EDAN40/EDAN95: Functional Programming — Reference Sheet
Lund University

Karl Hallsby

Last Edited: March 25, 2020

Contents

List of Theorems iii

1 Introduction 1
- 1.1 Rewrite Semantics 1
- 1.2 Paradigm Differences 2
  - 1.2.1 Side Effects 2
  - 1.2.2 Syntactic Differences 2
  - 1.2.3 Tendency Towards Recursion 2
  - 1.2.4 Higher-Order Functions 3
  - 1.2.5 Infinite Data Structures 3
- 1.3 Language Basics 5
  - 1.3.1 Mathematical Operations 5
    - 1.3.1.1 Precedences 5
    - 1.3.1.2 Associativity 5
  - 1.3.2 List Operations 6
    - 1.3.2.1 `head` 6
    - 1.3.2.2 `tail` 6
    - 1.3.2.3 `last` 6
    - 1.3.2.4 `init` 6
    - 1.3.2.5 Selection, `!!` 6
    - 1.3.2.6 `take` 7
    - 1.3.2.7 `drop` 7
    - 1.3.2.8 Appending Lists to Lists, `++` 7
    - 1.3.2.9 Constructing Lists, `:` 7
    - 1.3.2.10 `length` 7
    - 1.3.2.11 `sum` 7
    - 1.3.2.12 `product` 7
    - 1.3.2.13 `reverse` 7
  - 1.3.3 Function Application 8
  - 1.3.4 Haskell Files/Scripts 9
    - 1.3.4.1 Naming Conventions 9
  - 1.3.5 Language Keywords 9
  - 1.3.6 The Layout Rule 10
  - 1.3.7 Comments 10

2 Constructing Functions 10
- 2.1 Conditional Expressions 10
- 2.2 Guarded Equations 11
- 2.3 Pattern Matching 11
  - 2.3.1 Tuple Patterns 12
  - 2.3.2 List Patterns 12
  - 2.3.3 Integer Patterns 12
- 2.4 Lambda Expressions 13
- 2.5 Sections 13

3 List Comprehensions **14**
3.1 Generators . . . 14
3.1.1 Multiple Generators . . . 14
3.1.2 Dependent Generators . . . 15
3.2 Guards . . . 15
3.3 The `zip` Function . . . 15
3.4 String Comprehensions . . . 16

4 Recursion **17**
4.1 List Recursion . . . 17
4.2 Multiple Arguments . . . 17
4.3 Multiple Recursion . . . 19
4.4 Mutual Recursion . . . 19
4.5 Help With Recursion . . . 19
4.5.1 Define Function Type . . . 19
4.5.2 Enumerate Cases . . . 20
4.5.3 Define Simple Cases . . . 20
4.5.4 Define Other Cases . . . 20
4.5.5 Generalize/Simplify . . . 20

5 Monads **20**

6 Lambda Calculus **20**

A Complex Numbers **21**
A.1 Complex Conjugates . . . 21
A.1.1 Complex Conjugates of Exponentials . . . 21
A.1.2 Complex Conjugates of Sinusoids . . . 21

B Trigonometry **22**
B.1 Trigonometric Formulas . . . 22
B.2 Euler Equivalents of Trigonometric Functions . . . 22
B.3 Angle Sum and Difference Identities . . . 22
B.4 Double-Angle Formulae . . . 22
B.5 Half-Angle Formulae . . . 22
B.6 Exponent Reduction Formulae . . . 22
B.7 Product-to-Sum Identities . . . 22
B.8 Sum-to-Product Identities . . . 23
B.9 Pythagorean Theorem for Trig . . . 23
B.10 Rectangular to Polar . . . 23
B.11 Polar to Rectangular . . . 23

C Calculus **24**
C.1 Fundamental Theorems of Calculus . . . 24
C.2 Rules of Calculus . . . 24
C.2.1 Chain Rule . . . 24

D Laplace Transform **25**

List of Theorems

1 Defn (Imperative Programming Language) . . . 1
2 Defn (Functional Programming Language) . . . 1
3 Defn (Rewrite Semantics) . . . 1
4 Defn (Expression) . . . 1
5 Defn (Operand) . . . 1
6 Defn (Conditional Expression) . . . 10
7 Defn (Guarded Equation) . . . 11
8 Defn (Guard) . . . 11
9 Defn (Lambda Expression) . . . 13
10 Defn (Section) . . . 13
11 Defn (Generator) . . . 14
12 Defn (Recursion) . . . 17
13 Defn (Multiple Recursion) . . . 19
14 Defn (Mutual Recursion) . . . 19
15 Defn (Monad) . . . 20
16 Defn (Lambda Calculus) . . . 20
A.1.1Defn (Complex Conjugate) . . . 21
C.1.1Defn (First Fundamental Theorem of Calculus) . . . 24
C.1.2Defn (Second Fundamental Theorem of Calculus) . . . 24
C.1.3Defn (argmax) . . . 24
C.2.1Defn (Chain Rule) . . . 24
D.0.1Defn (Laplace Transform) . . . 25

List of Listings

1 Rewrite Semantics of a Factorial Function . . . 2
2 C-Like Code with Side Effects . . . 2
3 Basic List Summation . . . 3
4 List Comprehension Functions, No Higher-Order Functions Used . . . 3
5 List Comprehension Functions, Higher-Order Functions Used . . . 4
6 Infinite Data Structure, All Primes by Eratosthenes Sieve . . . 4
7 Integer Mathematical Operations . . . 5
8 Haskell `head` Function . . . 6
9 Haskell `tail` Function . . . 6
10 Haskell `last` Function . . . 6
11 Haskell `init` Function . . . 6
12 Haskell `!!` Function . . . 6
13 Haskell `take` Function . . . 7
14 Haskell `drop` Function . . . 7
15 Haskell `++` Function . . . 7
16 Haskell `:` Function . . . 7
17 Haskell `length` Function . . . 7
18 Haskell `sum` Function . . . 8
19 Haskell `product` Function . . . 8
20 Haskell `reverse` Function . . . 8
21 Define Function From Others . . . 10
22 Example Conditional Expression . . . 10
23 A Guarded Equation in Haskell . . . 11
24 A Guarded Equation with Early Matching . . . 11
25 Basic Pattern Matching in Haskell . . . 12
26 Pattern Matching with Multiple Parameters and Wildcards . . . 12
27 Tuple Pattern Matching . . . 12
28 List Pattern Matching . . . 13
29 Lambda Expressions in Haskell . . . 13
30 Haskell List Comprehensions . . . 14
31 Wildcard Generator . . . 14
32 Multiple Generators . . . 14
33 Dependent Generators . . . 15
34 Guarded List Comprehension for Prime Generation . . . 15
35 Using the `zip` Function . . . 16
36 Polymorphic List Comprehensions Used on Strings . . . 16
37 List Comprehensions Used on `String` s . . . 16
38 Factorial, Recursively Defined . . . 17
39 Product of a List . . . 17
40 Length of a List . . . 18
41 Multi-Argument Recursion of Haskell's `zip` Function . . . 18
42 Multi-Argument Recursion of Haskell's `drop` Function . . . 18
43 Multiple Recursion in Fibonacci's Sequence . . . 19
44 Mutually Recursive `odd` and `even` Functions . . . 19

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

```
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.

```
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

```
> 2+3
5
> 2-3
-1
> 2*3
6
>7 `div` 2
3
> 2^3
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.

$$\{1,4,9,16,25\} = \left\{x^2 | x \in \{1\ldots5\{\right\} \tag{3.1}$$

Equation (3.1) would be said to contain all numbers x^2 where each x is an element of the set $\{1\ldots5\}$. 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.

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 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, and in fact, you have already seen them in Section 3.1. 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.

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]
-- Because of lazy evaluation, the False result is returned as soon as ANY
-- factor other than one or the number itself is produced.

-- 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.

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

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.

```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

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.

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.

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.

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
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

```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

```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.

```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

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.

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

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

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

```
[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 Monads

Defn 15 (Monad)**.** *Monads* are fairly unique to Haskell.

6 Lambda Calculus

Defn 16 (Lambda Calculus)**.** *Lambda Calculus*

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.

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

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

$$\operatorname*{argmax}_x$$

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.