foldl` , in terms of an operator $\oplus$ (Was `?` earlier. The actual symbol doesn't matter.) that is assumed to associate to the left, as summarised by Equation (6.2).

$$foldl(\oplus)v[x_0, x_1, x_2, \ldots, x_n] = (\cdots((v \oplus x_0) \oplus x_1)\cdots) \oplus x_n \tag{6.2}$$

6.4 Composition Operator

The composition operator is a Higher-Order Function that returns the composition of 2 functions as a single function. This is illustrated by the type signature and definition of the composition operator, `.` in Listing 72.

```haskell
(.) :: (b -> c) -> (a -> b) -> (a -> c)
{- The type signature above seems wrong, but it is right. The composition
   operator takes 2 arguments, and produces 1 result.
   1) A function, g (a -> b)
   2) A function, f (b -> c)
   Returns a function (a -> c)
 -}
f.g = \ x -> f(g x)
```

Listing 72: Type Signature and Definition of the Composition Operator

The definition in Listing 72 has the x argument shunted to the body of the definition using a lambda expression because, it makes explicit the idea that composition returns a function as its result.

`f.g` (read as "`f` composed with `g`") is the function that takes an argument `x`, applies the function `g` to this argument, and then applies the function `f` to the result of that. This means that these definitions are equivalent:

1. `(f.g) x`
2. `f (g x)`

Composition is associative, meaning `f.(g.h) == (f.g).h => True` for any functions `f`, `g`, and `h` of the appropriate types.

Composition can be used to simplify nested function applications, by reducing parentheses and avoiding the need to explicitly refer to the initial argument. However, there is a trade-off here. Using the composition operator will likely make your programs more difficult to read to someone who has never seen your code before. In the example below, the definitions of oddity are functionally equivalent, but one may be harder to read.

```haskell
odd1 n = not (even n)
odd2 = not.even
```

7 Declaring Types

New Types that more closely model your problem can be very helpful when writing a program. This also helps ensure that the type behaves according to the type checking rules you write for it.

7.1 Type Declarations

The simplest way to define a new Type is to use a new name for an already existing type. This is done with the `type` keyword in Haskell. This keyword allows us to create Type Aliases, making things easier to deal with.

Defn 25 (Type Alias)**.** A *type alias* is the process of creating another name for an already existing Type. The langauge system can just "replace" each occurrence of a type alias with its definition until it has fully expanded the Types until it has reached its base types. Type aliases can also be nested, allowing for one to define itself **IN TERMS OF ANOTHER**.

Remark 25.1 (Recursive Types)*.* **IT IS NOT POSSIBLE TO CONSTRUCT A RECURSIVE DATA TYPE USING A Type Alias!!** That can only be done with the `data` keyword.

Some examples of Type Aliases used to create new Types are shown in Listing 73.

```haskell
type String = [Char]

type Board = [Position]
type Position = (Int, Int)
```

Listing 73: New Types Declared with `type`

Now, wherever there was a list of positions, we can replace `[Position]` with `Board`. However, like mentioned in Remark 25.1, we cannot specify a recursive data Type with the `type` keyword.

7.1.1 Parameterized Type Declarations

In addition to creating new Types entirely, we can also parameterize them. To do this, you must create a *type function*. Just like regular Functions, there is no limit to the number of parameters that is accepted. Just like using normal Type Aliases, parameterized ones also simplify the writing of programs with parametric dependencies. All of these are shown in Listing 74.

```haskell
type Parser a = String -> [(a, String)]
type IO a = World -> (a, World)

-- Multiple type parameters
type Association key value = [(key, value)]

-- Using the multiple type parameters to simplify functions
find :: Eq k => Association k v -> v
find k t = head [v | (k', v) <- t, k == k']
```

Listing 74: New Types Declared with `type`

7.2 Data Declarations

To declare a completely new Type with its own typechecking rules is done with the `data` keyword. For example, the `Bool` Type is defined in the standard library as

```haskell
data Bool = False | True
```

The `|` symbol is read as "or". Each of the new values is a Type Constructor, `False` and `True` in this case. The first letter of these constructors must be a capital letter, and constructor names must be unique to modules. It is important to note that the constructor values, names, and the data type's name are all arbitrary. The only way to attach any meaning to one of these constructor types is by defining functions on them. For example, the below code is functionally identical to the definition of `Bool`.

```haskell
data A = B | C
```

Defn 26 (Type Constructor)**.** A *type constructor* is a function that constructs a new type in the compiler/runtime system.

Listing 75 is a simple example of declare a new Type and functions to define it.

```haskell
type Position = (Int, Int)
type Board = [Position]
data Move = Left
          | Right
          | Up
          | Down

move :: Move -> Position -> Position
move Left (x, y) = (x-1, y)
move Right (x, y) = (x+1, y)
-- These seem backwards because the origin is upper-left on the monitor
move Up (x, y) = (x, y-1)
move Down (x, y) = (x, y+1)

moves :: [Move] -> Position -> Position
moves [] p = p
moves (m:ms) p = moves ms (move m p)

flip :: Move -> Move
flip Left = Right
flip Right = Left
flip Up = Down
flip Down = Up
```

Listing 75: Simple `data` Declaration and Definition

7.2.1 Parameterized Constructor Functions

We can attach parameters to the constructor functions in a new Type. An extended example of such a declaration and subsequent definition is shown in Listing 76.

```haskell
data Shape = Circle Float
           | Rect Float Float

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * (r^2)
area (Rect x y) = x * y
```

Listing 76: Parameterized Constructors

In this case, `Circle 1.0` is an Expression that represents a piece of data. It cannot be further simplified to anything.

7.2.2 Parameterized Data Declarations

Just like in Section 7.1.1, we can parameterize our `data` declarations as well. These allow us to express some interesting things. One example is the application of potentially error-causing functions without causing an error, and instead checking if a value is present. This is demonstrated in Listing 77

```haskell
data Maybe a = Nothing
             | Just a

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Listing 77: Parameterized `data` Declaration and Definition

7.3 Recursive Types

Some Types lend themselves to being self-recursive. As we saw earlier, we cannot use the `type` keyword to construct these. Instead, we must use the `data` keyword to build new Types that **are** self-recursive.

2 examples are shown below.

1. Tree Recursive Type, Listing 78
2. List Recursive Type, Listing 79

```haskell
data Tree a = Leaf a
            | Node (Tree a) a (Tree a) -- This nodes value is stored "in the middle"

t :: Tree Int
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))

flatten :: Tree a -> [a]
flatten (Leaf n) = [n]
flatten (Node l v r) = flatten l ++ [v] ++ flatten r

occurs :: a -> Tree a -> Bool
occurs m (Leaf n) = m == n
occurs m (Node l v r) = occurs m l ||
                        m == v ||
                        occurs m r
```

Listing 78: Tree Recursive Type

```haskell
data List a = Nil
            | Cons a (List a)
```

Listing 79: List Recursive Type

7.4 Typeclass Declaration

In Haskell, a new Typeclass can be defined with the `class` keyword. For example, the standard library defines the `Eq` typeclass.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

Listing 80: Haskell's Standard Library Implementation of `Eq` Typeclass

This states that for any Type `a` to be part of the `Eq` Typeclass, it must support the equality and inequality operators/functions. Note that because of the way that inequality is defined, with its reliance on equality, only equality needs to be defined. To add a Type to a Typeclass, the `instance` keyword is used.

```
instance Eq Bool where
    False == False = True
    True == True = True
    _ == _ = False
```

Listing 81: Haskell's `Bool` Instantiation with the `Eq` Typeclass

Only Types declared with the `data` keyword can be added to Typeclasses. Ones declared with `type` inherit their typeclass information from their constiuent parts.

7.4.1 Extending Typeclasses

Sometimes it makes sense for a Typeclass to depend on another. This is the case for the `Ord` typeclass, shown in Listing 82.

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    min, max :: a -> a -> a

    min x y | x <= y = x
            | otherwise = y
    max x y | x <= y = y
            | otherwise x
```

Listing 82: Haskell's Standard Library Implementation of `Ord` Typeclass

This means that for a Type to be added to the `Ord` Typeclass, it must instantiate the 6 methods defined in `Ord` and the 2 defined in `Eq` .

7.4.2 Derived Instances

When creating new Types, they are usually derived from built-in Types, or Type Aliased ones, which themselves already belong to Typeclasses. Instead of having to manually add our new `data` Type to every typeclass, we can instead derive from them. An example involving the definition of the standard library's definition of `Bool` is shown in Listing 83.

In addition to the simpler examples of unparameterized Type Constructors and Types, we can also derive Typeclasses on parameterized types too. This requires that the types used in as the types support those typeclasses in the first place.

```haskell
data Bool = False
          | True
          deriving (Eq, Ord, Show, Read)
```

Listing 83: Derived Instance of Standard Library's `Bool` Type

```haskell
data Shape = Circle Float
           | Rect Float Float
           deriving (Eq, Ord, Show, Read)

(Rect 1.0 4.0) < (Rect 2.0 3.0) -- True
(Rect 1.0 4.0) < (Rect 1.0 3.0) -- True
```

Listing 84: Derived Instance of `Shape` Type

```haskell
data Maybe a = Nothing
             | Just a
             deriving (Eq, Ord, Show, Read)
```

Listing 85: Derived Instance of `Maybe` Type

7.4.3 Monadic Types

A Monad can be seen by the Typeclass declaration seen in Listing 86.

```haskell
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Listing 86: Declaration and Definition of the Monad Typeclass

That is, a monad is a parameterised type `m` that supports `return` and `>>=` functions of the specified types. The fact that `m` must be a parameterised type, rather than just a type, is inferred from its use in the types for the two functions.

8 Monads

So far, we have only written Haskell programs as Batch Programs. However, most people today like Interactive Programs.

Defn 27 (Batch Program)**.** A *batch program* is a program that takes some input from the user at the beginning of program execution and returns some result afterwards. During the computations, there is no further interaction between user and program.

Defn 28 (Interactive Program)**.** An *interactive program* is a program that may take some input from the user at the beginning of execution, and may return some result afterwards, **AND** will ask for user interaction during its life. This means the user can provide new data to program during its execution.

By the very nature of Interactive Programs, side effects are needed. These side effects and their functionality are handled by Monads in Haskell.

Defn 29 (Monad)**.** *Monads* are fairly unique to Haskell. They represent a way to support side-effect creating operations in the purely functional language.

One example of a monad is the `IO` typeclass. In most type signatures, monadic elements are indicated with an `m`.

We generalise the type for interactive programs to also return a result value, with the type of such values being a parameter of the `IO` (in this case) type:

```
type IO a = World -> (a, World)
```

Expressions of type `IO a` are called actions. For example, `IO Char` is the type of actions that return a character, while `IO ()` is the type of actions that return the empty tuple `()` as a dummy result value.

8.1 Basic `IO` Actions

There are 3 main actions in the `IO` monadic typeclass that are useful for our use.

(1) `getChar`

(2) `putChar`

(3) `return`

The `getChar` action, (Action (1)) reads a character from the keyboard, echoes it to the screen, and returns the character as its result value. If there are no characters waiting to be read from the keyboard, `getChar` waits until one is typed. The

```
getChar :: IO Char
getChar = ...
```

Listing 87: `getChar` Type Signature

actual definition for `getChar` is built-in to the particular Haskell system you are using, and cannot be defined within Haskell itself.

The `putChar c` action, (Action (2)) writes the character `c` to the screen, and returns no result value, represented by the empty tuple. The actual definition for `putChar` is built-in to the particular Haskell system you are using, and cannot be

```
putChar :: Char -> IO ()
putChar c = ...
```

Listing 88: `putChar` Type Signature

defined within Haskell itself.

The `return v` action, (Action (3)) simply returns the result value `v` without performing any interaction. `return` provides

```
return :: a -> IO a
return v = \world -> (v, world)
```

Listing 89: `return` Type Signature

a bridge from the setting of pure expressions without side effects to that of impure actions with side effects.

> **Once we are impure we are impure for ever, with no possibility for redemption!** As a result, we may suspect that impurity quickly permeates entire programs, but in practice this is usually not the case. For most Haskell programs, the vast majority of functions do not involve interaction, and the relatively small number of those that do are at the outermost level.

8.2 Sequencing

The natural way of combining two actions is to perform one after the other in sequence, with the modified world returned by the first action becoming the current world for the second, by means of a sequencing operator, read as "then". It is defined in Listing 90.

```
(>>=) :: m a -> (a -> m b) -> m b
f >>= g = \world -> case f world of
                         (v, world') -> g v world'
```

Listing 90: `>>=` Operator Definition

In words, we apply the action `f` to the current world, then apply the function `g` to the result value and the modified world as a second action, yielding the final result.

Using the `do`-notation allows us to use the "then" or "bind" operator without having to type it in that fashion. In addition, `do`-notation is syntactic sugar for a special case of a `where` block. An example of using `do`-notation is shown in Listing 91.

```
getChar :: IO Char
getChar = do x <- getCh -- Gets character from KBD without echoing to screen
             putChar x
             return x
```

Listing 91: `do`-Notation Example

8.3 Derived Primitives

We can perform actions on Monads quite similarly to the way we operate on other Types in Haskell. We can extend the basic ones and get more complex ones as we go. Here, we introduce 3 more monadic functions that use the `IO` Monad to perform many operations that are useful for interaction.

1. `getLine` The definition of the `getLine` function is shown in Listing 92.
2. `putStr` The definition of the `putStr` function is shown in Listing 93.
3. `putStrLn` The definition of the `putStrLn` function is shown in Listing 94.

```
getLine :: IO String -- Really says () -> IO String
getLine = do x <- getChar
             if x == '\n' then
                 return []
             else
                 do xs <- getLine
                    return (x:xs)
```

Listing 92: `getLine` Definition

9 Lazy Evaluation

So far, the basic method of computation in Haskell has been the application of Functions to arguments. Because these functions are pure, changing the order in which functions are applied does not affect the final result. This is not specific to simple examples, but an important general property of function application in Haskell. Formally, in Haskell any two different ways of evaluating the same expression will always produce the same final value, provided that they both terminate.

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Listing 93: `putStr` Definition

```haskell
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Listing 94: `putStrLn` Definition

An Expression that has the form of a Function applied to one or more arguments that can be "reduced" by performing the application is called a Reducible Expression

Defn 30 (Reducible Expression)**.** A *reducible expression* or *redex* has the form of a Function applied to one or more arguments that can be "reduced" by performing the appropriate function application. These reductions do not necessarily decrease the size of an Expression, but often do.

Consider the expression `mult (1+2, 2+3)` . This expression contains three redexes:

1. The sub-expression $1 + 2$, the `+` function applied with two arguments.
2. The sub-expression $2 + 3$, the `+` function applied with two arguments.
3. The entire expression itself, which has the form of the function `mult` applied to a pair of arguments.

Performing the corresponding reductions gives the expressions:

1. `mult (3, 2 + 3)`
2. `mult (1 + 2, 5)`
3. `(1 + 2) * (2 + 3)`

The order in which we evaluate these redexes determines what "Pass-by" type we use.

9.1 Pass-by-Value

Innermost evaluation, always chooses the innermost redex, meaning it contains no other redex. If there is more than one innermost redex, by convention we choose that which begins at the leftmost position in the expression.

Innermost evaluation can also be characterised in terms of how arguments are passed to Functions. In particular, using this strategy ensures that the argument of a function is always fully evaluated before the function itself is applied. That is, arguments are *Passed-By-Value*.

9.2 Pass-by-Name

Outermost evaluation, dual to innermost evaluation, is to always choose a redex that is outermost, meaning it is contained in no other redex. If there is more than one such redex, then we choose the one that begins at the leftmost position.

For example, the expression `mult (1+2) (2+3)` is contained in no other redex and is hence outermost within itself.

$$
\begin{aligned}
&= mult(1 + 2, 2 + 3) \\
&= (1 + 2) * (2 + 3) \\
&= (3) * (2 + 3) \\
&= 3 * (5) \\
&= 15
\end{aligned}
$$

In terms of how arguments are passed to functions, using outermost evaluation allows functions to be applied before their arguments are evaluated. For this reason, we say that arguments are *Passed-By-Name*.

Remark. Many built-in functions require their arguments to be evaluated before being applied, even when using outermost evaluation. For example, built-in arithmetic operators ($*$ and $+$) cannot be applied until their two arguments have been evaluated to numbers. Functions with this property are called Strict.

9.3 Reduction of Lambda Expressions

If we write a curried version of `mult`, `mult x = \y = x * y`. The two arguments are now substituted into the body of the function mult one at a time. This is the expected behavior of currying, rather than at the same time as in the previous section. This behaviour arises because `mult 3` is the leftmost innermost redex in the expression `mult 3 (2+3)`, as opposed to $2 + 3$ in the expression `mult (3, 2 + 3)`.

In Haskell, the selection of redexes within lambda expressions is prohibited. The rationale for not "reducing under lambdas" is that functions are viewed as black boxes that we are not permitted to look inside.

> Formally, the only operation that can be performed on a function is that of applying it to an argument. As such, reduction within the body of a function is only permitted once the function has been applied.

9.4 Termination

Throughout this section, we will use a recursive definition that yields a value similar to infinity.

```
1 infty :: Int
2 infty = 1 + infty
```

Listing 95: Definition of Infinity for Section 9.4

In practice, trying to evaluate `infty` will quickly exhaust the available memory and produce an error message. Consider the expression `fst (0, inf)`, where `fst` is the library function that selects the first component of a pair, defined by `fst (x, y) = x`.

Call-by-value evaluation with this expression also results in non-termination. In contrast, call-by-name evaluation terminates in just one step, by immediately applying the definition of `fst` avoiding the evaluation of the non-terminating expression.

This shows that call-by-name evaluation may produce a result when call-by-value evaluation fails to terminate. In general, we have the following important property: if there exists any evaluation sequence that terminates for a given expression, then call-by-name evaluation will also terminate for this expression, and produce the same final result.

In summary, call-by-name evaluation is preferable to call-by-value for the purpose of ensuring that evaluation terminates as often as possible.

9.5 Number of Reductions

Call-by-name evaluation may require more steps than call-by-value evaluation, in particular when an argument is used more than once in the body of a function. Generally, we have the following property: arguments are evaluated precisely once using call-by-value evaluation, but may be evaluated many times using call-by-name.

In this section, we will use the definition of `square` below.

```
1 square :: Int -> Int
2 square n = n * n
```

Listing 96: Definition of Squaring an Integer for Section 9.5

Below, we evaluate the expression `square (1+2)` according to the definition of `square` in Listing 96. Each line of derivation performs one reduction.

Using call-by-value evaluation, we have:

$$\begin{aligned} &= square(1+2) \\ &= square3 \\ &= 3*3 \\ &= 9 \end{aligned}$$

Using call-by-name evaluation, we have:

$$\begin{aligned} &= square(1+2) \\ &= (1+2)*(1+2) \\ &= 3*(1+2) \\ &= 3*3 \\ &= 9 \end{aligned}$$

Thus, call-by-name requires one extra reduction step, because `1+2` must be evaluated twice. Fortunately, this problem can be solved easily by using pointers to indicate sharing of expressions during evaluation. Rather than physically copying an argument if it is used many times in the body of a function, we simply keep one copy of the argument and make many pointers to it. In this manner, any reductions that are performed on the argument are automatically shared between each of the pointers to that argument. This works because the evaluation of an Expression depends only on the expression itself, not the state of the program.

The use of call-by-name evaluation in conjunction with sharing is called Lazy Evaluation, making Haskell a lazy programming language.

9.6 Infinite Structures

The use of Lazy Evaluation allows Haskell to program with infinite structures.

Defn 31 (Lazy Evaluation)**.** *Lazy evaluation* is the combination of call-by-name evaluation and the use of pointers to share references to common Expressions. Lazy evaluation has the property that it ensures that evaluation terminates as often as possible. Sharing ensures that lazy evaluation never requires more steps than call-by-value evaluation. In addition, using lazy evaluation expressions are only evaluated as much as required by the context in which they are used.

In this section, we will use the definition of `ones` below.

```
1 ones :: [Int]
2 ones = 1:ones
```

Listing 97: Definition of an Infinite List of ones for Section 9.6

No matter what evaluation strategy you use, name or value, this definition of `ones` will never terminate. However, when we attempt to use it (for example with `head ones`), depending on the evaluation method, we might get an answer.

Using call-by-value evaluation results in non-termination.

$$\begin{aligned} &= headones \\ &= head(1 : ones) \\ &= head(1 : (1 : ones)) \\ &= head(1 : (1 : (1 : ones))) \\ &\vdots \end{aligned}$$

Using call-by-name evaluation, sharing is not required in this example results in termination in two steps.

$$\begin{aligned} &= headones \\ &= head(1 : ones) \\ &= 1 \end{aligned}$$

This works because Lazy Evaluation proceeds in a lazy manner as its name suggests, only evaluating arguments as and when strictly necessary to produce results.

9.7 Modular Programming

Lazy Evaluation also allows us to separate control from data in our computations. For example, a list of three ones can be produced by selecting the first three elements (control) of the infinite list of ones (data):

```
> take 3 ones
[1, 1, 1]
```

The data is only evaluated as much as required by the control, and these two take it in turn to perform reductions. Without Lazy Evaluation, the control and data parts would need to be combined in the form of a single function. Allowing us to modularize by separating them into logically distinct parts is an important goal in programming. Being able to separate control from data is one of the most important benefits of lazy evaluation.

A good example of this is generating **ALL** the prime numbers, an inherently infinite task. Using the Eratosthenes Sieve method:

1. Write down the infinite sequence $2, 3, 4, 5, 6, \ldots$
2. Mark the first number, p, in the sequence as prime
3. Delete all multiples of p from the sequence
4. Return to the second step

```haskell
-- The expression primes will NOT be computed until we ask the system to.
-- As soon as we do, it will be stuck in an "infinite loop", finding all primes
primes :: [Int]
primes = sieve [2..] -- Infinite list of natural numbers, starting from 2

sieve :: [Int] -> [Int]
sieve (p : xs) = p : sieve [x | x <- xs, (x `mod` p) /= 0]
```

Listing 98: Eratosthenes Primes Algorithm

Lazy evaluation ensures that this program does indeed produce the infinite list of all prime numbers.

1. Start with the infinite list `[2..]`.
2. Apply the `sieve` function that retains the first number `p` as being prime.
3. Filter all multiples of `p` from this list.
4. `sieve` calls itself recursively with the list it just obtained (with all multiples of `p` filtered out).

If we simply ask for the value of the `primes` expression, we get an infinite list of primes back. By freeing the generation of prime numbers from the constraint of finiteness, we have obtained a modular program on which different control parts can be used in different situations.

9.7.1 Caveats with Modular Programming

Even though we can modularize our code between control and data, we need to be careful when the data is an infinite structure (list or recursion). In Listing 99, this is described more.

9.8 Strict Application

Haskell uses Lazy Evaluation by default, but also provides a special Strict version of Function application, written as `$!`, which can be useful.

Defn 32 (Strict)**.** *Strict* function application, in Haskell, means that the specified argument provided to the Function is evaluated with Pass-by-Value, rather than Pass-by-Name.

Formally, an expression of the form `f $! x` is only a Reducible Expression once evaluation of the argument `x`, which itself uses Lazy Evaluation, has reached the point where it is known that the result is not an undefined value. At that point the expression can be reduced to the normal application `f x` .

- If the argument has a basic type, such as `Int` or `Bool`, then top-level evaluation is simply complete evaluation.
- For a Tuple type such as `(Int, Bool)`, evaluation is performed until a pair of expressions is obtained, but no further.

```
filter (<=5) [1..]
{- This WILL NOT TERMINATE!
   This is because filter will go through every element in the list and apply
    the predicate (<=5).
   Thus, it will return the first 5 elements [1, 2, 3, 4, 5] and will then get
    stuck running until the end of the infinite list.
 -}

takeWhile (<=5) [1..]
{- This WILL TERMINATE!
   This is because takeWhile will go through every element in the list and apply
    the predicate (<=5).
   However, it will STOP when the predicate ceases to return true.
   Thus, it will return [1, 2, 3, 4, 5]
 -}
```

Listing 99: Infinite List Gotchas

- For a List type, evaluation is performed until the empty list or the cons of two expressions is obtained.

When used with Curried Function Types with multiple arguments, Strict application can be used to force top-level evaluation of any combination of arguments. A curried function application of `f x y` has 3 possible different behaviors with `$!`:

1. `(f $! x) y` : Force top-level evaluation of `x`.
2. `(f x) $! y` : Force top-level evaluation of `y`.
3. `(f $! x) $! y` : Force top-level evaluation of `x` and `y`.

Strict application is mainly used to improve the space performance of programs. Take the code in Listing 100 as an example.

The Strict evaluation requires more steps, due to the additional overhead of using strict application, but now performs each addition as soon as it is introduced, rather than constructing a large summation.

Strict application is not a panacea that automatically improves the space behaviour of programs. Even for simple examples, the use of strict application requires careful consideration of the behaviour of lazy evaluation.

10 Lambda Calculus

Defn 33 (Lambda Calculus)**.** *Lambda Calculus*

```haskell
sumwith :: Int -> [Int] -> Int
sumwith v [] = v
sumwith v (x:xs) = sumwith (v+x) xs

sumwith 0 [1, 2, 3]
{- = sumwith (0 + 1) [2, 3]
   = sumwith ((0 + 1) + 2) [3]
   = sumwith (((0 + 1) + 2) + 3) []
   = ((0 + 1) + 2) + 3
   = (1 + 2) + 3
   = 3 + 3
   = 6
 -}

strictSumwith :: Int -> [Int] -> Int
strictSumwith v [] = v
strictSumwith v (x:xs) = (sumwith $! (v+x)) xs

strictSumwith 0 [1, 2, 3]
{- = sumwith £! (0 + 1) [2, 3]
   = sumwith £! 1 [2, 3]
   = sumwith 1 [2, 3]
   = sumwith £! (1 + 2) [3]
   = sumwith £! 3 [3]
   = sumwith 3 [3]
   = sumwith £! (3 + 3) []
   = sumwith £! 6 []
   = sumwith 6 []
   = 6
 -}
```

Listing 100: Lazy vs. Strict Function Application

A Complex Numbers

Complex numbers are numbers that have both a real part and an imaginary part.

$$z = a \pm bi \tag{A.1}$$

where

$$i = \sqrt{-1} \tag{A.2}$$

Remark (i vs. j for Imaginary Numbers). Complex numbers are generally denoted with either i or j. Since this is an appendix section, I will denote complex numbers with i, to make it more general. However, electrical engineering regularly makes use of j as the imaginary value. This is because alternating current i is already taken, so j is used as the imaginary value instad.

$$Ae^{-ix} = A\left[\cos\left(x\right) + i\sin\left(x\right)\right] \tag{A.3}$$

A.1 Complex Conjugates

If we have a complex number as shown below,

$$z = a \pm bi$$

then, the conjugate is denoted and calculated as shown below.

$$\overline{z} = a \mp bi \tag{A.4}$$

Defn A.1.1 (Complex Conjugate)**.** The conjugate of a complex number is called its *complex conjugate*. The complex conjugate of a complex number is the number with an equal real part and an imaginary part equal in magnitude but opposite in sign.

The complex conjugate can also be denoted with an asterisk ($*$). This is generally done for complex functions, rather than single variables.

$$z^* = \overline{z} \tag{A.5}$$

A.1.1 Complex Conjugates of Exponentials

$$\overline{e^z} = e^{\overline{z}} \tag{A.6}$$

$$\overline{\log(z)} = \log(\overline{z}) \tag{A.7}$$

A.1.2 Complex Conjugates of Sinusoids

Since sinusoids can be represented by complex exponentials, as shown in Appendix B.2, we could calculate their complex conjugate.

$$\begin{aligned} \overline{\cos(x)} &= \cos(x) \\ &= \frac{1}{2}\left(e^{ix} + e^{-ix}\right) \end{aligned} \tag{A.8}$$

$$\begin{aligned} \overline{\sin(x)} &= \sin(x) \\ &= \frac{1}{2i}\left(e^{ix} - e^{-ix}\right) \end{aligned} \tag{A.9}$$

B Trigonometry

B.1 Trigonometric Formulas

$$\sin(\alpha) + \sin(\beta) = 2\sin\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right) \tag{B.1}$$

$$\cos(\theta)\sin(\theta) = \frac{1}{2}\sin(2\theta) \tag{B.2}$$

B.2 Euler Equivalents of Trigonometric Functions

$$e^{\pm j\alpha} = \cos(\alpha) \pm j\sin(\alpha) \tag{B.3}$$

$$\cos(x) = \frac{e^{jx} + e^{-jx}}{2} \tag{B.4}$$

$$\sin(x) = \frac{e^{jx} - e^{-jx}}{2j} \tag{B.5}$$

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2} \tag{B.6}$$

$$\cosh(x) = \frac{e^{x} + e^{-x}}{2} \tag{B.7}$$

B.3 Angle Sum and Difference Identities

$$\sin(\alpha \pm \beta) = \sin(\alpha)\cos(\beta) \pm \cos(\alpha)\sin(\beta) \tag{B.8}$$

$$\cos(\alpha \pm \beta) = \cos(\alpha)\cos(\beta) \mp \sin(\alpha)\sin(\beta) \tag{B.9}$$

B.4 Double-Angle Formulae

$$\sin(2\alpha) = 2\sin(\alpha)\cos(\alpha) \tag{B.10}$$

$$\cos(2\alpha) = \cos^2(\alpha) - \sin^2(\alpha) \tag{B.11}$$

B.5 Half-Angle Formulae

$$\sin\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1-\cos(\alpha)}{2}} \tag{B.12}$$

$$\cos\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1+\cos(\alpha)}{2}} \tag{B.13}$$

B.6 Exponent Reduction Formulae

$$\sin^2(\alpha) = \frac{1-\cos(2\alpha)}{2} \tag{B.14}$$

$$\cos^2(\alpha) = \frac{1+\cos(2\alpha)}{2} \tag{B.15}$$

B.7 Product-to-Sum Identities

$$2\cos(\alpha)\cos(\beta) = \cos(\alpha-\beta) + \cos(\alpha+\beta) \tag{B.16}$$

$$2\sin(\alpha)\sin(\beta) = \cos(\alpha-\beta) - \cos(\alpha+\beta) \tag{B.17}$$

$$2\sin(\alpha)\cos(\beta) = \sin(\alpha+\beta) + \sin(\alpha-\beta) \tag{B.18}$$

$$2\cos(\alpha)\sin(\beta) = \sin(\alpha+\beta) - \sin(\alpha-\beta) \tag{B.19}$$

B.8 Sum-to-Product Identities

$$\sin(\alpha) \pm \sin(\beta) = 2\sin\left(\frac{\alpha \pm \beta}{2}\right)\cos\left(\frac{\alpha \mp \beta}{2}\right) \tag{B.20}$$

$$\cos(\alpha) + \cos(\beta) = 2\cos\left(\frac{\alpha + \beta}{2}\right)\cos\left(\frac{\alpha - \beta}{2}\right) \tag{B.21}$$

$$\cos(\alpha) - \cos(\beta) = -2\sin\left(\frac{\alpha + \beta}{2}\right)\sin\left(\frac{\alpha - \beta}{2}\right) \tag{B.22}$$

B.9 Pythagorean Theorem for Trig

$$\cos^2(\alpha) + \sin^2(\alpha) = 1^2 \tag{B.23}$$

B.10 Rectangular to Polar

$$a + jb = \sqrt{a^2 + b^2}e^{j\theta} = re^{j\theta} \tag{B.24}$$

$$\theta = \begin{cases} \arctan\left(\frac{b}{a}\right) & a > 0 \\ \pi - \arctan\left(\frac{b}{a}\right) & a < 0 \end{cases} \tag{B.25}$$

B.11 Polar to Rectangular

$$re^{j\theta} = r\cos(\theta) + jr\sin(\theta) \tag{B.26}$$

C Calculus

C.1 Fundamental Theorems of Calculus

Defn C.1.1 (First Fundamental Theorem of Calculus)**.** The *first fundamental theorem of calculus* states that, if f is continuous on the closed interval $[a, b]$ and F is the indefinite integral of f on $[a, b]$, then

$$\int_a^b f(x)\,dx = F(b) - F(a) \tag{C.1}$$

Defn C.1.2 (Second Fundamental Theorem of Calculus)**.** The *second fundamental theorem of calculus* holds for f a continuous function on an open interval I and a any point in I, and states that if F is defined by

$$F(x) = \int_a^x f(t)\,dt,$$

then

$$\begin{aligned} \frac{d}{dx}\int_a^x f(t)\,dt &= f(x) \\ F'(x) &= f(x) \end{aligned} \tag{C.2}$$

Defn C.1.3 (argmax)**.** The arguments to the *argmax* function are to be maximized by using their derivatives. You must take the derivative of the function, find critical points, then determine if that critical point is a global maxima. This is denoted as

$$\underset{x}{\operatorname{argmax}}$$

C.2 Rules of Calculus

C.2.1 Chain Rule

Defn C.2.1 (Chain Rule)**.** The *chain rule* is a way to differentiate a function that has 2 functions multiplied together.
If

$$f(x) = g(x) \cdot h(x)$$

then,

$$\begin{aligned} f'(x) &= g'(x) \cdot h(x) + g(x) \cdot h'(x) \\ \frac{df(x)}{dx} &= \frac{dg(x)}{dx} \cdot g(x) + g(x) \cdot \frac{dh(x)}{dx} \end{aligned} \tag{C.3}$$

D Laplace Transform

Defn D.0.1 (Laplace Transform)**.** The *Laplace transformation* operation is denoted as $\mathcal{L}\{x(t)\}$ and is defined as

$$X(s) = \int_{-\infty}^{\infty} x(t)e^{-st}dt \tag{D.1}$$

References

[Hut07] Graham Hutton. *Programming in Haskell*. English. 1st ed. Cambridge University Press, 2007. 183 pp. ISBN: 9781316626221.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. English. Apr. 2011. 290 pp. ISBN: 9781593272838. URL: `http://learnyouahaskell.com/`.

[OGS08] Bryan O'Sullivan, John Goerzen, and Donal Bruce Stewart. *Real World Haskell*. Code You Can Believe In. English. O'Reilly Media, Inc., Nov. 2008. 712 pp. ISBN: 9780596554309.