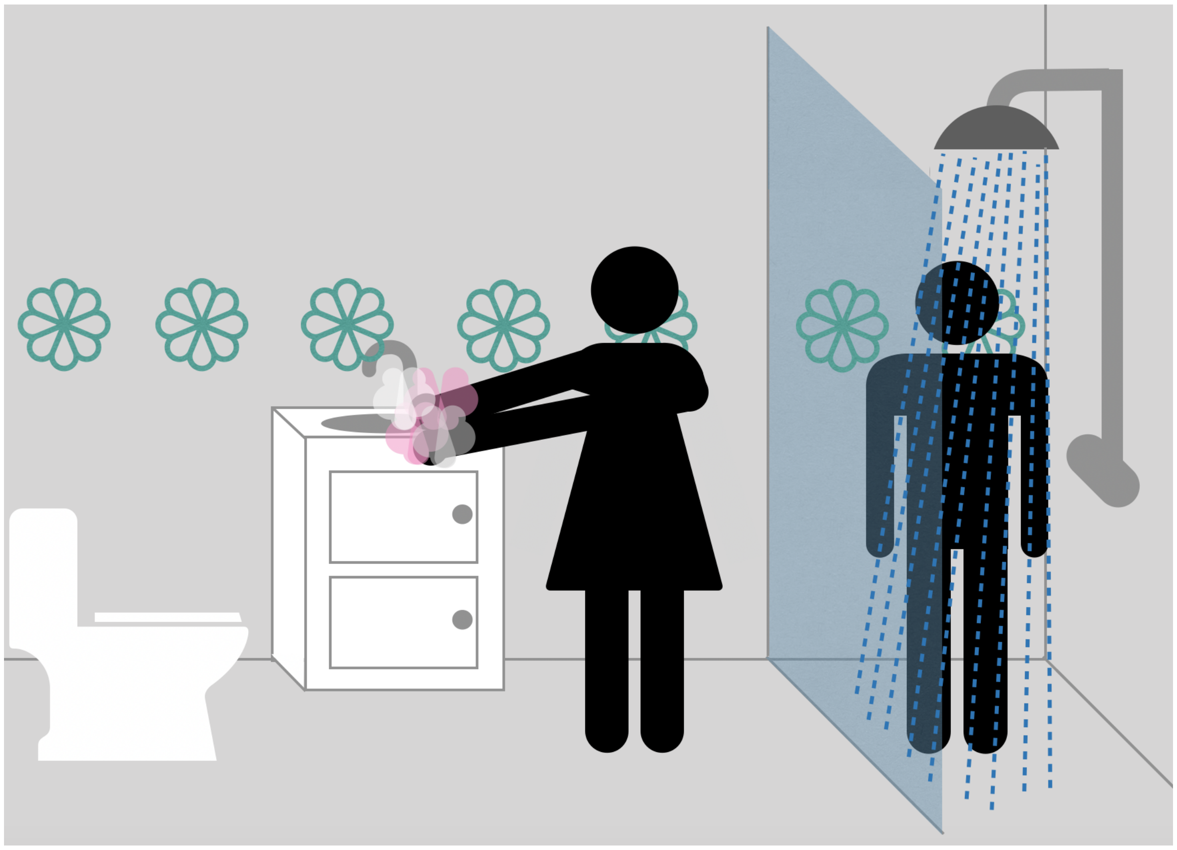
Domain-specific languages of Control Theory

A supplementary learning material for ERE103



Jakob Fihlman Simon Hägglund Christian Josefson Filip Nylander
Elin Ohlman Tommy Räjert

Spring - 2020

Table of Contents

1 Introduction

This is a supplementary learning material for the course “Control theory” (ERE103) at Chalmers University of Technology. The purpose of this material is to give computer science students a more familiar introduction to the subject of control theory. While aimed towards students taking the course ERE103, this learning material is suitable for everyone with an interest in learning more about control theory and is familiar with Haskell.

The method we use in this material is to create domain-specific languages (DSLs) for the course. A DSL is a programming language specialised for a domain, be it math, physics, astronomy, etc. In our case, we will introduce a new DSL for each section, each specialised for the content of said section. For students who already attended the course “DSLsofMath” this approach should be nothing new. For students who have not, additional details are provided throughout the first chapter, “Prerequisites.”

All DSLs in this material are written in Haskell, so the syntax should be familiar from previous courses. If you are not familiar with Haskell or just want to brush up your skills, we recommend “Learn you a Haskell for Great Good!” by Miran Lipovača (http://learnyouahaskell.com/).

Before you start, we just want to mention that this learning material hasn’t been tested yet, and as such we can’t guarantee that it will be helpful. We hope that reading it will be beneficial, however. We hope you find this reading useful. Best of luck!

1.1 Basic control theory

Control theory studies continuous systems and the control of these. A easy example of a control system, with human input, is when you take a shower. Your body sends signals to the brain when its to hot or cold and you adjust the temperature with your hand. In this case the input is the signal your body sends to your brain and the output is the water temperature. This is a controls system with human input. In control theory we want to automate systems. A example of a common automated control system is the temperature system in your house. A thermostat measures the outside and inside temperature and control the heat elements in you house. This is a control system with two inputs.

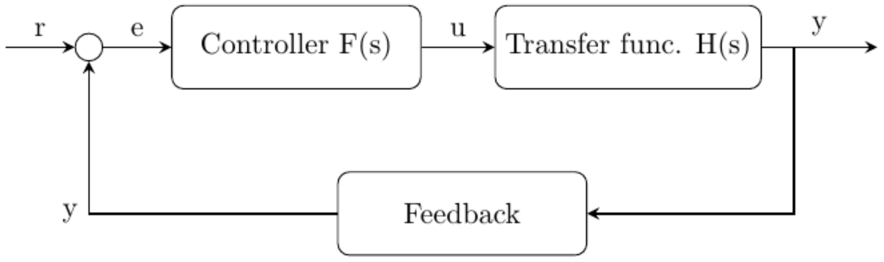


Figure 1: Feedback-loop

Control systems are created with the combination of measuring a magnitude and a controller. The magnitude measurements and controller create the system input u(t). A system can also utilize the output from the control. This is possible with a feedback loop and is a very common practice in control theory. The quest for all control systems is to create a stable system without delay or overshoot. In the course ERE103 this is fulfilled by combining tree different parameters (P, I, D) to create different controllers f(t). P is just a constant and can decrease the delay of the system. P creates the most basic controller f(t) = P. The I parameter is the integrating part of the controller and is used to remove the stationary fault. The D parameter is used to avoid overshoot. These parameters can be combined to fulfill your specifications. The most common combination of these parameters in this course are PD, PI and PID. A controller is just a function and can therefore easily be modeled in Haskell.

Control systems are often modelled in block diagrams as you can see above. When we analyze control systems we often transform the functions from the time domain to the frequency domain with the laplace transform. The laplace transform is very useful and will be explained more in-depth later.

In control systems we want our systems to be stable. The results of unstable systems can be horrific so we want to avoid this at all cost. We can use a few methods to ensure this like Ruth-Hurwitz and the Nyqvist Theorem.

2 Prerequisites

This section is dedicated to refresh your understanding of key concepts that are not technically part of control theory, but are nonetheless necessary in order to understand it. In addition, this section serves to introduce the DSL way of thinking about mathematical concepts so that you may get comfortable with the idea before moving on to the more difficult subjects control theory contains.

These subjects are only touched upon on a very basic level, thus if you are already feeling confident about them, this section can safely be skipped.

2.1 Complex numbers

In short, the complex numbers are an extension of the real numbers in the same way that the real numbers are an extension of the rational numbers. With the reals, we saw the addition of a number of constants (e, π, etc.). This time around, we add imaginary numbers.

First, some technicalities. The complex numbers are built on the real numbers, but the real numbers are unfortunately not very easily implementable in computers. In reality we have to approximate. In order to avoid this problem, we're going to introduce a type synonym:

```haskell
type R = Double
```

Now, whenever we want something to be a real number, we can use the type `R`. Type synonyms are just that—synonyms—so whenever you see `R` throughout the text you could exchange it with `Double` and get the exact same result. We hope you see the reason in using `R`, however.

2.1.1 The Complex Number Concept

So what are imaginary numbers? In short, imaginary numbers are a way to solve some equations that are otherwise impossible to solve. For example, if we wanted to find a square root to -1 we'd need to use imaginary numbers.

Let's start with the definition:

$$i = \sqrt{-1},$$

and as a consequence,

$$i^2 = -1.$$

What we've got so far is just one number. So what? Well, let's look at what we can do with it.

Imaginary numbers have a tendency to show up alongside real numbers. As such, mathematicians have standardized complex numbers as being read "a + bi." We'll get back to that soon, but first we want to introduce our own representation.

```haskell
data Complex = Complex (R, R)
    deriving (Show)
```

Example 1. Write the complex number $2 + 3i$ in `Complex`.

Solution. `z = Complex (2,3)`

This is the first step to creating a domain-specific language: establishing how our 'words' (or in this case numbers) are written. We create a data type called `Complex`, which holds a pair of values.

Now you might ask yourself: why did we choose to put our reals in parenthesis? The answer lies in the complex plane and how complex numbers can be seen as coordinates in a diagram. For example, see the following figure:

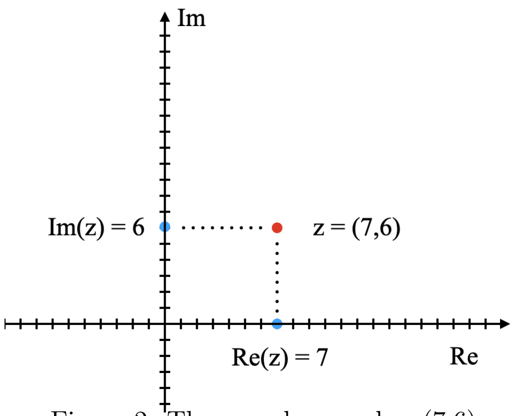


Figure 2: The complex number (7,6)

In the picture you see a complex plane with the complex number $(7, 6)$ marked. Note the similarity to coordinate planes.

2.1.2 Basic Operators

For our first operators, let's just create a pair of functions that extract the real and imaginary parts of the complex number. As can be seen in the picture above, the real numbers exist along the x-axis, and the imaginary ones exist along the y-axis. As such:

```haskell
whatsReal :: Complex -> R
whatsReal (Complex (x,y)) = x
```

Exercise 1. Implement the other function, `whatsImaginary :: Complex -> Real`.

Next, let's define the most fundamental of all math operators: addition. Please look at Figure 3.

In the picture we see three complex numbers; v, w and z. We can also tell that v and w can be interpreted as paths leading from the origin(0,0) to their respective values.

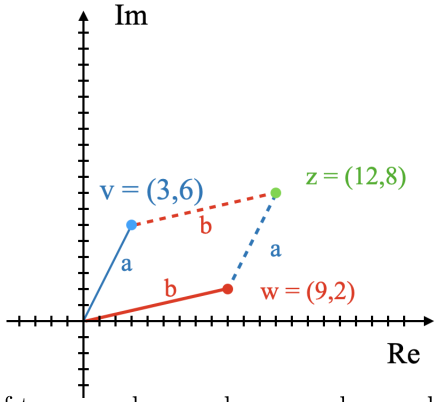


Figure 3: Addition of two complex numbers v and w, and the resulting number z.

Under this interpretation, adding complex numbers together can be interpreted as putting two paths back to back; either adding v to w(lower path), or w to v(upper path), in order to reach a third value(z).

We can also tell that the resulting answer(z) has an imaginary component equal to the sum of the imaginary components in v and w(6+2), and a real component equal to the sum of the real components in v and w(3+9). This is something we can implement directly into our DSL; we'll simply add reals to reals, and imaginaries to imaginaries.

To wit:

```haskell
add :: Complex -> Complex -> Complex
add (Complex (r1,i1)) (Complex (r2,i2)) = (Complex (r1+r2,i1+i2))
```

Let's also add subtraction. This works according to the same principle as addition; real affects real and imaginary affects imaginary, only this time with subtraction rather than addition.

```haskell
sub :: Complex -> Complex -> Complex
sub (Complex (r1,i1)) (Complex (r2,i2)) = Complex (r1-r2,i1-i2)
```

2.1.3 Complex Geometry

Now, let's move on to the main reason why we chose to mimic coordinates when defining our datatype. No, it wasn't to simplify addition and subtraction. That was just a bonus.

The real reason is that it allows us to use complex numbers to express geometry. To do this, we'll need two core functions. First off, the absolute value.

Note that, since complex numbers exist on a plane rather than a line, we can't just turn them positive and call it a day. Instead, we'll have to call upon our good old friend Pythagoras:

```haskell
absolute :: Complex -> R
absolute (Complex (real,imaginary)) = sqrt (real^2 + imaginary^2)
```

Since squaring always result in positive values, there's no need to worry about whether the values started out positive or negative.
Let's move on to arguments, a.k.a. angles.

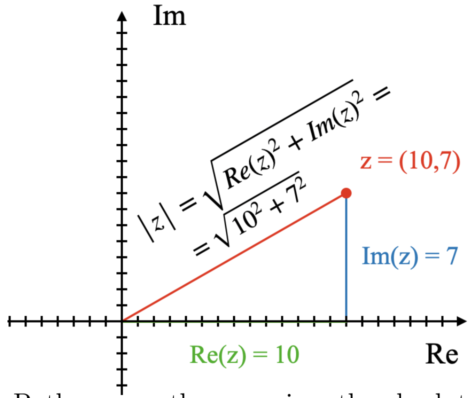


Figure 4: The Pythagorean theorem gives the absolute value of (10,7)

The Pythagorean theorem is not the only geometric rule that can be combined with complex diagrams/coordinates. In fact, all of them can. Let's do some tangents. Except when working with complex numbers, the resulting value is called an argument rather than an angle.

Unlike regular angles, however, the standard way to express a complex argument, also known as the "Principal Argument", does not include a complete 360-degree circle from positive real to positive real. Instead, the principal argument uses two scales from the positive reals to the negative reals. One covers positive imaginary parts and gives arguments ranging from 0 to π. The other covers negative imaginaries and gives arguments ranging from 0 to $-\pi$. See figure 5 below for clarity.

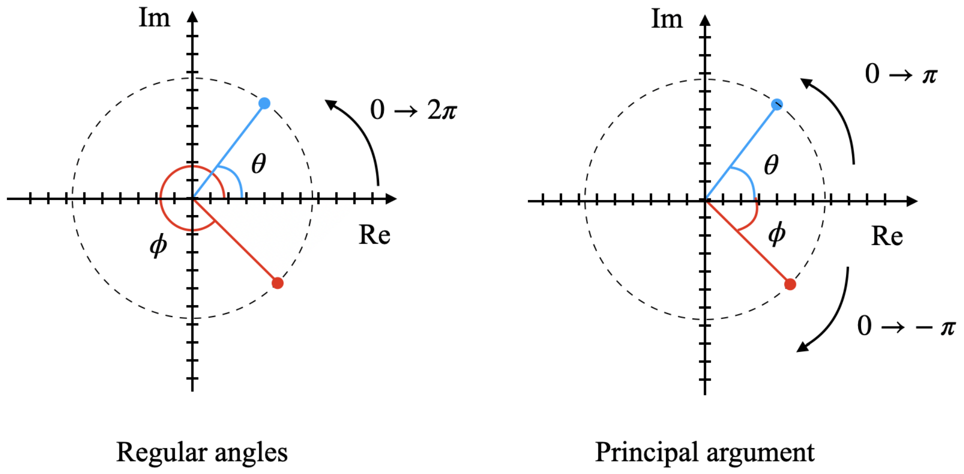


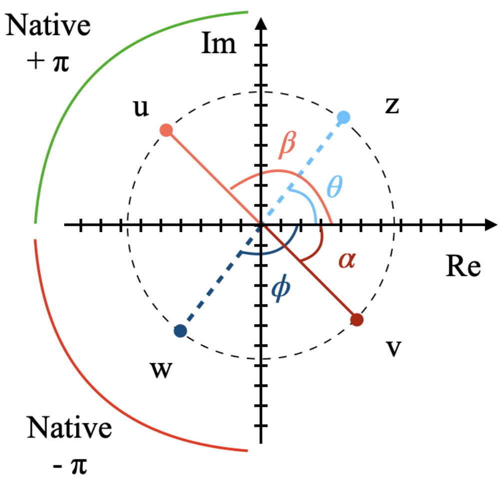
Figure 5: Angles vs arguments

The good news is that the arcus tangent (arctan) operator supports this split natively. The bad news is that the arctan operator input only describes the slope of the line; the coefficient.

As illustrated just below, what we'll have to do is program the operator to determine which quadrant the complex number is in, and "spin" the resulting argument accordingly.

```haskell
argument :: Complex -> Double
argument (Complex (real,imaginary))
  | real < 0 && imaginary > 0  = (atan (imaginary/real)) + pi
```

$$tan(\theta) = \frac{Im(z)}{Re(z)} = tan(\phi) = \frac{Im(w)}{Re(z)} = k \qquad tan(\alpha) = \frac{Im(u)}{Re(u)} = tan(\beta) = \frac{Im(v)}{Re(v)} = l$$



Where k is the slope of the "blue" dashed line and l is the slope of the "red" solid line

$$arctan(k) = \theta \qquad arctan(l) = \alpha$$

$$arg(z) = arctan(k) = \theta$$

$$arg(w) = arctan(k) - \pi = \phi$$

$$arg(v) = arctan(l) = \alpha$$

$$arg(u) = arctan(l) + \pi = \beta$$

Figure 6: Translating arcus tangents to principal arguments

```
  | real < 0                         = (atan (imaginary/real)) - pi
  | otherwise                        = atan (imaginary/real)
```

Please note that our function is currently undefined whenever the real component is zero, due to division by zero. We could solve this with a further three lines describing specific cases for +90°, -90°, and true zero. In this case we chose not to in order to keep our code as clean as possible. Alternatively, we could've just used Haskell's builtin version of the operator above, known as `atan2`. If we were lazy, that is. Which we are not.

The Absolute Value and the Principal Argument, with their powers combined, form an alternate way to express the "position" of a Complex number; by expressing the distance and direction from the origin. The later sections on Laplace transforms and Nyquist diagrams will further explore the possibilities of this, but for now let's move on.

2.1.4 Advanced Operators

There is one final function that awaits us; one final operator: multiplication! Multiplication might not sound like much compared to the geometry we just went through, but there's a reason why we chose to tackle it last. The reason is just that; it's not geometry. The specific domain of our domain specific language is geometry, and thus dealing with problems that lack an innate connection to geometry is inherently tricky.

That does not mean we can't build a function to multiply two Complex numbers, but it does mean we'll have to move outside the bounds of our DSL to accomplish it.

To wit; in regular math, complex numbers are normally written on the form "a + bi", I.E. reals + imaginaries. With that language in mind, basic algebra become a lot more intuitive. The expression `multiply (Complex (r1,i1)) (Complex (r2,i2))` in our DSL becomes $(x1 + y1i)(x2 + y2i)$. This can be solved the same way we solve any multiplication of additions; by multiplying each combination and adding the results:

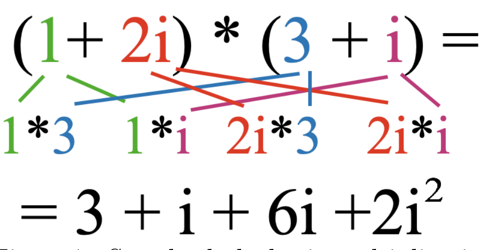


Figure 7: Standard algebraic multiplication

As such, by cleverly switching languages, we can find solutions to problems that our current DSL struggles with.

We can now define an operator in our own DSL using the information provided by traditional algebra to reach a correct answer. There is still one annoying complication remaining, however; figure 7 gives us one term containing i^2. How will we make that work with how we've defined our datatype? We never added a way to write powers!

If only there was a convenient comment regarding the definition of i that we brought up at the beginning of the section and fully expect you to have forgotten about by now.

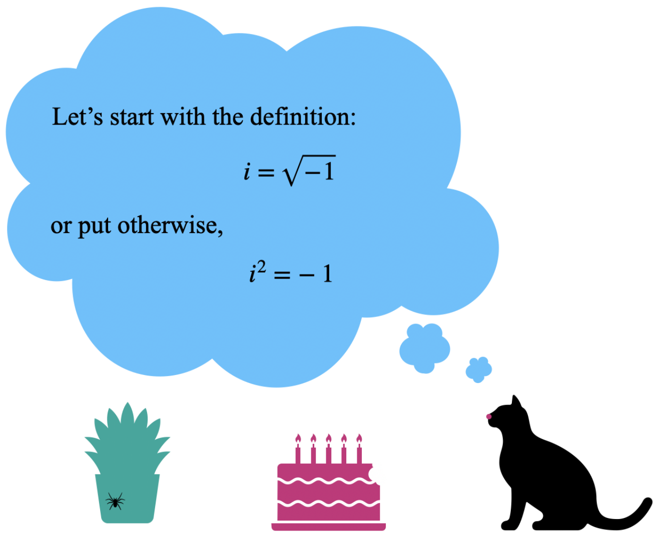


Figure 8: Comment regarding the definition of i that we brought up at the beginning of the section and fully expect you to have forgotten about by now

Oh wait, there is just such a thing! i^2 is the same as -1! We can now complete our final function and also operator.

```haskell
multiply :: Complex -> Complex -> Complex
multiply (Complex (r1,i1)) (Complex (r2,i2))
  = Complex (r1*r2 - i1*i2, r1*i2 + r2*i1)
```

This code may look a bit overwhelming, but it's quite simple, really. We're just inserting the rule from basic algebra above, except with the final i^2 expression replaced with a negation, in accordance with the definition of i.

Now we are done with our very first DSL. It might have been quite simple, but that's sometimes all that's needed. It is also an example of what we call a shallow DSL (contrasted by a deep DSL). By calling a DSL shallow we mean that the DSL has the syntax and the semantics on the same level; see 2.2 for an example of a deep DSL.

Note that we could have chosen another way of implementing our DSL for the same domain and gotten a DSL that works well for other parts of the domain. For example, we could have used the "absolute value + principal argument" method mentioned in the "complex geometry" section. This is called the "polar form" and would've made the `multiply` operator a breeze, but `whatsreal`, `whatsimaginary`, `add` and `sub` would've all been much, much harder.

As noted at the very top, complex numbers are an entire field of study. If we were to cover all there is to know about them, this section would take up a small library's worth of text and require an actual budget to write. The above should be just enough to give a basic understanding of the concept, along with all operators used in the sections below. Just remember that the coordinate comparison, while undeniably useful, does not apply to every possible operation and you'll do fine.

That said, if you'd like to see a slightly more extensive version, check out the appendix.

2.2 Integrals

Say we've got a function. This function can be expressed as a curve on a plane. This line separates the plane in two sections; one above and one below the line.

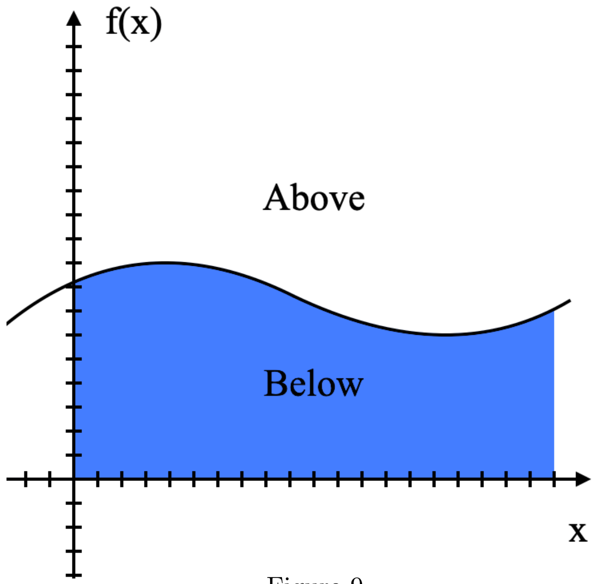


Figure 9

How large of an area exists below the curve?

2.2.1 Setting up the Language

A problem like the one described above is normally solved with an integral. To solve an integral, we'll first need a language that can describe the function (which in turn describes the line).

Unlike complex numbers, there is no one uniform expression that can describe any function. Polynomials, however, get pretty close, so let's use those as a basis:

```haskell
data Funk =
  Constant Double
  | X Double
  | XSquare Double
  | Add Funk Funk
  deriving (Show)
```

The idea behind polynomials is to split an expression into smaller chunks that are added together to create a greater whole. Each line in the above datatype is a "word" in our DSL. The constant should be self-explanatory; it's a constant value, expressed as a Double. "X Double" is our way of writing a (so far) undefined variable on the form "aX", i.e. 5X, 0.33X, or indeed just 1X. XSquare uses the same basic concept; "aX^2" can be 5X^2, 0.33X^2, etc. Finally, "Add" is used to string our words together into "sentences". As an example, the polynomial function "$5 + 4X^2$" would in our DSL be written as

```haskell
test :: Funk
test = Add test1 test2 where
    test1 = Constant 5
    test2 = XSquare 4
```

Simple enough, and easily expanded, too. If we ever wanted any other functionality, be it roots, pies, or trigonometric functions, we'd simply need to add a word for them. As such, we can keep our "language" limited to the above for now; the rest will be added in the sections that use them.

We're still not ready for integrals, however. First, we'll also need a function that takes the sentences(functions) we just created and calculates them to produce a result. In practice, this is simplicity itself; we'll simply replace each word with their algebraic equivalent and let Haskell roleplay as a calculator for a moment. To wit:

```haskell
calculate :: Funk -> Double -> Double
calculate (Constant double) value = double
calculate (X double) value        = double * value
calculate (XSquare double) value  = double*(value^2)
calculate (Add funk1 funk2) value = (calculate funk1 value) + (calculate funk2 value)
```

The "value" input in the above code is whichever value we desire X to be.

Exercise 2. Expand the DSL with a word for powers other than squares. Like the other words, it should include a constant multiplier and integrate correctly with `calculate`, above. Hint: as evidenced by `Add`, there is no rule saying a word can only include one value.

2.2.2 Brute Force Mathematics

Right, with our basic language defined, it's time to take a first stab at that graph.

The traditional first step to solving an integral is to, instead of look for an entirely accurate answer, instead look for an estimated answer.

To wit, let's use our test function from above ($5 + 4x^2$), and say we want to know how much space exists beneath the line and between x=2 and x=5. To do this, we could take the average of these numbers, (2+5)/2=3.5, and see how much space would exist between the line if this average was true throughout the function.

```haskell
-- bfIntegral (brute force integral)
--input: function, startvalue, stopvalue
--output: result
bfIntegral :: Funk -> Double -> Double -> Double
bfIntegral funk start stop = (calculate funk ((start+stop)/2)) * (stop-start)
```

There, quick and easy. Sadly not accurate. Our main problem is that our test function does not grow linearly. This means that taking a value "in the middle" of the graph and expecting the values on either side to balance each other out is foolish.

Rather than despair, however, let's try to make our integral a little more accurate. The way we go about this is that rather than use an average for the entire graph, we cut the problem into equally sized chunks, calculate each chunk individually, and add the results together.

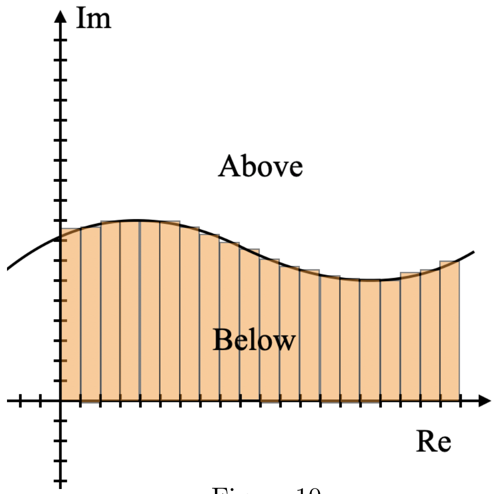


Figure 10

```haskell
-- bfIntegral' (brute force integral alternative)
--input: function, start, end
--output: result
bfIntegral' :: Funk -> Double -> Double -> Double
bfIntegral' funk start end
  | start >= end = 0
  | otherwise    = (calculate funk start) + (bfIntegral' funk (start+1) end)
```

We made each chunk precisely 1 wide for simplicity, using a simple recursive function. The result should be considerably more accurate now, though still not perfect.

We also learnt an important lesson. See, the first example used chunks as well. Or rather, a single chunk. By cutting the chunk into smaller chunks, we increased accuracy. This is something we can repeat. In theory, we can repeat it infinitely; cutting our problem into an infinite number of infinitely small chunks. If each size reduction on the part of the chunks brings us closer to the true answer, and we reduce the size infinitely, then we will become infinitely close to the true answer. I.E. We will have the true answer.

2.2.2.1 In practice, no computer can actually calculate infinity, but they can get close enough for most purposes. So let's give that a shot and see what happens;

Exercise 3. Create a version of the brute-force Integral operator that uses 100 times the width worth of chunks.

```
bfIntegral'' :: Funk -> Double -> Double -> Double
-- bfIntegral'' (brute force integral: electric boogaloo)
--input: function, start, stop
--output: result
```

If this is still not enough.. well, there's still one subsection left to read. Beware though, it'll require a slightly more abstract view of mathematics.

2.2.3 An Elegant yet Primitive Solution

deep breath

During prior math courses, you should've ran into derivative functions.

A derivative function, conceptually speaking, is a function transformed from describing a value for a specific input, to describing how that value changes depending on input.

You may realize that this sounds eerily similar to what we are hoping to do with integrals; let's take another look at the figure from the beginning of this section:

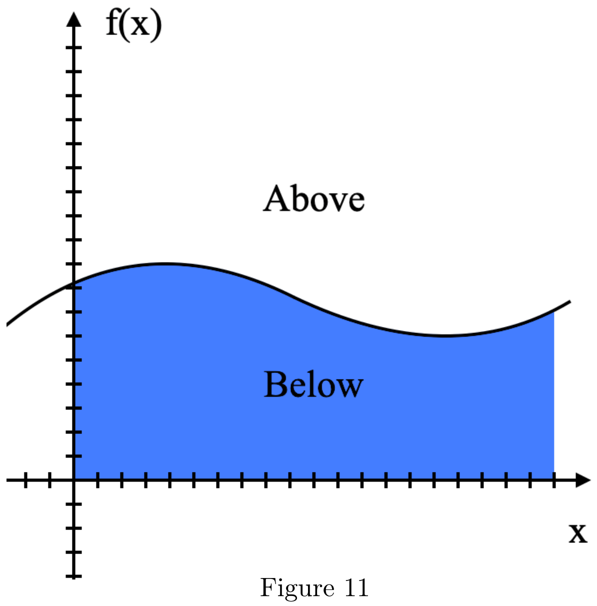


Figure 11

We are looking for the area underneath a curve described by a function; a line that describes the change in that value depending on input. We are looking for the exact opposite of a derivative function. We are looking for what mathematicians refer to as ”primitive functions”. This name is an homage to the idea that our function has already been derived, and we are seeking its ancient, gnarled, ”primitive” ancestor. In this manner, all functions can be considered to exist in secret pairs; a primitive ancestor, and a punk derivative.

For the sake of brevity, let's not go indepth on how derivatavation function on a micro-level, and instead limit ourselves to the more commonly used laws of derivation. Specifically, let's invert them:

```haskell
primitivize :: Funk -> Double -> Funk
primitivize funk constant = Add (primitivize' funk) (Constant constant)

primitivize' :: Funk -> Funk
primitivize' (Constant double) = X double
primitivize' (X double)        = XSquare (double/2)
primitivize' (XSquare double)  = -- Your implementation of powers goes here; power=3, double/3
primitivize' (Add funk1 funk2) = Add (primitivize' funk1) (primitivize' funk2)
```

Assuming the rules of derivation are known, the bulk of primitivize' shouldn't be overly surprising. When deriving, each X is multiplied with their power, and then said power is reduced by one. Thus, if we're seeking the primitive function version of the function, we instead increase the power by one, and then divide by the new power. Standalone constants are removed when deriving a function. Thus, any remaining constants in the derived version of the function must've once had an attached variable. In addition, we must consider the possibility that there once was another constant that was lost on derivation; this is why we must supply a replacement constant when finding the primitive function. For integral purposes, this constant will always be zero, but it's still important to know that it technically exists. Add remains unchanged.

Going back to the example used in the brute force solution, we should now be able to get a perfect answer:

```haskell
primitiveTest :: Funk
primitiveTest = primitivize test 0
(calculate primitiveTest 5) - (calculate primitiveTest 2)
```

Exercise 4. Extend support for other powers to this form of integral.

Exercise 5. ADVANCED: Expand the DSL with a word for a complex constant, plus associated functionality for both types of integral.

Once set up, the primitive solution is indeed elegant, but it does have one major flaw; it only works so long as we can find a primitive version of the function, and depending on the operators involved, this might be easier said than done.

Finally, there are more extensive versions of the above Haskell code in the appendix, if you're interested.

3 Laplace transform

One very important tool when learning control theory is the Laplace transform. It's mostly used to solve differential equations. We will explain how to do that, as well as some other important rules often used in control theory. However, first we need to do some groundwork and build what will essentially be the scaffolding for the Laplace transform. There are two major tools we will use to implement the Laplace transform: A DSL for Complex Numbers and a datatype representing mathematical expressions, aptly named `Expression`. We will also include `FunNumInst` from DSLsofMath[1], which

[1]https://github.com/DSLsofMath/DSLsofMath/

allows us to write addition etc. between functions (in more technical language: it's a `Num` instance for functions).

The DSL we will use for Complex Numbers is an extension of the one developed in section 2.1. The actual implementation is not important, but if you want to read it, see listing ??.

```haskell
module Laplace where
import ComplexNumbers
import FunNumInst
```

3.1 The Expression datatype

Like in section 2.2, we create a recursive datatype. We will not state it in it's entirety here, but you can find it in Appendix A. What we will do is show some parts of it and talk about how they fit together.

First, we declare a new datatype, `Expression`, which takes a parameter `a`.

```haskell
data Expression a = ...
```

The parameter is the type of the expression: we will mostly use `Expression Complex`, which in essence is a syntactic representation of a function of a complex variable. After that, we give some examples of how to construct `Expression`s. The two most important of these are `Const a` and `Id`. These represent the constant function and the identity function, respectively.

Example 2. Create the constant $2 + 3i$ as an `Expression`.

Solution.

```haskell
testExp1 :: Expression Complex
testExp1 = Const (Complex (2, 3))
```

The next four are the four arithmetic operations for `Expression`, e.g. we can create an `Expression` by adding two other together. Instead of just naming them `+`, `*`, etc. they are instead named with colons around them, e.g. `:+:`. This is in order to highlight that it is addition between expressions.

Example 3. Now, you might wonder what an actual expression might look like. Well, look no further:

```haskell
testExpression :: Expression Complex
testExpression = Sin (Const 3 :*: Id) :+: Exp (Negate (Id))
```

which in math can be written

$$f = t \mapsto \sin(3t) + e^{-t}$$

One way to visualise `Expression`s is to drawn them in syntax tree, see figure 12.

Exercise 6. Create your own test expression.

Now, we've mentioned how `Expression` is a syntactic representation of some sort of semantic expression. What does this mean and what is the semantic expression? Well, the expressions we have written seem to represent mathematical expressions alright, but we can't really do anything with them. Besides, there are many mathematical properties we aren't really capturing with `Expression` — for example, `Const 1 :+: Const 2 == Const 2 :+: Const 1` is `False`, they aren't equal. Of course, we know that mathematically $1 + 2 = 2 + 1$. How do we solve this disconnect between math and our

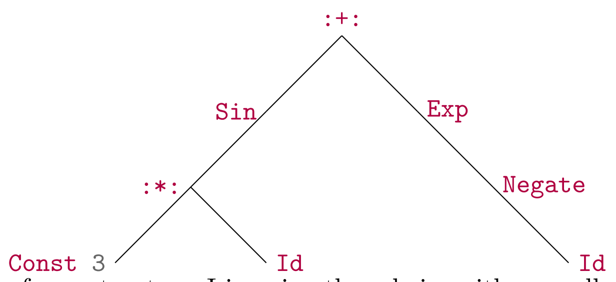


Figure 12: Sketch of a syntax tree; I imagine there being either small circle around each command or possibly just space (like in Communicating Mathematics ...)

$$\begin{tikzcd}\text{Syntax} \arrow[d, "\texttt{eval}"] \\ \text{Semantics}\end{tikzcd}$$

Figure 13: Diagram showing the syntactic and semantic levels, as well as the eval function between them. (The arrowhead is a little small)

DSL? We've previously mentioned that we'll distinguish between syntax and semantics. So far, this section has discussed syntax a lot, but not really semantics. What do our `Expression`s represent? That's where the semantics come in, they are the meaning of our `Expression`s. To move from the syntactic level to the semantic, we will define a function `eval` which takes us from the syntactic level to the semantic one. We will then have to define the semantics somehow so that this is actually the case. The full function is quite long and works through pattern matching.

```haskell
shift :: Num a => a -> (a -> a) -> (a -> a)
shift tau f = \t -> f (t-tau)

eval :: (Num a, Floating a, Eq a) => Expression a -> a -> a
eval (Const c)            = \t -> c
eval Id                   = id
eval Pi                   = eval (Const pi)
eval (e1 :+: e2)          = eval e1 + eval e2
eval (e1 :*: e2)          = eval e1 * eval e2
eval (Shift tau e)        = shift tau (eval e) -- shifts e by tau, i.e. begins after tau seconds
eval (Negate e)           = negate (eval e)
eval (Exp e)              = exp    (eval e)
eval (Sin e)              = sin    (eval e)
eval (Cos e)              = cos    (eval e)
eval (Tan e)              = tan    (eval e)
eval (Asin e)             = asin   (eval e)
eval (Acos e)             = acos   (eval e)
eval (Atan e)             = atan   (eval e)
eval (Integral   e e0)    = undefined -- TODO
eval (Derivative e e0)    = undefined -- TODO
```

Example 4. Run `eval` on `testExpression` above, insert some values and compare to the mathematical function.

Solution. The expression

```
 -- TODO
tmp = eval testExpression
 -- testExpression = Sin (Const 3 :*: Id) :+: Exp (Negate (Id))
```

Exercise 7. Create your own expression and run the eval function on it. Make sure to test some values, ensuring that the semantic function gives the right values.

3.1.1 Some conventions and notations

Sometimes in math books there is some notational abuse: does $f(t)$ mean the function f or f applied to the value t? We can often understand which is meant from context, but not always. We will avoid this problem by being explicit when a function is applied to a value or not: the function will be written f, the function applied to a value will be written $f(t)$. Sometimes we need to write out the functions definition explicitly. If so, we will write it out using $\mapsto$: for example, writing that f is the exponential function will be $f = t \mapsto e^t$.

Another convention we will use is that when talking about a function named with a letter, for example f, we will sometimes denote the Laplace transform of said function with the capital version of the letter, e.g. F. Put more succinctly: F is the Laplace transform of f. This could be confused for the primitive function of f, but we will be explicit whenever we talk about primitvie functions instead of Laplace transforms. We know, we haven't even spoken about the Laplace transform yet, but keep it in mind for the following sections. One notation we won't use but which might show up in other literature is to write $\tilde{f}$ for the Laplace transform of f.

Some conventions we will use that are not strictly related to the Laplace transform is that we will write addition and multiplication between (real and complex) functions the same way as we do between (real and complex) numbers, using the ordinary symbols $+$ and $\cdot$, i.e. we will let the operators have polymorphic types. We will however be consistent and write out $\cdot$ instead of just keeping two functions next to each other (e.g. $x \cdot y$ instead of xy).

3.2 The definition of the Laplace transform and corresponding types

Before we start: this section will start a little bit math heavy — don't be discouraged! We will give the mathematical definition of the Laplace transform, take it apart a little and analyse the types. When we're done with that, however, we will just skip the calculating and show some shortcuts. If you're feeling like the math just isn't making sense, you can safely jump to section 3.3.

Now, no more skirting around the issue, here's the definition of the Laplace transform $\mathscr{L}$:

$$\mathscr{L}\{f\} = s \mapsto \int_0^\infty e^{-st} f(t) \mathrm{d}t.$$

There's a lot going on here. On the left hand side we have two symbols: $\mathscr{L}$ and f. We see that f shows up on the other side of the equation, meaning it's bound over the equality. On the right hand side, there are a few things to see: we have an integral from 0 to infinity, we have the function f again, and we have an exponential function. Note that due to t being bound by the integral, the exponential is a function of s in the wider scope. Thus the Laplace transform of a function f is an integral of that function multiplied by an exponential function.

What variables are free in this expression? After some inspection, we might see that f and s are the variables of the equation: 0 and ∞ are obviously not variable, and t is bound by the integral. As f is bound over the equation, it acts as a variable. Thus the final type of the Laplace transform can be identified as

```haskell
laplace :: (R -> R) -> (Complex -> Complex)
```

or using `Expression`:

```haskell
laplace :: Expression R -> Expression Complex
```

i.e. it takes a real function and returns a complex function.

Now, unfortunately we have no way of binding a variable with an integral, and thus implementing the definition of the Laplace transform does not work well (and besides, it's an integral including infinity, it's not going to be very easy to implement either way). One possible naive implementation might look like

```haskell
laplace' :: Expression R -> Expression Complex -> Expression Complex
laplace' f = Integ (Const 0) (Infinity) (exp :*: f)
                where exp = Exp (Negate (Const s) :*: Id)
```

but this is clunky and does not reflect how we typically use Laplace transforms.

What we will do is to write `laplace` using pattern matching. This will act similarly to what we will call the tabular approach, i.e. calculating the Laplace transform using the rules and formulas found in the table.

3.3 Common rules, how to use them

Throughout this section we will show you how to use the rules in the table to calculate Laplace transforms, since that makes life a lot easier.

The Laplace transform $\mathscr{L}$ will be implemented as the command `laplace`, e.g. `laplace e` is the Laplace transform of the expression `e`. See `laplace` below for the table adapted to our DSL. Note: this is not the full implementation, it's just a refrasing of the table. For the full implementation, see appendix (Listing 8)

Before we start, one important property: if two functions have the same Laplace transform, then they are the same function. In our DSL this means that `eval (laplace e)`

3.3.1 Superposition

One of the most important rules of the Laplace transform is what's often called either superposition or linearity. Informally (and a little imprecisely), it means that we know the Laplace transfers of two functions, the Laplace transform of their sum is the sum of the Laplace transforms. Mathematically, it's often phrased

$$\mathscr{L}\{\alpha f + \beta g\} = \alpha\mathscr{L}\{f\} + \beta\mathscr{L}\{g\}.$$

Thus, if we know that the Laplace transforms of f and g are F and G, respectively, then the Laplace transform of $f + g$ would be $F + G$.

Table 1: Two tables of the Laplace transform.

(a) Table of Laplace transforms in more mathematical writing. This is a shortened and reworked version of the table allowed on the exam.

f	F
$f_1 + f_2$	$F_1 + F_2$
f'	$s \mapsto s \cdot F(s) - f(0)$
$f^{(k)}$	$s \mapsto s^k F(s) - s^{k-1} f(0) - s^{k-2} f'(0) - \dots - f^{(k-1)}(0)$
$t \mapsto \int_0^t f(\tau) \mathrm{d}\tau$	$s \mapsto \frac{1}{s} \cdot F(s)$
$t \mapsto f(t - T)$	$s \mapsto e^{-sT} F(s)$
$t \mapsto e^{-at} f(t)$	$s \mapsto F(s + a)$
$t \mapsto \int_0^t f_1(\tau) \cdot f_2(t - \tau)$	$F_1 \cdot F_2$
δ	1
1	$s \mapsto \frac{1}{s}$
$t \mapsto t$	$s \mapsto \frac{1}{s^2}$
$t \mapsto e^{-at}$	$s \mapsto \frac{1}{s+a}$

(b) Laplace transform in code; see 8 for full definition

```haskell
-- rules
laplace :: Expression Complex -> Expression Complex -- TODO: maybe instead do Time->Freq
laplace (e1 :+: e2) = laplace e1 :+: laplace e2 -- Superposition
laplace (Const a :*: e) = Const a :*: laplace e -- scaling
laplace (Derivative e e0) = Id :*: laplace e :-: (Const e0) -- L {f'} = \s -> s * L {f} - f(0)
laplace (Integral e e0) = laplace e :/: Id -- L {int_0^t f(x) dx} = s -> L {f} /s
laplace (Shift tau e) = Exp (Id :*: Const tau) :*: laplace e -- TODO: double check signs
  -- TODO: double check signs on next two
laplace (Exp (Negate (Const tau :*: Id)) :*: e) = Shift tau (laplace e)
laplace (Conv e1 e2) = laplace e1 :*: laplace e2 -- convolution turns into multiplication
-- transforms
laplace (Impulse) = Const 1 -- L {delta(t)} = \s -> 1
laplace (Const 1) = Const 1 :/: Id -- L {a} = \s -> a/s; a = 1 => L {1} = 1/s
laplace (Id)      = Const 1 :/: (Id :*: Id) -- L {t} = \s -> 1/(s^2)
laplace (Exp (Negate (Const a :*: Id))) = Shift (negate a) (Const 1 :/: Id)-- TODO: Check signs
laplace (Const 1 :-: Exp (Negate (Const a :*: Id))) = Const a :/: (Id :*: Shift a Id) -- TODO: Doubl

laplace (Exp (Negate (Const a)) :-: Exp (Negate (Const b))) =
  Const (b - a) :/: (Shift (negate a) Id :*: Shift (negate b) Id)
laplace (Id :-: (Const 1 :-: Exp (Negate (Const a) :*: Id)) :/: Const a) =
  Const a  :/: (Id :*: Id :*: Shift (negate a) Id)
laplace (Const 1 :-: (Const 1 :+: Const a :*: Id) :*: Exp (Negate (Const a) :*: Id)) =
  Const (a*a) / (Id :*: Shift (Negate a) (Id :*: Id)) -- Const a :*: Const a instead of Const (a*a)?
-- TODO: Doublecheck signs on the next two
laplace (Exp (Negate (Const a :*: Id)) :*: Sin (Const omega :*: Id)) =
  Const omega :/: (Shift a (Id :*: Id) :+: (Const (omega*omega)))
laplace (Exp (Negate (Const a :*: Id)) :*: Cos (Const omega :*: Id)) =
  (Shift a Id) :/: (Shift a (Id :*: Id) :+: (Const (omega*omega)))
```

Superposition is a rule which might on first glance seem like it doesn't say anything. However, it's also often used by students without them knowing. If we'd want to calculate $\mathscr{L}\{t \mapsto e^t + e^{-t}\}$, for example, knowing that

$$\mathscr{L}\{t \mapsto e^t\} = s \mapsto \frac{1}{s-1}$$

and that

$$\mathscr{L}\{t \mapsto e^{-t}\} = s \mapsto \frac{1}{s+1}$$

does not help us anything if we don't have superposition. Of course, we do have superposition, so we know that

$$\begin{aligned}\mathscr{L}\{t \mapsto e^t + e^{-t}\} &= \mathscr{L}\{t \mapsto e^t\} + \mathscr{L}\{t \mapsto e^{-t}\}\\ &= s \mapsto \frac{1}{s-1} + \frac{1}{s+1}.\end{aligned}$$

transforms. Thus superposition is one of our most useful tools when calculating Laplace transforms.

But how about calculating Laplace transforms of `Expressions`? Luckily, superposition is fairly easy to implement in our DSL, so we can use the same rules to calculate transforms of `Expressions`! We get the following snippet:

```
laplace (e1 :+: e2) = laplace e1 :+: laplace e2
```

i.e. the Laplace transform of a sum of `Expressions` is a sum of `Expressions` that are the result of applying `laplace`, just as we wanted. Note that we have `:+:` between both the expressions `e1` and `e2` and their respective Laplace transforms, since they are all of type `Expression`. Had `laplace` not returned an `Expression`, we might need to define a new addition.

3.3.2 Derivative rule

Another very important rule regards the Laplace transform of a derivative. This is important because it helps us solve (some) differential equations a lot easier. It's also used to more easily implement and calculate PID controllers.

Informally, the rule can be described as turning differentiation into multiplication (and subtraction of an initial value). Mathematically, this can be written

$$\mathscr{L}\{f'\} = s \mapsto s \cdot \mathscr{L}\{f\}(s) - f(0)$$

i.e. if we have a derivative of a function, the corresponding Laplace transform is the Laplace transform of the function (not the derivative) multiplied with id. Finally we subtract the function evaluated in 0. It's not uncommon for $f(0) = 0$, which makes the rule even easier.

In our DSL, the rule is represented:

```
laplace (Derivative e e0) = Id :*: laplace e :-: (Const e0)
```

i.e. the derivative of `e` with inital value `e0` has Laplace transform `Id` multiplied with the Laplace transform of `e` minus the constant `Const e0`.

Example 5. Solve the differential equation

$$f = -f', \quad f(0) = 1$$

by applying the Laplace transform on both sides of the equation and setting them equal.

Solution. Unfortunately, we can't just ask our DSL to solve the equation - it isn't advanced enough. However, we can use some pseudocode to reason about it. Remember that if two functions have the same Laplace transform they are equal. In our case, this means that if we can find a function such that `laplace e == laplace (Derivative e 1)`, then we know that `e` is a solution to the diffential equation.

```
  laplace (Derivative e 1)
== {- derivative rule -}
  Id :*: laplace e :-: Const 1
== {- from differential equation -}
  laplace e
```

So far so good. We've shown that for `e` to be a solution to the differential equation, we also need it to satisfy the equation `laplace e == Id :*: laplace e :-: Const 1`, which translated into mathematics means $\mathscr{L}\{f\} = s \mapsto sF(s) - 1$.

Now you might think "We've exchanged one equation for another, how does that help us?", which is a valid question. So what is the rationale for this? Well, when given the choice to solve one of the two following equations

$$f = f', \quad f(0) = 1$$
$$F = s \mapsto sF(s) + 1$$

we know which one we'd prefer.

Hint: it's the second.

The first is a differential equation, which requires great guesswork or some more advanced methods to solve, while the second one is solvable using simple algebra. Solving the second equation results in the function $F = s \mapsto \frac{1}{s-1}$. Looking at your favorite table of Laplace transforms, you might find somewhere the formula $\mathscr{L}\{t \mapsto e^{-at}\} = s \mapsto \frac{1}{s+a}$ (or `laplace (Exp (Const a :*: Id)) = Const 1 :/: (Id - Negate (Const a))`, and if that's the case, we're flattered that's your favorite table of transforms!) If we let $a = 1$, we see that this is the function we found! Thus we can conclude that the function we're looking for is $t \mapsto e^{-t}$.

Now, we're technically done. However, a good idea when solving differential equations (using the Laplace transform or another method) is to check whether the found solution actually satisfies the equation.

Typically this means two things: differentiating the found function (maybe several times, depending on the order of the equation), and then inserting the result in one side of the equation, seeing if it's possible to get to the other.

In our case, this means differentiating $t \mapsto e^{-t}$. From single variable calculus (or Beta) we get

$$t \mapsto \frac{\mathrm{d}}{\mathrm{d}t} f(t) = t \mapsto \frac{\mathrm{d}}{\mathrm{d}t} e^{-t} = t \mapsto -e^{-t} = -f,$$

i.e. the function satisfies the equation.

Now we just need to check the initial value:

$$f(0) = e^{-0} = 1,$$

which is what we wanted, and thus we're done.

But what are the types included in this rule?

Well, as previously mentioned, `laplace` is a function of type `(R -> R) -> (Complex -> Complex)`, i.e. it takes a real function and returns a complex function.

3.3.3 Integral rule

Just like we can use the Laplace transform to transform differentiation into multiplication, we can use it to transform integration into division. This is also used for PID controllers.

Mathematically, the rule is

$$\mathscr{L}\left\{t \mapsto \int_0^t x \mapsto f(x)\mathrm{d}x\right\} = s \mapsto \frac{\mathscr{L}\{f\}}{s}.$$

Note that the integral is a function of t.

In our DSL, this is written:

```haskell
laplace (Integral e e0) = laplace e :/: Id
```

3.3.4 Convolution

The convolution of two functions f and g is written $f \circledast g$, defined

$$f \circledast g = \int_0^t f(\tau)g(t-\tau)\,\mathrm{d}\tau.$$

Convolution is used in many applications, but the one that's likely most important is when looking at a system.

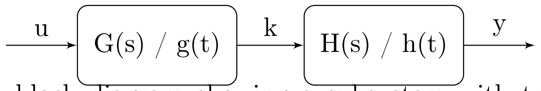


Figure 14: A simple block diagram showing a subsystem with transfer function G(s) feeding into a subsystem with transfer function H(s). u(t) and y(t) are the in- and output signals, respectively. k(t) is the input of the H-system and output of the G-system. Adapted from 15

Typically, what we want to do is Laplace transform this expression, as that turns convolution into multiplication.

Mathematically, this means

$$\mathscr{L}\{f \circledast g\} = \mathscr{L}\{f\} \cdot \mathscr{L}\{g\} = F \cdot G$$

and in our DSL, this is written

```haskell
laplace (Conv e1 e2) = laplace e1 :*: laplace e2
```

This turns the expression above into

3.3.5 Time shift

The time shift rule allows us to start a system later, i.e.

3.3.6 Exponential decay (frequency shift)

3.4 Inverse laplace transform

So far, whenever we've found an expression for the Laplace transform of a function, we've handwaved it by saying "and we recognize that this other function transforms into this function, thus it should be the answer". This gives the right answer (obviously, or else we wouldn't teach you that...), but it's not entirely rigorous (and mathematicians love rigor!)

The "proper" way to do it is by using what's (appropriately) called the inverse Laplace transform.

Like the regular Laplace transform, the inverse Laplace transform is defined using an integral. This one is even more clunky than the ordinary Laplace transform, however. The definition goes that the inverse Laplace transform of a function F, denoted $\mathscr{L}^{-1}\{F\}$, is given by

$$\mathscr{L}^{-1}\{F\}\,s = \frac{1}{2\pi}\lim_{T\to\infty}\int_{\gamma-iT}^{\gamma+iT} e^{st}F(s)\mathrm{d}s.$$

Now, your reaction here might be: what's this monstrosity? An integral where both endpoints are complex numbers, and a limit outside of that? Luckily, we don't have to calculate it[2], since we can just use the rules to find the inverse instead!

4 Transfer Functions and LTI Systems

```haskell
{-# LANGUAGE GADTs              #-}
{-# LANGUAGE StandaloneDeriving #-}

module LTIandTF where
  import Lib
```

A transfer function is a function describing how a linear time-invariant (LTI) system (or part of such a system) transforms a time-varying input signal to give the system output. A transfer function is stated in the (complex) frequency domain (i.e. after using the Laplace transform). Thus—from the naming convention from the Laplace transform—a function f describing an LTI system has a corresponding transfer function labelled F.

In order to understand transfer functions well, we need to first gain some insight into LTI systems:

4.1 Linear Time-Invariant Systems

A linear time-invariant (LTI) system is a system that is only indirectly dependent on time (t). More precisely, input signals may be linearly combined (i.e. superposed) or differentiated/integrated with respect to time. As a consequence, an LTI system reacts exactly the same regardless of what value t has, given identical inputs. (As a real-life parallel: the shower will change the water temperature in the same way when turning the valve, regardless of whether you do it in the morning or in the afternoon, today or tomorrow). We can express this property in code in the following way:

```
sys :: LTI; d :: Time; f :: Time -> Amplitude
shift :: Time -> (Time -> Amplitude) -> (Time -> Amplitude)
shift d f = \t -> f (t - d)
```

[2]However, if you want to understand how to calculate it, we recommend "A First Course in Complex Analysis", which is freely available at http://math.sfsu.edu/beck/complex.html

```
sys (shift d f) == shift d (sys f)
```

In this code a signal is represented by the type `Time -> Amplitude`, and `shift d f` shifts a signal `f` with some time `d`. `sys` represents an LTI system. We will explore its type in a bit. We can then express the property of linearity for LTI systems in code like this:

```
a, b :: Double; f, g :: Time -> Signal
(+) :: (a -> b) -> (a -> b) -> (a -> b)
f + g = \x -> f x Prelude.+ g x
-- Scales the signal (amplitude)
scale :: Double -> (Time -> Signal) -> (Time -> Signal)
scale k f = \t -> k * f t

sys (scale a f + scale b g)
    == scale a (sys f) + scale b (sys g)
```

In the code above, `scale k f` scales the signal amplitude of f with k.

These systems are described with transfer functions and what we call LTI-functions (i.e. the inverse Laplace transform of the transfer function), but in an unusual way, which turns out to make the math easier, but tends to impede understanding. Let's take a closer look at the types involved to understand this:

4.2 Types

Let's start with the LTI system itself. It takes an input signal ($u(t)$), transforms it, and yields an output signal ($y(t)$). The first thing to take notice of is what type the in- and output signals have:

$$Time = \mathbb{R}^{>0},\ Signal = \mathbb{R}$$
$$u, y : Time \to Signal$$

The LTI system, *lti*, transforms its input signal in full. In other words, *lti* is a second-order function—mapping a (single-parameter) function to another such function:

$$lti : (Time \to Signal) \to (Time \to Signal)$$

The LTI system needs the entirety of its input function—and not just the momentary value at t—in order to transform it into the output function. It turns out that we can describe this transformation with another function of the same type as the in- and output functions. This is what we call the LTI-function, whose Laplace transformation is the system's transfer function.

Because of this roundabout way of describing this system, we cannot easily link two LTI systems together in an intuitive manner. Sure, we can define a system to be composed of two other LTI systems, like the following:

```
lti = lti1 . lti2
```

This is works well since *lti* is an endofunction (i.e. a function of type $a \to a$), but we do generally not have an expression for this transformation. What we do have, however, is the LTI-function (more on how it is defined in the next section; 4.3). The equivalent to composing LTI systems when working with LTI-functions (and in-/outputs to such systems since they have the same type) is convolution, which we saw (in 3.3.4) simplifies to regular multiplication when Laplace-transformed. With an input function $u(t)$, an LTI system `sys` with LTI-function $f(t)$, and an output function $y(t)$, we have the following relations (where `|*|` denotes convolution):

```
lti u == y
U = laplace u; Y = laplace y; F = laplace f
Y s == \s -> U s * F s
y t == \t -> u t |*| f t
```

Consider the LTI system shown in the following block diagram:

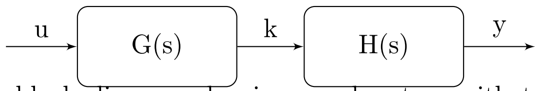


Figure 15: A simple block diagram showing a subsystem with transfer function G(s) feeding into a subsystem with transfer function H(s). u(t) and y(t) are the in- and output signals, respectively. k(t) is the input of the H-system and output of the G-system.

The functions U(s), G(s), K(s), H(s), and Y(s) are transfer functions derived from the LTI-functions u(t), g(t), k(t), h(t), and y(t). The LTI-functions are functions of time that yield a signal strength, so we label their types:

$$Time = \mathbb{R}^{\geq 0},\ Signal = \mathbb{R}$$
$$u, g, k, h, y : Time \to Signal$$

```haskell
type Time   = Real
type Signal = Real
u, g, k, h, y :: Time -> Signal
```

This determines the type of their transfer functions. Recall, $F(s) = \mathscr{L}\{f\}(s)$ and the type of the Laplace transform is $\mathscr{L} : (\mathbb{R}^{\geq 0} \to \mathbb{R}) \to (\mathbb{C} \to \mathbb{C})$, making $F(s) : \mathbb{C} \to \mathbb{C}$. Thus,

$$U, G, K, H, Y : \mathbb{C} \to \mathbb{C}$$

Note that even though we call all these functions transfer functions (and they are all of the same type), they do not all serve the same purpose. G(s) is the transfer function for the system represented by the left block in the block diagram and H(s) is the transfer function for the right one. U(s) is a transfer function representing the input signal. We can imagine this to be the transfer function of a system with $\delta(t)$ as input, feeding into the G-system (see figure 16). We here think of δ as a function that actuates the system. k(t) is an intermediate function sometimes written out to give a variable to multiple systems (this can be useful in more complex systems)—in this case $K(s) = U(s) \cdot G(s)$, i.e. the combined system of the input signal and the G-system. Y(s) is the transfer function for the full system *and* U(s). The transfer function for the entire system is therefore $Y(s)/U(s) = G(s) \cdot H(s)$.

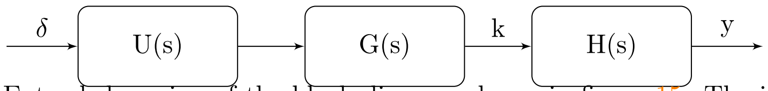


Figure 16: Extended version of the block diagram shown in figure 15. The input signal u(t) has been replaced by its transfer function with δ as input, acting as an actuator for the system. This is identical to the previous representation, but highlights how the input can be shown as a system with a transfer function U.

Let's summarise this with some relations:

```
type LTI = (Time -> Signal) -> (Time -> Signal)
sys :: LTI
f :: Time -> Signal; F :: C -> C

sys = (|*| f)
```

Or equivalently:

```
sys = \sig -> invlaplace $ laplace sig * F
```

where `sys` has LTI-function f and transfer function F.

4.3 Linear Time-Invariant Systems – Part 2

As mentioned above (in 4.1), a linear time-invariant (LTI) system is a system that is only indirectly dependent on time. An interesting property, stemming from this time invariance and the linear property, is that any sine function input into the system would still be a sine function, whereas any other type of wave function would change shape (not counting translation and scaling). We let the following represent a signal processed by an LTI system:

```haskell
data Signal a where
  -- Amp -> Freq -> Time-shift -> (a -> a)
  Sin   :: Num a => a -> a -> a -> Signal a
  Sum   :: Signal a -> Signal a -> Signal a
  Scale :: a -> Signal a -> Signal a
  Deriv :: Signal a -> Signal a
  Integ :: Signal a -> Signal a
    deriving instance Eq a => Eq (Signal a)
```

It should now be possible to take any Signal instance and simplify it to a single Sin constructor. Or, at least as long as frequencies match. But this is okay since an LTI system cannot change the frequency of a sine function.

```haskell
simplifySignal :: (Num a, Eq a, Floating a, Ord a) => Signal a -> Signal a
simplifySignal (Scale k (Sin a f s)) = Sin (a * k) f s
```

Recall how the derivative of $sin(t)$ is $cos(t)$, and that its antiderivative is $-cos(t)$, as well as the relationship $cos(t) = sin(\pi/2 - t)$, in order to find the Sin constructor form of the following:

```haskell
simplifySignal (Deriv   (Sin a f s)) = Sin (a * f) (-f) (pi/2 - s)
simplifySignal (Integ   (Sin a f s)) = Sin (-a/ f) (-f) (pi/2 - s)
```

Summing two sine functions isn't quite as neat (no need to pay attention to the details here). And as mentioned, we disallow differing frequencies:

```haskell
simplifySignal (Sum (Sin a1 f1 s1)
                    (Sin a2 f2 s2)) | f1 == f2 = Sin a f1 s where
                      a = sqrt $ q1 ^ 2 + q2 ^ 2
                      s = atan $ q2 / q1
                      q1 = a1 * cos s1 + a2 * cos s2
                      q2 = a1 * sin s1 + a2 * sin s2
```

Since we now have this line which is not guaranteed to evaluate to a Sin constructor (because f1 might not equal f2) and in order to propagate the function `simplifySignal` over the syntax tree, we include the following to the function definition:

```haskell
simplifySignal other = let ss = simplifySignal in case other of
  Scale k (Sum o1 o2)               -> Sum (ss $ Scale k o1) (ss $ Scale k o2)
  Deriv   (Sum o1 o2)               -> Sum (ss $ Deriv   o1) (ss $ Deriv   o2)
  Integ   (Sum o1 o2)               -> Sum (ss $ Integ   o1) (ss $ Integ   o2)
  Scale k a                         -> ss $ Scale k $ ss a
  Deriv   a                         -> ss $ Deriv   $ ss a
  Integ   a                         -> ss $ Integ   $ ss a
  Sum a b | ss a /= a || ss b /= b -> ss $ Sum (ss a) (ss b)
  Sum a@(Sum a1 a2) b@(Sum _ _)     -> ss $ Sum (ss $ Sum a1 b) (ss a2)
  Sum a b | maxfreq a < maxfreq b   -> ss $ Sum (ss b) (ss a)
  Sum (Sum c a@(Sin _ af _)) b@(Sin _ bf _)
                          | af == bf -> ss $ Sum (ss c) (ss $ Sum a b)
  _                                 -> other
```

Code 1: The function `maxfreq` returns the maximum frequency in the expression tree. See Appendix 4 for details.

Apart from the caveat with frequency, we can quite easily see how it is that sine functions retain their shape in an LTI system. We could exclude frequency from the definition of the data type in order to make things easier, but that would at the same time make it less expressive.

Lastly, let's define the semantics of the data type:

```haskell
evalSignal :: (Num a, Floating a, Eq a, Ord a) => Signal a -> (a -> a)
evalSignal     (Sin a f s) t = a * sin (f * t + s)
evalSignal     (Sum   a b) t = evalSignal a t + evalSignal b t
evalSignal     (Scale a b) t = a * evalSignal b t
```

Since we cannot find a derivative only from `evalSignal` a and `t`, we will have to rely on `simplifySignal` to evaluate `Deriv` and `Integ`:

```haskell
evalSignal sig@(Deriv a)   t = evalSignal (simplifySignal sig) t
evalSignal sig@(Integ a)   t = evalSignal (simplifySignal sig) t
```

4.4 Combining Transfer Functions

Since the LTI-functions are not endofunctions (i.e. are not functions *on* a set), they cannot be composed. So how do we combine several transfer functions in a larger system? To understand that, we first have to understand exactly what a transfer function is describing:

The LTI system is by definition time-invariant, so functions are not directly expressed in time, though, they vary with time. They are not expressed as a function of their input signal (but rather with their time-varying LTI-function), so how does it vary with input signal? The basic idea is that its LTI-function, $f(t)$, expresses the "impulse response" of the system, i.e. the output signal given an instantaneous impulse as input (the dirac delta function δ). Any other input to the system has to be convoluted with the system function (which was defined in 3.3.4). The LTI-function is defined this way because δ is the identity of the convolution operator—in other words:

```
(dirac |*|) == (id :: (Time -> Signal) -> (Time -> Signal))
```

Luckily, the Laplace transform of two convoluted LTI-functions is simply the product of their transfer functions.

```
\f -> laplace (\t -> diracdelta t |*| f t)
\f -> laplace (\t -> diracdelta t) * laplace (\t -> f t)
\f -> (\t -> step t) * (\t -> F t)
{- step = const 1 when t>=0 -}
\f -> \t -> F t
```

Because this "impulse response" is the same thing as what we've been calling "LTI-function," this is what the function is usually referred to as. But this name suggests that it is merely a special case among input responses, when, in fact, it is essential for defining LTI systems. We will henceforth use the commonly accepted term "impulse response," but do also think of the function as the function that specifies how an LTI system transforms its input.

4.5 Finding Transfer Functions

An LTI system is often described with a differential equation. In this section we will take a closer look at how to find a transfer function for an LTI system, given a differential equation.

We will use the following differential equation as a running example (as per usual, u is the input signal and y is the output signal):

$$9y(t) + 6\dot{y}(t) + \ddot{y}(t) = 3u(t) + 2\dot{u}(t)$$

We give the following representation for a linear differential equation with constant coefficients:

```haskell
data LCDE a where
  LCDE :: ([a],[a]) -> LCDE a

lcde = LCDE ([9,6,1], [3,2]) :: Fractional a => LCDE a
```

The type consists of two lists, representing each side of the equation. The leftmost (first) digit in each list represents the coefficient of the lowest order derivative. No constant term is included (since this would make both the math and the syntax more complicated).

$$\mathrm{LCDE}([a_0, a_1, ..., a_k], [b_0, b_1, ..., b_l]) \iff \sum_{n=0}^{k} a_n y^{(n)}(t) = \sum_{n=0}^{l} b_n u^{(n)}(t)$$

We provide a definition of the Laplace transform (introduced in the previous section), specific to this set of equations. For the purposes of the demonstration of transfer functions an exact definition is not essential, but is provided below in code snippet 2.

In a similar manner to the above type we use a list definition, but a third list is needed to represent additional constant–s^n-terms. In order to calculate this last list's coefficients, values of $y^{(n)}(0)$ and $u^{(n)}(0)$ are needed (notice how `laplace` in code snippet 2 requires these values). Specifically, c_n below (equation 1) is a sum of $a_i y^{(v)}(0) - b_j u^{(w)}$. (We are going to simplify this in a bit).

$$\mathrm{Transformed}([a_0, a_1, ..., a_k], [b_0, b_1, ..., b_l]; [c_0, c_1, ..., c_m]) \iff$$
$$\iff \sum_{n=1}^{k} a_n s^{n-1} Y(s) = \sum_{n=1}^{l} b_n s^{n-1} U(s) + C,$$

$$C = \sum_{n=0}^{m} c_n s^n \tag{1}$$

```haskell
data Transformed a where
  Transformed :: ([a],[a]) -> [a] -> Transformed a

--                    equation  f^i(0) g^j(0)   L{equation}
laplace :: Num a => LCDE a -> [a] -> [a] -> Transformed a
laplace (LCDE (lhs, rhs)) f0 g0 = transformed where
  transformed = Transformed (lhs, rhs) (zipWithL 0 (-) lc rc)
  lc = map sum [zipWith (*) f0 (drop n lhs) | n <- [1..length lhs-1]]
  rc = map sum [zipWith (*) g0 (drop n rhs) | n <- [1..length rhs-1]]
```

Code 2: A data type representing the equation after applying the Laplace transform to its left and right hand sides; A function definition for the Laplace transform of an instance of LCDE. The function `zipWidthL` referenced works like `zipWith` but extends the shorter list with the first parameter such that the output has the same length as the longer of the two lists. See appendix 3 for details.

Albeit a bit convoluted, we now have a representation of LCDE when it is laplace transformed. Notice how LCDE (lhs, rhs) $\mapsto$ Transformed (lhs, rhs) _. The hole is a polynomial in s, $p(s)$, that gets evaluated with values of $y^{(n)}(0)$ and $u^{(n)}(0)$. We make the assumption that all of these values are constant zero, $f^{(n)}(0) = 0$, since we can assume $u(t)$ is zero $\forall t \leq 0$, and thus we can drop these two extra lists of values and re-define the data type as follows:

```haskell
data Transformed a where
  Transformed :: ([a],[a]) -> Transformed a deriving Eq

laplace :: Num a => LCDE a -> Transformed a
laplace (LCDE (lhs, rhs)) = Transformed (lhs, rhs)
```

The values don't change here—only the semantic information. For our running example, lcde defined above, would evaluate to the following:

```haskell
lcdeTrans1 = laplace lcde
lcdeTrans2 = Transformed ([9,6,1], [3,2])
truth = lcdeTrans1 == lcdeTrans2
```

We expect our linear differential equation with constant coefficients to result in a rational expression, so let's define a data type for rational expressions.

```haskell
data RatExpr a where
  RatExpr :: Num a => ([a],[a]) -> RatExpr a
```

We find the rational expression simply by solving for G(s)=Y(s)/U(s):

$$\sum_{n=0}^{k} a_n s^n Y(s) = \sum_{n=0}^{l} b_n s^n U(s) \tag{2}$$

$$Y(s) \sum_{n=0}^{k} a_n s^n = U(s) \sum_{n=0}^{l} b_n s^n \tag{3}$$

$$Y(s)/U(s) = \sum_{n=0}^{l} b_n s^n / \sum_{n=0}^{k} a_n s^n \tag{4}$$

```haskell
solve2RatExpr :: Num a => Transformed a -> RatExpr a
solve2RatExpr (Transformed (lhs, rhs)) = RatExpr (lhs, rhs)
```

Again, this is in the end just a change of constructor name, i.e. changing the semantics. Combining the two we see nothing new:

```haskell
findTF :: Num a => LCDE a -> RatExpr a
findTF = solve2RatExpr.laplace
```

To make sense of this construction, we can make a simple eval-function:

```haskell
evalRE :: Fractional a => RatExpr a -> a -> a
evalRE (RatExpr (den,num)) s = numerator / denominator where
  numerator   = unravel num
  denominator = unravel den
  unravel []   = 0
  unravel list = head list + s * unravel (tail list)

tfG s = evalRE (findTF lcde) s
```

The function tfG above is the transfer function from the differential equation lcde.

We can use this for any arbitrary differential equation of the same form. Consider $3\dot{y}(t) + 2y(t) = \ddot{u}(t) - u(t)$, and the following code snippet:

```haskell
de = LCDE ([2,3],[-1,0,1]) :: Fractional a => LCDE a
tf2 = evalRE $ findTF de
```

4.6 Combining Systems

We previously talked about combining transfer functions of LTI systems (see 4.4). Let's now abstract away any details of transfer functions and focus on them as atomic blocks. In the code below, we represent a transfer function with a string name (like `TF "G"`).

```haskell
data LTI where
  TF :: String -> LTI
```

There are a few ways of combining systems. First, recall (see 4.2) how systems can be simply fed into each other. As a nod to the block diagrams, we now label this with an arrow:

```
  (:->) :: LTI -> LTI -> LTI
```

A signal may also be either split in two or combined with another signal by addition or subtraction (but to simplify matters we will only consider addition since a signal could be negated with a transfer function). We will represent this will a single operator to not have to deal with systems of several inputs or output:

```
  (:<>) :: LTI -> (LTI, LTI) -> LTI
```

The expression `a :<> (b, c)` symbolises splitting the signal from system `a` into `b` and `c`, and then adding them together. We can do longer chains of systems before adding them together by combining systems within the split pair. For example: `a :<> (b1 :-> b2, c)`.

On the semantic end, recall how connecting two control systems together will multiply their transfer functions. The function `evalLTI` below takes a dictionary[3] of transfer functions and evaluates the LTI data type to its transfer function.

```
evalLTI :: (String -> (C -> C)) -> LTI -> (C -> C)
evalLTI dict (TF s) = dict s
evalLTI dict (a :-> b) = \z -> evalLTI dict a z * evalLTI dict b z
```

A system splitting a signal, operating on it with separate subsystems, and adding the signals together can be thought of as one single system. Thus `a :<> (b, c) :-> d` should be evaluated as `eval a * eval sys * eval d`. Additionally, the system is comprised of a sum of subsystems `b` and `c`, and as such, (using the composition of systems mentioned in 4.2) `sysA . (sysB + sysC) . sysD` should mean the composite system should evaluate to `eval a * (eval b + eval c) * eval d`.

```
evalLTI d (a :<> (b, c)) = \z -> eval d a z * (eval d b z + eval d c z)
```

Let's try a couple of system definitions: first, consider the open-loop system defined by the block diagram from figure 15.

```
testSys1 = TF "G" :-> TF "H"
```

Evaluating this with `evalLTI` will as expected yield the product of the functions given as definitions for G and H:

```
evalLTI d testSys1 == \z -> (z-2)/(z-3) where
  d "G" = \z -> 1/(z-3)
  d "H" = \z -> (z-2)
```

If we try to define a closed-loop system, we will have a hard time. Even if we use recursion, we will not be able to do so, since `:<>` leaves no loose ends (i.e. inputs or outputs). In order to create feedback in the system we have to introduce a fourth constructor to the `LTI` data type:

[3] A dictionary is a function mapping a string name to a value. `dict :: Num a => String -> (a -> a)` in this case.

```haskell
data LTI where
  TF    :: String -> LTI
  (:->) :: LTI -> LTI -> LTI
  (:<>) :: LTI -> (LTI, LTI) -> LTI
  FB    :: LTI -> LTI -> LTI
```

The expression `FB a b` represents a feedback loop from `a`'s output, through `b`, and into `a`'s input. Finding the semantic value of `FB` requires some simple algebra. (In the following pseudo-code, assume u has transfer function `const 1`).

```
eval (FB a b) == eval (u :-> FB a b)

sys == (u + sys * b) * a
sys == u * a + sys * b * a
(1 - b * a) * sys == u * a
sys == a / (1 - a * b) * u

eval (FB a b) == a / (1 - a * b)
```

Thus,

```haskell
evalLTI d (FB a b) = eval d a / (1 - eval d a * eval d b)
```

Let's now try to define a closed-loop system. Consider the following system: We can define it simply as follows:

```haskell
testSys2 = TF "H" :-> FB (TF "G") (TF "S")
```

Evaluating this step by step, we have the following (where h, g, and s are the semantic functions representing the transfer functions H, G, and S, respectively, and eval is short for `evalLTI d`):

```
eval testSys2 ==
  == eval (TF "H") * eval (FB (TF "G") (TF "S"))
  == h * eval (TF "G") / (1 - eval (TF "G") * eval (TF "S"))
  == h * g / (1 - g * s)
```

If we take a look again at how we semantically defined `:<>`, we can notice how this only adds and multiplies transfer functions. In the closed-loop example above, the result had only multiplication, addition, and their inverse operators, division and subtraction. We can by this see that it's not far-fetched to assume a combined (by these operators) system's transfer function can always be written as a fraction of sums of products of transfer functions (yes, a bit of a mouthful). And, indeed, this is the case (for any transfer function)!

$$\mathrm{G} = \frac{A_1 * A_2 * ... + B_1 * B_2 * ... + ...}{A'_1 * A'_2 * ... + B'_1 * B'_2 * ... + ...}$$

5 Stability

Determining the stability in a system is important for a number of applications. How do we ascertain a system will not react uncontrollably to some input? Will a certain system come to an equilibrium or rapidly diverge?

We have a pretty good intuition for what stability is, but to give a more formal definition: a system is said to be stable if for some bounded input, also the output will remain bounded. Technically, by this definition, a system may act unstable given an impulse input ($\delta(t)$) since this signal isn't bounded, but in practise this does not pose a problem because we can't produce unbounded signals in reality.

There are several methods for determining system stability, which are all useful in different cases. These methods include Bode plots, the root locus method, the Routh-Hurwitz criterion, and the Nyquist criterion. In the next section we will study this last method in detail.

5.1 The Nyquist Criterion of Stability

The Nyquist criterion uses the location of zeroes and poles in the transfer function of a system in order to determine stability. In order to locate these poles and zeroes, "Cauchy's argument principle" is used. As this is key to understand the Nyquist criterion, we will take a closer look at what it states in a little bit.

The Nyquist criterion states that if all poles in a system's transfer function lies in the negative half of the complex number plane (i.e. real part of $z < 0$), the system is stable. Let's take a look at why this is the case.

As shown in section 4.6, a control system can always be expressed as a fraction of two transfer functions. The question about poles of a system's transfer function can therefore be rephrased as a question about zeroes in the transfer function in the denominator.

Let's first define a transfer function by its poles and zeroes in the following way:

```haskell
--                                zeros poles
data TransferFunc = TransferFunc [C]  [C]
```

$$\text{TransferFunc}([z_0, z_1, ...]; [p_0, p_1, ...]) = \frac{(s - z_0)(s - z_1)...}{(s - p_0)(s - p_1)...}$$

As the poles do not change when multiplying the function by some bounded function (we assume the fraction is fully reduced, i.e. nothing like (s-1)(s-2)/(s-1)), `TransferFunc a b` is equivalent to `TransferFunc [] b` in terms of where poles are. Recall the fact that a fraction of the form $\frac{1}{ab}$ can be written $\frac{A}{a} + \frac{B}{b}$. In the following pseudocode, let `f ~== g` denote that functions f and g have the same poles (not zeroes), and `TF` be short for `TransferFunc`.

```
TF a b ~== TF [] b
TF [] [b0,b1,...] ~== TF [] [b0] + TF [] [b1] + ... ==
  == laplace $ \t -> exp (t*b0) + exp (t*b1)+...
```

Notice how `\t -> exp (t*bx)` would be exponential decay if `bx` is negative. This is to say that when `t` is large, the signal tends to 0. Crucially, this is *only* true when bx (denoting b0, b1, etc.) is negative. If bx would be positive, then the function would be exponentially increasing instead, which would be an unstable system. (If bx would be 0, then the function would be constant 1). Since b are the poles of `TransferFunc [] b`, we see that system stability is equivalent to whether the poles of the system's transfer function lies in the negative half of the complex plane. To understand how to determine this, we should take a look at Chauchy's arguement principle.

This theorem uses a the notion of a contour to relate movement around poles and zeroes and movement around the origin, so we must first be clear about what a contour is:

5.1.1 Contours

Think of a contour as all of the values along a closed loop in the complex plane (in generality not limited to the complex plane; a contour may be on any (multidimensional) "sheet"). For example, consider the unit circle in the complex plane, centered at the origin. We might mathematically describe this contour by the set of points' unit distance from the origin: $\{z \in \mathbb{C} : |z| = 1\}$. Since $\mathbb{C}$ is not only infinite in size, but *uncountably* infinite (meaning elements cannot be listed, even with infinite terms/time), we cannot accurately represent this in Haskell—even with infinite lists. What we can do, though, is approximate it in Haskell (with finite or infinite lists). In the code below, `cos` and `sin` are used to trace out the unit circle, and `d` specifies the distance between consecutive points. When $d \to 0^+$ `ucContour` tends to *an approximation* of the unit circle contour.

```haskell
ucContour :: R -> [C]
ucContour d = [C (cos (2*pi*k//n)) (sin (2*pi*k//n)) | k<-[0..n]] where
  n = floor (2*pi/d)
  a // b = fromIntegral a / fromIntegral b
```

Let's now define a more general function for contours, that will take a list of points and then subdivide this into an approximated contour of specified accuracy. We start with defining `subdiv` as a function that subdivides one such line segment (defined by two complex numbers):

```haskell
(*&) :: R -> C -> C
c *& C r i = C (c * r) (c * i)
infixl 7 *& -- same as *

subdiv :: R -> (C, C) -> [C]
subdiv d (a, b) = [ a + n *& seg | n <- [1..nseg]] where
  len = re.abs $ b - a
  nseg = fromIntegral.floor $ len / d
  seg = (d / len) *& (b - a)
```

Now, we use this helper function to define `contourAng`. This function will subdivide a polygon described by a given list of vertices, into a more numerous list of points, to specified accuracy.

```haskell
contourAng :: R -> [C] -> [C]
contourAng dist points = concatMap (subdiv dist) $ pairs ps where
  ps = points ++ [head points]
  pairs (a:b:cs) = (a,b) : pairs (b:cs)
  pairs [_]      = []
```

We can use this function in order to create polygonal contours in the complex plane.

5.1.2 Cauchy's Argument Principle

Cauchy's argument principle states that, given a transfer function F(s) (in code denoted `f`) and a contour C (in code denoted `c`) (with counter-clockwise direction), if F has z zeroes and p poles inside C, then the contour `map f c` will travel around the origin (z-p) times in the counterclockwise direction.

To show why this is the case, consider the minimal case where we have a transfer function F(s)=(s-z0), which has a zero at z0. If we map F onto a contour C surrounding z0 in the complex plane, this is naturally equivalent to translating C with -z0. Since C is defined to surround z0, the translated contour

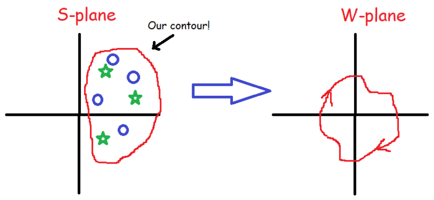


Figure 17: Mapping the values in a contour surrounding three zeroes and four poles in the s-plane (for a transfer function) is mapped to a contour surrounding the origin once. (This mapping is sometimes said to be in the "W-plane.")

will surround z0+(-z0)=0 (i.e. the origin). If we add a zero to the function (F_2(s)=(s-z0)(s-z1)) and let C surround both of these, `map f c` will be the element-wise product of the contours translated by -z0 and -z1, respectively. We can imagine how this will still encircle the origin, but, in fact, it will do so twice. This is because (as reminded in section 2.1) when multiplying two complex numbers, their arguments add—if both contours surrounds the origin once, their element-wise product should do so twice. If, instead C surrounds only z0, when mapping (s-z1) on C, this translates C to surround (z0-z1) strictly without surrounding the origin.

In the general case, let F_3(s)=(s-z0)g(s), with a zero at z0, for some function g. In this case, the contour is translated separately by -z0 as well as the roots of g(s), and multiplied in the same way as before.

All in all thus far, if C surrounds z zeroes of F, `map f c` will envelop the origin z times in the counter-clockwise direction.

The same argument could be made about the poles of a transfer function F(s). Let F(s)=Z(s)/P(s), where Z contains the zeroes of F, and P the poles of F as zeroes. The previous argument goes for Z(s), as well as the zeroes of P. When P is reciprocated, each point in `map p c` is reciprocated, and is as a consequence flipped over the real line[4]. This flips the direction of C from counter-clockwise to clockwise.

From these two arguments, we see that F mapped over a contour C which encircles z zeroes and p poles of F once in the counter-clockwise direction, will encircle the origin (z-p) times. This is what Cauchy's argument principle states.

5.1.3 Nyquist Criterion

As mentioned earlier, the Nyquist criterion states that if there are no poles in the positive side of the complex plane of a transfer function F, its corresponding system is stable. If we assume we have the function P in F(s)=Z(s)/P(s), where Z contains all zeroes of F and P all poles of F as zeroes, we can check whether P has any zeroes in the positive side of the complex plane using Cauchy's argument principle. We do this by placing a contour that surrounds all of the positive side of the complex plane. By mapping P onto this contour, we get the number of zeroes minus the number of poles in this half-plane, as the so-called winding-number (the number of times the contour travels around the origin in the counter-clockwise direction). Assuming we know P to have no poles (in the positive side of the complex plane), we can use this principle directly: if the mapped contour encircles the origin, the system is unstable; if it does not, it is stable.

Since stability is only relevant for closed-loop systems (and therefore contain a feedback loop), P will have the form P(s)= $1 - \mathrm{G}(s) = 1 - \sum_x \prod (s - \mathrm{px})$, as seen in section 4.6. Therefore, the Nyquist

[4] $1/(a+bi) = (a-bi)/((a+bi)(a-bi)) = (a-bi)/(a^2-b^2) = (a-bi)c$. When reciprocated, $z = (a+bi)$ is conjugated (flipped over real line) and scaled by $c = a^2 - b^2$

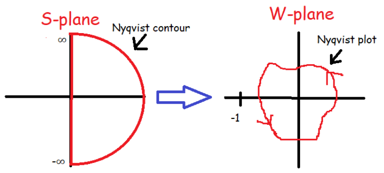


Figure 18: Placing a contour that surrounds all of the right-hand side of the complex plane, and mapping with a (transfer) function gives us the so-called Nyquist plot. If this surrounds the origin, the system the transfer function describes is stable.

criterion is often stated in terms of the transfer function G, where the mapped contour instead would wind around (-1) instead of the origin.

The criterion can be simplified somewhat by changing the contour to a non-closed contour going from `C 0 (-Infinity)` to `C 0 (Infinity)` and looking only at whether `map g c` passes above or below (-1) on its way to the origin. This only works for negative feedback-loops, however. This is often called the simplified Nyquist criterion.

A Code

```haskell
--        null-val zip-function   left    right   zipped
zipWithL :: a -> (a -> a -> b) -> [a] -> [a] -> [b]
zipWithL i f (l:left) (r:right) = f l r : zipWithL i f left right
zipWithL _ _ []       []        = []
zipWithL i f []       (r:right) = f i r : zipWithL i f []   right
zipWithL i f (l:left) []        = f l i : zipWithL i f left []
```

Code 3: Extension of `zipWith`. If lists are equal in length `zipWithL` and `zipWith` are identical. When one list has exhausted their elements, `i` is used instead. `zipWithL k f [a0,a1,a2] [b0,b1,b2,b3,b4]` becomes `[a0 `f` b0, a1 `f` b1, a2 `f` b2, i `f` b3, i `f` b4]`.

```haskell
maxfreq :: (Ord a, Num a) => Signal a -> a
maxfreq (Sin _ f _) = f
maxfreq (Scale _ a) = maxfreq a
maxfreq (Deriv   a) = maxfreq a
maxfreq (Integ   a) = maxfreq a
maxfreq (Sum a b) | af < bf   = bf
                  | otherwise = af where
                    af = maxfreq a
                    bf = maxfreq b
```

Code 4: `maxfreq` is a helper function on the `Signal` data type for the function `simplifySignal` (see code snippet 1). It returns the maximal frequency found in the given `Signal` expression tree defined by `Sum` constructors as tree nodes and `Sin` as leaves.

```haskell
{- DSL for complex numbers. Adapted with minor modifications from
   https://github.com/DSLsofMath/BScProj/ -}
module ComplexNumbers where
import GHC.Real

type R = Double

-- Komplexa tal kan ses som ett par av reella värden.

data Complex = Complex (R, R)
    deriving Eq

-- Där första komponenten representerar realdelen och den andra komponenten
-- imaginärdelen.

realPart :: Complex -> R
realPart (Complex (re, im)) = re

imPart :: Complex -> R
imPart (Complex (re, im)) = im

conjugate :: Complex -> Complex
conjugate z = Complex ((realPart z), (negate (imPart z)))

```

```haskell
{-
   Argumentet av ett komplext tal.
   Bökigt värre på grund av tråkiga kvadranter och bös,
   men genom att använda funktionen atan2 så blir det mycket snyggare!
-}

arg :: Complex -> R
arg z = atan2 (imPart z) (realPart z)

{-
   Utan atan2 hade vi fått skriva så här:

   arg :: Complex -> R
   arg z  | (imPart z) < 0  && (realPart z) < 0 = atan (imPart z / realPart z)  - pi
          | (realPart z) < 0                    = atan (imPart z / realPart z) + pi
          | otherwise                           = atan (imPart z / realPart z)

-}

scale :: R -> Complex -> Complex
scale a z = Complex ((a * realPart z), (a * imPart z))

{-
   j är det komplexa talet med realdelen 0 och imaginärdelen 1
   Många matematiktexter kallar detta talet också för `i`
-}

j :: Complex
j = Complex (0, 1)

-- TODO: Försök få   read . printComplex = id genom att göra printComplex
-- enklare
-- TODO: "name and reuse" something like showIm im = show im ++ "j" to make it
-- easy to change the spacing (or change "j" to "i", or the type from String
-- to ShowS) in only one place

instance Show Complex where
    show = printComplex

printComplex :: Complex -> String
printComplex z
  | r == 0    = imj
  | im == 0   = show r
  | im < 0    = show r ++ "-" ++ show (abs im) ++ "j"
  | otherwise = show r ++ "+" ++ imj
    where im  = imPart z
          r   = realPart z
          imj
            | im == 0   = "0"
            | otherwise = show im ++ "j"
```

```haskell

-- | Skapar ett komplext tal utifrån en vinkel.
euler :: R -> Complex
euler phi = Complex ((cos phi), (sin phi))

instance Num Complex where
-- Plus är förhållandevist trivialt
  z + w = Complex ((realPart z + realPart w), (imPart z + imPart w))

-- Multiplikationen följer ifrån att vi multiplicerar komponenterna
-- parvis med varandra (i likhet med (a+b) * (c+d))
  z * w = Complex ((realZ*realW - imZ*imW), (realZ*imW + realW*imZ))
    where realZ = realPart z
          realW = realPart w
          imZ   = imPart z
          imW   = imPart w

-- Negationen av ett komplext tal är ett komplext tal där båda komponenter är negerade

  negate z = Complex ((negate $ realPart z), (negate $ imPart z))

-- Den naturliga generaliseringen av signum från reella till komplexa tal:

  signum z = z / abs z

-- Absolutbeloppet av ett komplext tal är pythagoras sats på dess komponenter.

  abs z = Complex (hyp, 0)
    where hyp  = sqrt (r*r + im*im)
          r    = realPart z
          im   = imPart z

-- Heltal är bara komplexa tal utan imaginärdel

  fromInteger n = Complex (fromInteger n, 0)

instance Fractional Complex where
-- Division utförs genom att man förlänger hela bråket med nämnarens konjugat
  z / w = Complex (realPart zw' / realPart ww', imPart zw' / realPart ww')
    where zw' = z * (conjugate w)
          ww' = w * (conjugate w)

-- En kvot av två heltal är helt reell och därför kommer imaginärdelen vara 0
  fromRational z = Complex (re, 0)
    where re = fromInteger (numerator z) / fromInteger (denominator z)

instance Floating Complex where
-- Det komplexa talet pi är ett tal med realdelen pi och imaginärdelen 0
  pi = Complex (pi, 0)

{-
```

```haskell
    Potenslagarna ger att e^(a+bj) <=> e^a * e^bj och
    Eulers formel ger att e^bj <=> cos b + jsin b
-}
  exp z = scale (exp (realPart z)) (euler (imPart z))

{-
    Eulers formel ger att man kan skriva om sin x som (e^jx  -  e^-jx)/2j
    Eftersom vi definerat exponentialfunktionen för komplexa tal kan vi använda
    eulers formel som vår sinus implementation för komplexa tal
-}
  sin z = (exp (j*z) - exp (-(j*z)))/(scale 2 j)

-- Eulers formel ger att man kan skriva om cos x som (e^jx + e^-jx)/2
  cos z = (exp (j*z) + exp (-(j*z)))/2

  cosh z = Complex (cosh (realPart z) * cos (imPart z),
                    sinh (realPart z) * sin (imPart z))

  sinh z = Complex (sin (realPart z) * cosh (imPart z),
                    cos (realPart z) * sinh (imPart z))

{-
    Eulers formel ger att z = re^(j*phi) och eftersom log och exponentialfunktionen
    är varandras inverser och logaritmlagarna ger ex. `log (a*b)` <=> `log a + log b`.
    Därför kan vi skriva log `z = log r + log (e^(j*phi))` = `log r + j*phi`
-}
  log z = Complex (logR (abs z), arg z)
    where logR = log . realPart

{-
  Funktioner vi troligen / förhoppningsvis inte kommer behöva och som vi därför lämnar
  odefinierade än så länge.
-}
  atanh = undefined
  atan  = undefined
  asinh = undefined
  asin  = undefined
  acosh = undefined
  acos  = undefined
```

Code 5: ComplexNumbers, a DSL for complex numbers, which is used as a building block for the Laplace transform. Made by [1], with slight modifications by us to ensure it fit our DSL in 2.1.

```haskell
data Expression a where
  Const      :: a -> Expression a
  Id         :: Expression a
  Impulse    :: Expression a
  Pi         :: Expression a
  (:+:)      :: Expression a -> Expression a -> Expression a
  (:*:)      :: Expression a -> Expression a -> Expression a
  (:-:)      :: Expression a -> Expression a -> Expression a
  (:/:)      :: Expression a -> Expression a -> Expression a
  Conv       :: Expression a -> Expression a -> Expression a
  Shift      :: a -> Expression a
  Negate     :: Expression a
  Exp        :: Expression a
  Sin        :: Expression a
  Cos        :: Expression a
  Tan        :: Expression a
  Asin       :: Expression a
  Acos       :: Expression a
  Atan       :: Expression a
  Integral   :: Expression a -> a -> Expression a -- function and initial value, i.e. int_y0^t f(x)
  Derivative :: Expression a -> a -> Expression a -- function and initial value, i.e. f' and f(0)
 deriving (Eq, Show)
```

Code 6: Expression datatype.

```haskell
shift :: Num a => a -> (a -> a) -> (a -> a)
shift tau f = \t -> f (t-tau)

eval :: (Num a, Floating a, Eq a) => Expression a -> a -> a
eval (Const c)           = \t -> c
eval Id                  = id
eval Impulse             = undefined -- TODO
eval Pi                  = eval (Const pi)
eval (e1 :+: e2)         = eval e1 + eval e2
eval (e1 :*: e2)         = eval e1 * eval e2
eval (e1 :-: e2)         = eval e1 - eval e2
eval (e1 :/: e2)         = eval e1 / eval e2
eval (DefInteg e1 e2 e3) = undefined -- TODO
eval (Conv      e1 e2)   = undefined -- TODO
eval (Shift tau e)       = shift tau (eval e) -- shifts e by tau, i.e. begins after tau seconds
eval (Negate e)          = negate (eval e)
eval (Exp e)             = exp    (eval e)
eval (Sin e)             = sin    (eval e)
eval (Cos e)             = cos    (eval e)
eval (Tan e)             = tan    (eval e)
eval (Asin e)            = asin   (eval e)
eval (Acos e)            = acos   (eval e)
eval (Atan e)            = atan   (eval e)
eval (Integral   e e0)   = undefined -- TODO
eval (Derivative e e0)   = undefined -- TODO
```

Code 7: Function eval. Takes a syntactic representation and turns it into a semantic function.

```haskell
-- rules
laplace :: Expression Complex -> Expression Complex -- TODO: maybe instead do Time->Freq
laplace (e1 :+: e2) = laplace e1 :+: laplace e2 -- Superposition
laplace (e1 :-: e2) = laplace e1 :+: Negate (laplace e2) -- special case of superposition
laplace (Const a :*: e) = Const a :*: laplace e -- scaling
laplace (Negate e) = Negate (laplace e) -- can be seen as scaling with -1 as well
laplace (Derivative e e0) = Id :*: laplace e :-: (Const e0) -- L {f'} = \s -> s * L {f} - f(0)
laplace (Integral e e0) = laplace e :/: Id -- L {int_0^t f(x) dx} = s -> L {f} /s
laplace (Shift tau e) = Exp (Id :*: Const tau) :*: laplace e -- TODO: double check signs
  -- TODO: double check signs on next two
laplace (Exp (Const tau :*: Id) :*: e) = Shift (negate tau) (laplace e) -- semantically same as next
laplace (Exp (Negate (Const tau :*: Id)) :*: e) = Shift tau (laplace e)
laplace (Conv e1 e2) = laplace e1 :*: laplace e2 -- convolution turns into multiplication
-- transforms
laplace (Impulse) = Const 1 -- L {delta(t)} = \s -> 1
laplace (Const a) = Const a :/: Id -- L {a} = \s -> a/s; a = 1 => L {1} = 1/s
laplace (Id)      = Const 1 :/: (Id :*: Id) -- L {t} = \s -> 1/(s^2)
laplace (Exp (Negate (Const a :*: Id))) = Shift (negate a) (Const 1 :/: Id)-- TODO: Check signs
laplace (Exp (Const a :*: Id)) = Shift a (Const 1 :/: Id)-- TODO: Check signs
laplace (Const 1 :-: Exp (Negate (Const a :*: Id))) = Const a :/: (Id :*: Shift a Id) -- TODO: Doubl

laplace (Exp Id) = laplace (Exp (Const 1 :*: Id))
-- ... some more special cases, not sure we care?
-- TODO: Doublecheck signs on the next two
laplace (Exp (Negate (Const a :*: Id)) :*: Sin (Const omega :*: Id)) =
  Const omega :/: (Shift a (Id :*: Id) :+: (Const (omega*omega)))
laplace (Exp (Negate (Const a :*: Id)) :*: Cos (Const omega :*: Id)) =
  (Shift a Id) :/: (Shift a (Id :*: Id) :+: (Const (omega*omega)))
```
Code 8: Definition of `laplace`, the Laplace transform of an `Expression`.