# EDAN40/EDAN95: Functional Programming — Reference Sheet
# Lund University

Karl Hallsby

Last Edited: March 23, 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

# 2 Monads

**Defn 6** (Monad)**.** *Monads* are fairly unique to Haskell.

# 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.