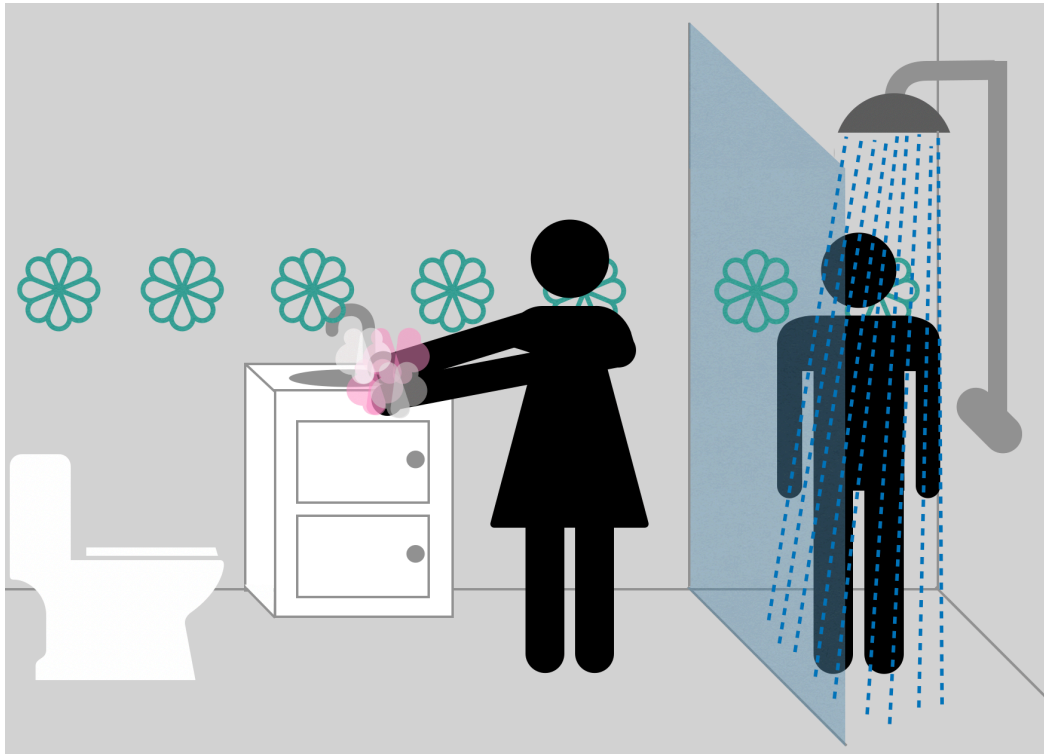




CHALMERS
UNIVERSITY OF TECHNOLOGY



A Computer Science Approach to Teaching Control Theory

Developing Learning Material Using
Domain-Specific Languages

Bachelor of Science Thesis in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2020
www.chalmers.se

A Computer Science Approach to Teaching Control Theory

Developing Learning Material using
Domain-Specific Languages

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

Spring - 2020

A Computer Science Approach to Teaching Control Theory
Developing Learning Material using Domain-Specific Languages
Jakob Fihlman
Simon Hägglund
Christian Josefson
Filip Nylander
Elin Ohlman
Tommy Räjert

© Jakob Fihlman, 2020.
© Simon Hägglund, 2020.
© Christian Josefson, 2020.
© Filip Nylander, 2020.
© Elin Ohlman, 2020.
© Tommy Räjert, 2020.

Supervisor: Patrik Jansson, Department of Computer Science and Engineering
Examiner: David Sands, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: The cover picture of the learning material developed in this project. Temperature regulation is a very simple control system which is exemplified in this picture by the man showering and the woman washing her hands. The learning material can be found in [appendix A](#). Illustrator: Elin Ohlman.

Typeset in L^AT_EX
Gothenburg, Sweden 2020

Abstract

Different disciplines use varying approaches to teaching, which may lead to difficulties when learning subjects from other disciplines. Previous research has proposed using teaching techniques from computer science in order to improve the learning for computer science students in mathematical domains. Some of these techniques include the implementation and usage of domain-specific languages, the study of types, and using an explicit distinction between syntax and semantics. For example, the project “Domain-Specific Languages of Mathematics” (“DSLsofMath”) has applied these techniques, resulting in both a course at Chalmers University of Technology and accompanying course material. In the course, domain-specific languages are implemented for mathematical concepts, and mathematics is analysed from a computer science perspective. Passing the course has shown positive correlation with passing later courses, one of which is Control Theory, that many students struggle with.

This report presents the learning material “Domain-specific languages of Control Theory – A supplementary learning material for ERE103”. It is specifically targeted towards the Chalmers control theory course ERE103 and takes inspiration from the DSLsofMath project. Course surveys and communication with the examiner of the course were used to decide which areas within control theory would be included in the learning material. The selection is intended to capture the areas that students have most trouble with and are most suitably represented with domain-specific languages. The material describes the implementation of several domain-specific languages for control theory using the programming language Haskell as well as analysing the included mathematics from a computer science perspective. The learning material covers some prerequisites for the course, such as integrals, complex numbers, and the Laplace transform. It also contains some central parts of control theory such as LTI systems, transfer functions, and the Nyquist criterion. Some didactic theory is studied and used to motivate the methods and design choices used in the learning material.

Some possible future derivative projects include continuing to develop the learning material and the languages contained therein, empirically evaluating the presented learning material, or developing a similar material for another domain.

Keywords: Domain-specific language, DSL, learning material, control theory, Haskell

Sammandrag

Olika discipliner använder olika tillvägagångssätt för att undervisa, vilket kan skapa svårigheter för de studenter som rör sig mellan dessa discipliner. Tidigare forskning har föreslagit att lärtekniker från datavetenskap skall appliceras för att förbättra lärandet hos datavetenskapsstudenter inom matematiska ämnen. Bland annat inkluderar dessa lärotekniker domänspecifika språk, typteori, och tydlig uppdelning mellan syntax och semantik. Projektet "Domain-Specific Languages of Mathematics" ("DSLsofMath") har använt dessa tekniker och resulterat både i en kurs på Chalmers universitet och tillhörande kursmaterial. I kursen implementeras domänspecifika språk för matematiska koncept, och matematiken analyseras från ett datavetenskapligt perspektiv. Studenter som avslutat kursen har påvisat ökade chanser att även klara två senare kurser som underkänt många av deras klasskamrater.

Denna rapport presenterar det supplementära läromaterialet "Domain-specific languages of Control Theory - A supplementary learning material for ERE103". Det är speciellt utformat för Chalmers' Reglerteknikkurs ERE103, vilken är en av tidigare nämnda kurser som underkänt många, och tar inspiration från DSLsofMath. För att avgöra vad som skulle ingå i läromaterialet användes återkoppling från både kursansvarig och elever. Tanken var att fånga de avsnitt som elever haft svårast att greppa, samt de avsnitt som effektivast kan förklaras via domänspecifika språk. Läromaterialet beskriver implementationen av ett flertal domänspecifika språk för reglerteknik med hjälp av programmeringsspråket Haskell, och diskuterar även hur dessa fungerar. Läromaterialet är uppdelat mellan avsnitt som behandlar förkunskaper till kursen; integraler, komplexa tal och Laplace transformen, samt avsnitt som hanterar mer centrala koncept inom reglerteknik, såsom LTI-system, överföringsfunktioner och Nyquistkriteriet. Det används även en del didaktisk teori för att utöka effekten av läromaterialet.

Framtida projekt skulle kunna baseras på att fortsätta utöka läromaterialet och de domänspecifika språken som skapats. De skulle även kunna evaluera projektet empiriskt, eller utveckla ytterligare läromaterial för andra ämnen.

Nyckelord: Domänspecifika språk, DSL, läromaterial, reglerteknik, Haskell

Preface

This report is the result of the project “Domain-Specific Languages of Mathematics for other courses” at Chalmers University of Technology and University of Gothenburg. The idea was to explore how the teaching of control theory could benefit from a similar approach as “Domain-Specific Languages of Mathematics”, a project originating at Chalmers University of Technology. We are an interdisciplinary group of students from the programmes “Computer Science and Engineering”, “Engineering Physics”, “Engineering Mathematics” and “Information Technology” at Chalmers University of Technology and “Computer Science” at University of Gothenburg.

We want to thank Patrik Jansson, our supervisor, for his guidance and patience during these trying times.

Table of Contents

Glossary	V
1 Introduction	1
2 Theory	3
2.1 Control Theory Course ERE103	3
2.2 Domain-Specific Languages	3
2.2.1 Embedded Domain-Specific Languages	4
2.3 Control Theory	5
2.4 Didactics	6
2.4.1 Teacher provided support	6
2.4.2 Constructivism	7
3 Process	9
3.1 Product Specification	9
3.2 Delimitations	9
3.3 Implementation	10
4 Learning material	12
4.1 Prerequisites	12
4.1.1 Complex numbers	12
4.1.2 Integrals	13
4.2 Laplace Transform	13
4.3 Transfer Functions	13
4.4 The Nyquist Stability Criterion	15
5 Discussion	16
5.1 Evaluation of the Learning Material	16
5.1.1 Prerequisites	17
5.1.2 Core Course Subjects	18
5.1.3 Exercises	18
5.2 The Use of Haskell as a Host Language	18
5.3 Medium of Presentation	19
5.4 Future Projects	20
5.5 Ethics and Utility	20
6 Conclusions	22
A Learning material	26

Glossary

Control theory An area of study which deals with the control of continuously operating dynamical systems in engineered processes and machines. [1–3](#), [5](#), [6](#), [8](#), [9](#), [16](#), [18–22](#)

DAT326 Course code for “Domain-Specific Languages of Mathematics”. [1](#)

Didactics The science of teaching. [6](#), [9](#), [21](#), [22](#)

Domain-Specific Language A programming language created and tailored for a specific narrow domain. [1](#)

Domain-Specific Visual Language A DSL that is not programmatic, but visual. They are often based on different types of diagrams. [3](#)

DSL See Domain-Specific Language. [1–5](#), [8–14](#), [16–20](#), [22](#)

DSLsofMath Short for “Domain-specific languages of Mathematics,” a project and related BSc-level course at Chalmers. [1](#), [2](#), [10](#), [16](#)

DSVL See Domain-Specific Visual Language. [3](#), [4](#), [14](#), [19](#)

EDSL Embedded domain-specific language, a DSL implemented within another, general purpose, language. [4](#), [16](#)

ERE103 Course code for the course “Control theory” at Chalmers. [2](#), [3](#), [7](#), [9](#), [18](#), [21](#), [22](#)

GADTs See Generalised Algebraic Datatypes. [4](#), [5](#), [9–11](#)

Generalised Algebraic Datatypes A non-standard extension of Haskell, enabling more advanced type behaviour. [5](#)

Haskell A functional programming language. [1](#), [4](#), [5](#), [8](#), [9](#), [11](#), [13](#), [16–19](#), [22](#)

Linear Time-Invariant Systems where the output does not depend on when the input was applied and where the output is linearly related to the input. [V](#)

LTI See Linear Time-Invariant. [11](#), [13](#), [14](#), [19](#)

SSY080 Course code for “Transforms, Signals and Systems”. [2](#), [10](#)

TIF085 Course code for “Physics for Engineers”. [2](#)

ZPD Zone of Proximal Development, the collection of tasks the student can do only with guidance. [6](#), [16](#)

1 Introduction

Learning a new and difficult subject can be arduous. When knowledge can be connected to already familiar knowledge, learning can be easier than when knowledge is entirely new. In this text we will focus on the difficulties for computer science students when learning control theory and possible solutions to the problem.

It has been argued that the computer science perspective could be beneficial for mathematics education. In “Communicating mathematics: Useful ideas from computer science” [1], Wells have many ideas on how to improve mathematical education, but two ideas in particular are worth mentioning: writing out the types of introduced objects and being explicit about the distinction between syntax and semantics. As an example of the distinction between syntax and semantics, Wells discusses the difference between $2/7$ and $2/(4+3)$. Semantically they are identical, but syntactically they are different. In short, Wells argues that, typically, there are differences between how teachers and students view these mathematical objects. Wells argues that confusion arising from that distinction might be avoided if the students have words for it.

There have been attempts to put in practice the idea of teaching mathematics and related domains using computer science tools. Some examples include “Functional Differential Geometry” [2] where differential geometry is implemented in the programming language Scheme, “A Logical Guide to Discrete Math” [3] which is written from the perspective of two programming methodologists, “Learn Quantum Mechanics with Haskell” [4] which describes how Haskell can be used to model quantum mechanics, and finally “Learn Physics with Haskell” [5] which describes a functional programming approach to teaching mechanics and electromagnetic theory in a course at Lebanon Valley College.

One attempt has been a special inspiration for this project: “Domain-specific languages of Mathematics” (henceforth “DSLsofMath”) which is a project aiming to both analyse mathematics from a computer science perspective and to teach how to construct Domain-Specific Languages in Haskell [6]. A domain-specific language (often shortened DSL) is “a computer programming language of limited expressiveness focused on a particular domain” [7, p. 27]. There is a multitude of domains one can implement DSLs for, and DSLsofMath aims to show how different areas of mathematics such as group theory and logic can be seen as such domains [6]. DSLsofMath has resulted in a course at Chalmers University of Technology, “Domain-specific Languages of Mathematics” (“Matematikens Domänspecifika språk” DAT326 [8]). Passing the course has shown positive correlation with passing later courses that many students struggle with [9].

Control theory is a subject which might benefit from being taught using DSLs. Control theory is, in short, the study of systems and their control. A rudimentary example of a system is a shower, where the temperature of the water is controlled by turning the shower knob. This example involves human input, but in many cases we would prefer to not require human input. Control theory is used to automate the control of systems using different mathematical analyses. Thus previously mentioned problems with mathematics might hinder a student from learning control theory. One way to solve this could be to use an approach similar to DSLsofMath, using DSLs to teach control theory. This project will attempt to implement this

approach.

As a concrete example of the difficulties in teaching control theory: at Chalmers, third-year students at the “Computer Science and Engineering” programme take an introductory course in control theory. The course, “Control Theory” (“Reglerteknik” ERE103 [10]) is a course students typically struggle with [9]. As it is a mandatory course it can represent a significant stumbling block which makes it a good candidate for our project.

The aim of this project has been to create a supplementary learning material for control theory following the approach used in DSLsofMath. The learning material describes the different domain-specific languages for control theory and uses them to explain it. Some of the tools required to successfully study control theory such as complex numbers, the Laplace transform, and transfer functions are also described using their own DSLs. Which parts to include in the learning material were identified in three ways: asking the examining teacher in the course, utilising team members experience with the subject and reading the course evaluation minutes.

Other projects building on DSLsofMath have been carried out. One project [11] explores the uses of DSLs to teach parts of the course “Transforms, Signals and Systems” (“Transformer, Signaler och System” SSY080 [12]). Some of the content taught acts as prerequisites for ERE103. The other [13] explores DSLs for physics—more specifically classical mechanics—and resulted in the supplementary learning material “Learn You A Physics For Great Good!” [14] for the course “Physics for engineers” (“Fysik för ingenjörer” TIF085 [15]). Since the first of these handles some of the tools required for control theory, there will be some overlap between the material described therein and our learning material. See section 4 for more information on said overlap.

2 Theory

This section provides theoretical background about DSLs, control theory and its Chalmers course ERE103, as well as some didactic theory which is used in order to improve the effectiveness of teaching in the learning material.

2.1 Control Theory Course ERE103

The Chalmers course “Reglerteknik” (“Control Theory” ERE103 [10]) had a failure rate of approximately 50 % in 2019 [16]. Not only is this an unreasonably high number in its own right, the course is also mandatory for students at the CSE¹ programme. Thus, having a high rate of failure makes the control theory course a roadblock for many students.

In an interview [11], the examiner of ERE103 speculated that the main problem that students have is based on not having enough experience in mathematical thinking. As an example, they may have trouble understanding the connection between abstract mathematics and the real-world thing being modelled. This may cause students to spend a majority of their time and energy on understanding the mathematics and not enough on understanding the material taught in the course. For a more concrete example there are parts of control theory that utilise the Laplace transform to solve differential equations, yet it is possible that students spend more time trying to understand what the Laplace transform does and less on understanding what the differential equations mean and how to use the results.

According to the course evaluation minutes of the 2019 instance of ERE103 [16], they have not been taught the prerequisites of the course to a satisfactory level. This is in spite of the course beginning with a short review of said prerequisites. For example, the previously mentioned problems with using the Laplace transform to solve differential equations. The syllabus [10] of the course explains the necessary prerequisites thusly: “Basic concepts of mathematics that must be mastered before the start of the course are: Complex numbers, Linear algebra, Taylor series, Ordinary differential equations, Laplace transform.”

2.2 Domain-Specific Languages

A domain-specific language (DSL) is—as opposed to a general purpose language—a language that is tailored to a specific problem domain. It aims to provide an “ultimate abstraction” [17, pp. 196–es] that helps improve communication [7], and often yields better solutions to problems in their domain [18]. Generality and domain expressiveness are largely opposing powers, where DSLs favour expressiveness [19]. DSLs are often considered a subset of programming languages, but this notion is not accepted by everyone [7], [18], [20]. Arguably, any language that describes a domain with more specificity than a general purpose language might be considered a domain-specific language [20]. Commonly known DSLs include L^AT_EX, SQL, and HTML [18].

In addition to these programmatic DSLs, one could argue there are other kinds of DSLs using other mediums, such as Domain-Specific Visual Languages (DSVL).

¹“Computer Science and Engineering”, “Datateknik” in Swedish

DSLs are often different types of diagrams and may also have an accompanying programmatic DSL [20].

In this report the term DSL refers to the more generally accepted notion of a DSL as being a programming language or other executable specification language, except for when explicitly referring to DSLs or other kinds of DSLs.

2.2.1 Embedded Domain-Specific Languages

Implementing DSLs from scratch takes a lot of time and effort, and it is therefore common to implement them within another (general-purpose) language [17], called a host language [21]. This kind of DSL is referred to as an embedded DSL (EDSL). These are relatively easy to both implement and extend as they use already established languages. A secondary advantage of this is that the host language can provide general-purpose capabilities to the DSL, if need be [18]. Functional languages such as Haskell are popular as host languages for their capabilities of defining custom operators, using higher-order types, as well as overloading [21].

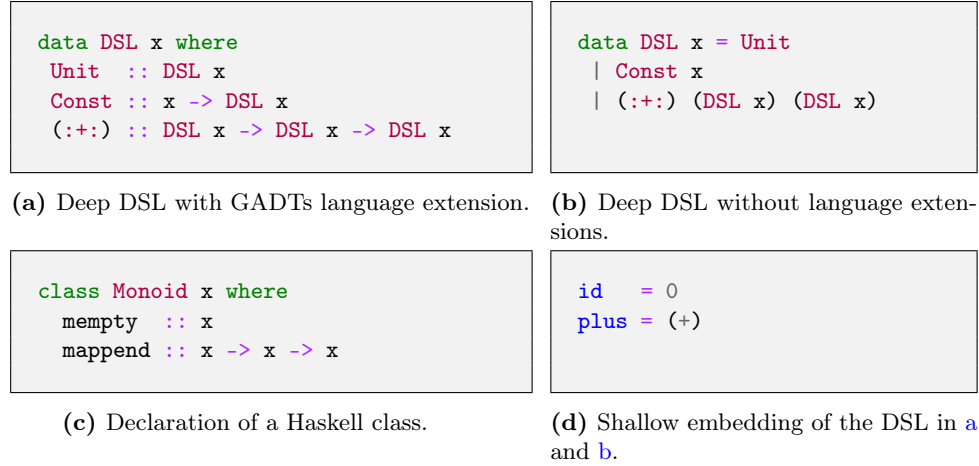


Figure 1: Three different implementations of a small DSL in Haskell are shown in figures a, b, and d. Two of these, a and b, are deep DSLs, while the third, d, is a shallow DSL. Figure c shows the likeness in declaration of classes and GADTs data types.

In figure 1a, an example of a small EDSL is presented. When the syntax of the DSL is encoded into the algebraic data type as constructors, the DSL is called a deeply embedded DSL, as opposed to a shallow embedding as is shown in figure 1d. Shallow embeddings do not have the same clear distinction between syntax and semantics. The usage of this kind of DSL will always result in instant semantic evaluation, since the syntax and semantics are embedded in the same structure. In contrast, a deep embedding—where the constructors encode the structure (syntax) and the semantic evaluation separately—will determine the result of that structure at some later point in time, and may instead manipulate the syntactic structure by itself.

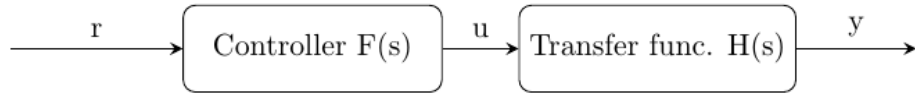
There are many language features of Haskell that simplify the implementation of embedded DSLs. The most noteworthy extension for this project is Generalised Algebraic Datatypes (GADTs) which adds Haskell syntax that facilitates declaring algebraic data types much like one would declare a Haskell instance or class.

In the code in figure 1a a small DSL over addition is implemented, where GADTs allows class-like syntax like in 1c when declaring a new algebraic data type. Figure 1b shows a version of figure 1a without the GADTs extension. These three snippets precisely describe the same behaviour in Haskell, but GADTs offers a type declaration syntax closer to that of type theory—making these types more easily readable.

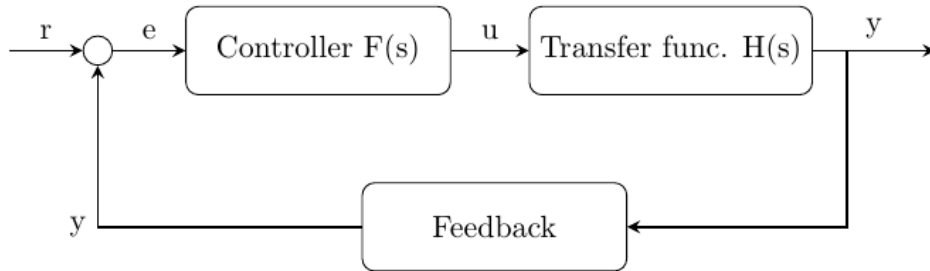
2.3 Control Theory

As a subject, control theory is relatively new, and became more relevant during the space race in the 60s [22]. However, the techniques therein have been used for much longer. The Romans used control theory to control the water levels in their aqueducts and it was later used to control the velocity of windmills [23]. Today, control theory is utilised in advanced technical systems, such as cell phone networks and fighter jets.

Control theory deals with dynamic continuous systems and their control. A system can be controlled with one of two basic loops. A closed-loop, or feedback-loop, utilises the output of the process to control the system. For an example see figure 2b. An open-loop works without the use of feedback, see figure 2a.



(a) Block diagram of a system with open loop, without feedback.



(b) Block diagram of system with closed loop, using feedback to control system.

Figure 2: Block diagrams of the general forms of two different loops often used in control theory, the open loop (which can be seen in a) and the closed loop (which can be seen in b).

These systems are all around us, and the systems where control theory is applied

often have disturbances, i.e. signals that for some unintended reason affects the system. If the system was unaffected by these disturbances, it would not need any feedback signals and could thus be modelled using an open loop, see figure 2a. The main objective of control theory is to find suitable controllers for signals (with or without feedback) given some condition —like the r signal in figure 2b. The purpose of this is to accomplish stable systems, without delay or overshoot. In the course [10] a combination of proportional (P), integral (I) and derivative (D) controllers are used to achieve this goal, resulting in what is commonly referred to as PID-controllers. When analysing systems, they are often transformed from the time domain to the frequency domain using the Laplace transform. This is done since the models of the controllers are typically easier to calculate and manipulate when working in the frequency domain.

2.4 Didactics

During their journeys towards degrees, students will likely meet as many didactic methodologies as they take courses. These may or may not be explicitly built on didactic theory, but they can likely be described from a constructivistic viewpoint.

In order to motivate the didactic design choices in the learning material, this section aims to provide some didactic background. Didactics is a large area of study, trying to cover even a significant part would require a Herculean effort. However, a few central ideas for this project will be presented. With this in mind it is important to realise that the didactic theory has been chosen to fit the project. There are other theories that might have radically different views of learning than what is presented here. As this project is not intended to be any sort of overview of didactic theory, they will not be covered.

2.4.1 Teacher provided support

Common for all well executed courses and other learning environments is that they have predefined instrumental scaffolding. Scaffolding can be thought of as the means by which a student leverages their own ability to understand, until such support is no longer required and the learner can achieve results independently.

There are two types of instrumental scaffolding, social and cognitive, which act as supporting structure for learners. These structures are created by the teacher, and can be considered temporary. They are meant to aid students reach a deeper understanding than they could have done by themselves [24]. Depending on the subject, instrumental scaffolding is often implemented as a mix of both social scaffolding and cognitive scaffolding. Social scaffolding is the scaffolding that is created when a group works together to solve a problem. Cognitive scaffolding is well described as a series of related tasks that incrementally improves the learners knowledge [24].

Instrumental scaffolding is an application of the Zone of Proximal Development (ZPD), which aims to describe the learning settings where the student learns most optimally [25]. When a learner tries to understand a new subject such as a course—according to ZPDs creator Vygotski—there is only so much a learner can achieve alone without support. ZPD is a learning tool which describes learning as different regions, where ZPD is the region between the current ability of the learner and

what the learner can achieve given the right scaffolding and with help of their peers or mentors enabling them to reaching beyond their own ability [25]. The zone that would describe what the learner currently can achieve consist only of what the student currently understands, i.e. the student's prior knowledge from other subjects.

2.4.2 Constructivism

Constructivism is a school of thought in didactics which emphasises that learners construct knowledge themselves [26, s. 48].

According to Talis, a recurring study done by the organisation OECD, among other [27], teachers working in Sweden adhere more to constructivism than those from other participating countries. Teachers in Sweden differ in that they to a higher degree view their role as a mentor, as someone who facilitates the learning environment rather than a someone that transfers their knowledge [27]. According to constructivism, this is the role of any teacher. Teachers in Sweden rarely check their students' exercise books, and would rather try to find a real world example to get their points across [27], which also fits well into constructivism. O'Loughlin have quite strong explanatory wording as to why teachers accept this theory:

The power of the image lies in the contrast between the passive, powerless learner in the traditional approach, and this image of an active, constructive knower, empowered to take charge of his or her own learning. [...] [C]onstructivism makes a strong appeal to our commonsense understanding of how learning ought to be. [28, p. 792].

It's important to regard the mindset of the teachers in Sweden when producing a learning material for the product of their work, the students of ERE103.

Constructivism began branching off from psychology in the 1860s, but solidified early in the last century. It's a large field of study. One of its prominent figureheads, Piaget, coined the terms assimilation and accommodation. The expressions classify two different experiences any student has while learning a new subject.

Assimilation refers to the experience a learner has when realising that a concept that they are trying to learn follows the same pattern as a concept they have already understood [26, p. 283]. This results in a framing of new knowledge within a preexisting concept. For example if a learner is trying to learn multiplication while already understanding addition, the student might realise that multiplication is the same as adding multiple times. This experience is called assimilation.

Accommodation on the other hand refers to when a learner realises that new knowledge does not fit within any preconceived pattern or concept [26, p. 284]. They must then instead construct a new concept in which the new knowledge fits. Take for example operator precedence: it is infeasible to understand the precedence of operators from addition and multiplication, since they are based on different conventions. In this case when the student has the realisation of operator precedence, and this experience is what is called accommodation.

Hadjerrouit has researched constructivism in software engineering and developed a framework of questions to ask when designing education [29]. In the context of this project two of these are relevant to ask:

1. Which knowledge needs to be connected?
2. Which skills are needed to connect knowledge?

In this project the answer to the first question is that the concepts in control theory needs to connect with the students existing knowledge in Haskell and programming. It aids the students in building new knowledge by relating it to the same concepts in Haskell, in constructivistic terms assimilating control theory concepts by leveraging their existing understanding of Haskell concepts.

The answer to the second question is that a necessary skill in control theory is being able to distinguish between the notation and the meaning the notation carries. In Haskell this same difference is the difference between syntax and semantics. Understanding this difference can be achieved by creating syntactic representations in Haskell which maps one to one with notation in control theory, while keeping the semantics of control theory. Thus they can accommodate the semantics, and when further studying the literature on control theory they can further assimilate the syntax of the field. More concretely, knowledge is constructed by using the learners prior knowledge as a foundation by showing the student how to construct a DSL, which will hopefully aid the students to construct new knowledge.

In conclusion, constructivism dictates that knowledge needs to be related and integrated into knowledge structures that can be retrieved at any time in order to be useful for problem solving.

3 Process

The learning material is developed based on a product specification created to fit the aims of the project. The theory presented in section 2 is used to construct the learning material in an appropriate form.

3.1 Product Specification

The learning material should have certain properties. Its purpose is to work as a learning material for students of the course ERE103. The learning material should not act as a replacement of the course book but instead be a supplementary material that offers a different approach to learning control theory and provide a perspective more familiar to computer science students, in order to ease the transition between the subjects.

The learning material should accomplish this by constructing DSLs for different areas of control theory. Some, but not all, code should be presented and explained in the text. The programming language used should be Haskell. GADTs should be used. The text should not use very formal language but instead be light and use pictures to be more easily digestible for the reader. The text should have examples and exercises.

As it is a large and difficult course as established in section 2.1 it should not be expected that this learning material will be able to cover the entirety of the material covered in the course, or even the majority of it. As such the material should focus on the initial areas of the course, including some prerequisite knowledge.

3.2 Delimitations

Control Theory is a large subject. As such, trying to create a comprehensive guide to the entire subject would risk limiting the quality of each individual concept. Therefore, an early part of the project was to analyse which concepts of control theory to focus the team's efforts on.

Using feedback from staff, records of prior exams, as well as the team's own experiences of the subject, subject areas were established and analysed. The analysis included, in a rough order of importance, which subjects students frequently struggled with, which concepts could be effectively described using DSLs, and which other didactic techniques could be employed.

Ultimately, five subject areas were chosen for further work:

- Prerequisites
- Basic Control Theory
- Laplace Transform
- Transfer Functions
- Nyquist criterion of stability

Prerequisites and Basic Control Theory are both rundowns of multiple smaller subjects that should provide little issue but are nonetheless important to bring up.

Laplace, Transfer Functions and Nyquist are larger and more difficult subjects that were deemed worthy of their own dedicated sections.

There was also a plan to include a section covering Bode diagrams. However, due to time constraints, it had to be cut.

As previously discussed in section 1, there is overlap between the content covered in the learning material and previous projects. One such project, henceforth “TSS with DSLs,” used a similar approach to teach the course “Transforms, Signals and Systems” (“Transformer, Signaler och System” SSY080 [12]) and was an important inspiration for this project. “TSS with DSLs” created DSLs for, among others, complex numbers, signals and LTI systems. As these subjects were to be included in the learning material, the choice whether to use the already developed DSLs or to develop new ones had to be taken. New DSLs were developed for all these domains, and the DSL for complex numbers was used as a dependency for the DSL for the Laplace transform. Similarly, DSLsofMath develops a DSL for complex numbers, which is not used in the learning material. See section 5 for more information on the choice to make new DSLs instead of using the previously developed ones.

3.3 Implementation

The clearest example of how some of the ideas presented in section 2 and section 2.4 were used to create the learning material is the usage of GADTs. In figure 3b there is an example from the learning material of how the GADTs help emphasize the types. The figure contains two (abbreviated) versions of the `Expression` datatype from the learning material. For the sake of space and ease of reading, `Expression` has been shorted into `Expr` and some lines have been removed. In 3a we see that the datatype has been declared by listing the different constructors for `Expr`. However, the types might not be apparent. Interpreting them as different patterns on how to create `Exprs` is one way to understand them. For example, `Const x` is of type `Expr x`. Another example is that given two expressions of type `Expr x`, adding them by putting `++` between them creates a new expression of type `Expr x`. The types should be stated explicitly, however. Implementing the datatype using GADTs allows for explicit typing, see 3b. For example, we can see that `Const` takes something of type `x` and returns something of type `Expr x`, just like interpreted above.

```

data Expr x = Id      -- The identity function
| Const x           -- A constant expression
| Negate (Expr x)    -- Negation of an expression
| Shift x (Expr x)   -- Time shifted expression
| Expr x :+: Expr x -- A sum of expressions

```

(a) Definition of `Expr` data type without using GADTs. This is done by listing different constructors for the datatype with their parameters filled in, e.g. `Const a` and `Expr x :+: Expr x`, which construct expressions representing the constant function and a sum of two expressions, respectively. Note that the types of the different constructors are not explicit, possibly requiring interpretation to understand.

```

data Expr x where
  Id      :: Expr x           -- The identity expression
  Const   :: x      -> Expr x -- A constant expression
  Negate   :: Expr x -> Expr x -- Negation of an expression
  Shift    :: x      -> Expr x -> Expr x -- Time shifted expression
  (:+:)    :: Expr x -> Expr x -> Expr x -- A sum of expressions

```

(b) Definition of `Expr` data type using GADTs. This is done by listing the different constructors (without parameters) and their respective types, e.g. `Const` and `(:+:)` are two functions of types `a -> Expr x` and `Expr x -> Expr x -> Expr x`, respectively. In this case, the types are explicit and thus likely require less interpretation.

Figure 3: Comparison of `Expr` datatype with and without GADTs.

Another main idea that the learning material is built on is the development of DSLs as a learning tool. For example, a DSL to describe Linear Time-Invariant (LTI) Systems is created, consisting of a datatype and functions between them. Explicitly building the language step-by-step is a way to help the student build an understanding of the DSL. The DSLs were developed by identifying the function declarations. This is done by starting with a very basic case of the DSL like `Id` and then continuing building on it and making sure it all works together.

In order to help learning, the learning material is built using constructivistic ideas. The idea of assimilation is especially used when we connect the students previous knowledge of Haskell through aforementioned building of DSLs. As this is a supplementary learning material we believe it will often function in an didactically accommodating way. For example a student may learn about the shift function in the main course but not fully understand it, but when they read about it again from another perspective in our learning material they have the opportunity to relate the shift function to other prior knowledge and understand it.

4 Learning material

This section provides an overview of the learning material, along with explanations for the decisions made while writing the text. This is primarily meant to elucidate the overall layout and direction of the text. For a more in-depth look, it is recommended to read the learning material itself.

The complete text as of 2020-05-14 has been included in appendix A. Please note that the text is still in development as of the writing of this report, and an up-to-date version is available at [GitHub](#)².

4.1 Prerequisites

The prerequisites section in the learning material contains a repeat of some key course prerequisites, and serves two purposes. First, to rehearse both these prerequisites and Haskell itself, in order to flatten the initial learning curve to some degree.

Secondly, to introduce and familiarise the reader with the approach of teaching with DSLs and explicit typing, and doing so with subjects that should already be familiar to the reader in order to make this transition more natural.

4.1.1 Complex numbers

The complex numbers subsection covers the core concept of the subject, along with a few basic operators. Its associated DSL is based around representing complex numbers as coordinates, thus giving the students a useful way to remind them how real and imaginary numbers interact with one another. Especially when dealing with arguments and absolute values, as understanding these is vital in the sections covering Laplace transforms and Nyquist diagrams. Finally, it also introduces the limitations of DSLs, and the value of adapting our language to engage problems from different perspectives.

Here is an example from the learning material where we see how code and explanation work together:

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.

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

²<https://github.com/simonhagglund/DATX02-dsl>

4.1.2 Integrals

The integral subsection introduces the purpose of integral operations along with two different solutions. The DSL is built to mimic polynomial algebra and introduce students to the idea of DSLs with compositional word structure. There are different ways of calculating integrals. We present two different ways: first we show how to numerically calculate integrals, and second we show how to do it analytically using primitive functions.

4.2 Laplace Transform

In this section of the learning material the Laplace transform is introduced to the reader. The definition of the Laplace transform and type is described in the beginning and later the table of common Laplace transforms is introduced. The section shows the reader how the table is used to solve problems and why this approach is preferred. Below is an example from the introduction of the section.

The Laplace transform is a tool that enables us to look at a function or equation from a different perspective. More specifically, it takes a function in time domain and transforms it into a function in frequency domain.

This sections also explains some useful rules that apply to the transform. Superposition (linearity), the derivative rule and the integral rule are all explained in detail. Convolution is also introduced in this part of the material and how convolution is Laplace transformed. Furthermore time shifting and frequency shifting are also a part of this section. At the end of this section the inverse Laplace transform is explained.

4.3 Transfer Functions

In this section transfer functions and linear time-invariant systems (LTI systems) are introduced, as well as the connection to the Laplace transform. The first part of this section explains LTI systems and their properties. This part shows the reader how to manipulate these systems and express them with Haskell. Many of the concepts that were introduced in the Laplace transform section are used to analyse LTI systems. Later in the section, the type of LTI systems is discussed with some examples, and the connection to transfer functions is established. The section also explains how transfer functions are used in different ways and how this relates to their type.

The second part of this section explores how LTI systems preserve sinusoidal input signals' shape. In this part the reader can see, step by step, how an LTI system will process the sine function. Below, a deeply embedded DSL for LTI systems on sinusoidal input taken from this section is shown.

```

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

```

```

evalSignal :: (Num x, Floating x, Eq x, Ord x) => Signal x -> (x -> x)
evalSignal (Sin  amp frq ts) t = amp * sin (frq * t + ts)
evalSignal (Scale amp s)      t = amp * evalSignal s t
evalSignal (Sum    s1 s2)     t = evalSignal s1 t + evalSignal s2 t

```

Later in this part convolution is used to combine LTI systems. It is also shown how a transfer function can be derived from differential equation of the system input and output, using the Laplace transform.

Lastly, how entire LTI systems are combined is explored in order to connect this to the algebraic manipulation of variables representing systems, which later is used in the section on the Nyquist stability criterion. This uses a DSL based on the DSLV that block diagrams provides in the context of control theory, shown below. The DSL consists of four constructors, one of which (**TF**) constructs an LTI system from a name given to its transfer function. Its semantic value is specified by a parameter in the evaluation function. The other three constructors compose LTI systems by combining other LTI systems by functional composition (**:->**), summation (**:<>**), or by closing a loop (**FB**). These constructors' semantics is reflected by the evaluation function show below.

```

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

```

```

evalLTI :: (String -> (C -> C)) -> LTI -> (C -> C)
evalLTI d (TF s) = d s
evalLTI d (a :-> b) = \z -> evalLTI d a z * evalLTI d b z
evalLTI d (a :<> (b, c)) = \z -> eval d a z * (eval d b z + eval d c z)
evalLTI d (FB a b) = eval d a / (1 - eval d a * eval d b)

```

4.4 The Nyquist Stability Criterion

The section starts by informally explaining stability and introducing the Nyquist stability criterion—a method used to determine whether a system is stable. The importance of poles and zeroes of a function for the Nyquist criterion is then explained, whereafter contours are introduced in order to allow for contour mapping as a tool to determine the location of zeroes and poles of a function. Where earlier sections rely more heavily on deep embeddings, this section uses shallow embeddings to a greater extent, as this is used to describe contour mapping. Since the Nyquist stability criterion describes the relation between poles of a transfer function and whether the corresponding system is stable, the criterion is then explained with these new tools.

5 Discussion

The use of DSLs and Haskell to teach other subjects is—as previously discussed in section 1—an interesting approach that has proven itself useful in various mathematical domains for a number of reasons. In this section, the project will be evaluated and discussed in detail.

5.1 Evaluation of the Learning Material

The idea of using types and EDSLs to teach mathematical subjects such as control theory relies on the idea of instrumental scaffolding, discussed in section 2.4.1. Students that are comfortable with programming—and especially functional programming—can use this knowledge as cognitive scaffolding to have an easier, more intuitive way into new subjects. To enable this, students must have a learning material or guided instruction that uses this scaffolding in order to teach the subject matter, which is what the learning material developed for this report does. The material also has the advantage of being able to instruct a narrower group of students—a teacher has to accommodate everyone’s needs simultaneously, whilst the material allows itself to have a much smaller target audience. The intent is that this will be enough for a student to enter a ZPD, and after having used the material have the cognitive scaffolding required to be able to continue their learning of control theory in a non-computer science environment.

For the scope of this project, it is not possible to cover all of the course content of the targeted course. Some essential parts had to be prioritised over others, some areas are more central for the understanding of control theory and fundamental for further learning than others, and some areas are not as well suited for programmatic DSLs as others.

Section 3 mentions which domains described in the learning material have already had DSLs developed in previous projects [6], [11], and whether or not the DSL is used in the learning material, however, the reasoning behind whether to reuse a previously developed DSL or to make a new one is not explained. When investigating the DSLs from the previous projects, all of them were developed in ways not congruent with the envisioned description in the learning material. For example, the section on Laplace transforms heavily leans on the tabular approach, which in our opinion was served by allowing the creation of expressions that could be transformed. The approach taken by DSLsofMath is to not implement the Laplace transform, but to show how it can be used to calculate the solution to a differential equation. This approach is more similar to the ordinary mathematical approach to using the Laplace transform, and thus our implementation could fill what we regarded as an empty niche. Similar thoughts guided the decisions on whether to keep the DSLs developed in previous projects, with one exception: in later parts of the learning material, the DSLs developed relied on an implementation of complex numbers. In order to maximise utility for these DSLs, a more extensive DSL for complex number would be better. Thus the DSL for complex numbers developed by “TSS with DSLs” [11] was adapted to fit our DSL and used as a dependency in later DSLs.

5.1.1 Prerequisites

A significant part of the learning material is taken up by content that is related to the prerequisite knowledge of the course, and one might argue that this puts the focus of the material on what is not of the subject at hand. But, as mentioned in section 2.1, many students do not have enough prerequisite knowledge to take full advantage of the course. For this reason, an introductory part was included to in part serve as aid for some of this missing or unpractised knowledge. As to exactly what prerequisites this section would best address, there are many possible answers. The course plan [10] for the course lists complex numbers, linear algebra, Taylor series, ordinary differential equations, the Laplace transform, and knowledge of basic relations from physics, as prerequisites—any of which would be good candidates for this section. Additional concepts could also be argued are lacking in the students’ repertoire of knowledge, such as line integrals which are only briefly discussed in the course before applying them to Nyquist diagrams.

Of all of these areas complex numbers, integrals, and Laplace transforms were chosen to be included in the material. Complex numbers are arguably, for many students, probably of the least concern, since we can assume they have this knowledge from their secondary education. However, the section on complex numbers in the learning material has an additional purpose of serving as a refresher on Haskell syntax and familiarising the reader with the DSL approach of thinking about mathematical domains, and would therefore benefit from describing an area already familiar to the reader.

Integrals are presumably also rather familiar to the intended reader, whose section in the learning material has a similar purpose to that of the section on complex numbers. What is different from that section, however, is how the respective subjects are traditionally taught in relation to their types. The type of the integration operation is a higher-order type, which is generally not taught when learning about integrals—likely giving a less rigorous understanding of the operation. As the understanding of types is central to this DSL approach of learning, integrals were deemed important that they be included in the introductory part of the learning material. Furthermore, integrals may be seen as very essential to subsequent parts in the learning material.

Two different ways of calculating integrals were presented in the learning material. A potential downside of this structure is that once students have understood the first solution, they may lose interest in the latter. For our purposes, leaving it up to the students how deep they want their understanding to be is acceptable, as understanding only the numerical solution should be sufficient for later sections. Another thing to consider is the choice of using a numerical way to calculate integrals. In the course you mostly solve integrals analytically. It can also be argued that numerical solutions do not give very good understanding and intuition for the technique. On the other hand, not all integrals can be solved analytically and it is also important that the student understands how the integrals are computed in that case. We do however provide two ways of calculating integrals which means we get the benefits of both approaches.

Lastly among the prerequisites, the Laplace transform was explored in the learning material. This section comes somewhere between introductory content and main content of the subject, as the Laplace transform is quite heavily and

directly used in many of the areas addressed in the course. Tables of different known Laplace transforms and combinations of them are used when learning how to compute the Laplace transform. The same technique is used in the implementation described in the learning material. The amount of code required might be intimidating to the reader at first, but a pattern quickly becomes clear. This is probably beneficial to the students because this shows that the Laplace transform always uses the same methods.

5.1.2 Core Course Subjects

Transfer functions are used in most of control theory and the course has a clear focus on linear time-invariant systems, so these parts were deemed important and take up a significant part of the material.

The Laplace transform, while not easy, was well suited to implement as a DSL. Some of the more involved areas of control theory proved quite difficult to construct DSLs for. For example, the Nyquist criterion, which uses numerical approximations and intuitions that are hard to represent with DSLs. The implementation has begun, but is not complete. Bode diagrams was initially intended to be included in the learning material, but similar problems hindered its development. We believe that it is possible to construct a good and useful DSL for Bode diagrams, but it was not possible in the time frame of this project.

The other core course subjects such as PID-regulators were not implemented due to their dependency on other areas not implemented, such as Bode diagrams. This is however something that can be worked more upon in the future.

5.1.3 Exercises

There are a few select exercises included in the learning material. It would have been useful to have more exercises included in the material, but this ended up in second priority to other aspects of the material such as DSL, text design, and subject research. Optimally, more exercises would have been included in order to allow students to interact with the learning material to a greater degree, yet ERE103 has an abundance of exercises. Exercises differs somewhat from text describing correlations and how the concepts in a field fit together. The main difference being that exercises teaches students what problems are typical of the field, and how to apply the theory to those kinds of problems.

5.2 The Use of Haskell as a Host Language

Of the various ways of implementing a DSL, embedding it in a host language, as discussed in section 2.2.1, requires some of the least effort and is easy to extend and modify. Functional languages are popular as host languages as they are often very expressive and flexible [21]. Operators are often overloadable which opens up the possibilities for making even more tailored DSLs—making implementation of DSLs for many mathematical domains especially natural. Haskell is one such language, that, as mentioned in 2.2.1, has become popular for making embedded DSLs [21], which is in part why it was deemed suitable for the project. In the case of Haskell, implementing types as instances of type classes takes the role of

overloading operators. Previous work in the same area [11], [13], [30] have also used Haskell as a host language, and the students of the CSE program—in which the targeted control theory course resides—at Chalmers has a course about Haskell in their first year, so this choice came quite naturally.

One possible drawback to embedding DSLs in Haskell for mathematical domains is the strictly one-dimensional nature of such code. Mathematical notation often utilises vertical alignment—writing above and below other mathematical symbols—to convey information and make the syntax easily readable. For example, the notebook-style application Wolfram Mathematica [31] allows for this kind of two-dimensional syntax input via the Wolfram Language. This kind of input, although very useful for its purposes, was not deemed as suitable for this project as the syntax is much closer to the original mathematical syntax, making it less familiar to those comfortable with regular program code. In order to utilise this familiarity we chose to use Haskell to tailor the DSLs to computer science students.

Using Haskell—or any other programming language—as a host language for DSLs has a risk of alienating a significant portion of the target audience to some extent, as they may not be very familiar with the language. It could thus arguably be a good idea to be restrictive with language-specific implementation, or even use pseudo-code. As has been done in previous work [6], Haskell’s type checker could be relied on without specific implementations of functions, and by these means still serve as effective learning material. Arguably, there could still be beneficial for those familiar with the language used to study the implementation to some extent, and those unfamiliar need not direct their attention to implementation details. The learning material developed relies on this latter view, which is beneficial to disclose to the reader so that they have appropriate expectations and use the material in an effective way.

As a secondary kind of DSL—a DSVL of block diagrams native to control theory—has been used. These are meant to serve as a comfortable middle-ground between control theory and the Haskell DSLs presented in the learning material. Discussing the types of the components involved in the diagrams can provide a more rigorous understanding of the diagrams’ relation to LTI systems, transfer functions, and impulse response, among other things. Relating these diagrams to the Haskell DSLs presented makes for a clearer connection between the different representations.

5.3 Medium of Presentation

The material was chosen to be presented in a textual form, and specifically as a PDF. This is in line with a lot of other course material, and as the content is mostly code based, and previous work has been presented in the same way, it was a natural choice to present it in text form. A website with interactive exercises was long considered, but did not fit the time constraints of the project and was deemed less important than other aspects such as the content scope of the material. Other mediums could have been suitable as well, such as interactive tools with a DSVL focus, or video lectures. The reason these were not chosen for this project is that these would have taken a lot more time and effort away from expanding the content scope and refining the presentation. A textual presentation was deemed the most

time-efficient, assessed to yield the most amount of valuable content, and have the least amount of hurdles in the way of reaching our intended goal.

The choice of language to present the material in was not entirely trivial, as the course that the material specifically targets is held exclusively in Swedish [10]. The students in question are assumed to have an English language proficiency adequate to effectively use the material. However, since a lot of terminology differs between the languages, the material risks becoming less accessible to the attendants of that course, and terminology might get confused. On the other hand, presenting the topic in English opens up for a larger audience that may access the learning material; both for learners of control theory and for those who wish to conduct further research on the matter. In addition, this may work as a middle ground, possibly helping students learn the English terminology, allowing them to read a broader literature on the subject.

5.4 Future Projects

There are three main ideas for projects that can build on our work. The first would be a project that evaluates our learning material. This could be done by testing it on a group of students and evaluating whether it improves their understanding of the subject or improves their performance on the exam. We were unable to test it on students taking the targeted course as the course runs in fall and this project took place during spring.

The other project would simply be a continuation of this project's material. As mentioned before, the learning material is not complete in relation to the scope of the targeted course, or the scope of control theory as a whole, and there are several areas of the learning material that could be improved. As such, a similar project could be performed where the groundwork has already been laid, and could therefore be much further developed than this project has. Such a project could explore areas of control theory that was not covered by this project, or add more material to the existing parts such as exercises for the reader to do.

Finally, the third idea for a project that can build on our work is a project that develops a learning material for a different course or subject. The same methodology using DSLs can be applied to many different areas and can be beneficial to learning.

5.5 Ethics and Utility

The most important societal aspect to mention of this project is the fact that this is a free supplementary learning material which can benefit students and help them in their learning. The impact of this, of course, depends on how many students use the material and the quality of this material. A possible negative economic effect of this is that students may choose to use this instead of the recommended course literature. However we do not think our project will have a very large impact on the sales of control theory literature as the material is not complete and does not cover all the areas covered in the control theory course literature.

There are some other ethical considerations concerning this project. One of which concerns the fact that this project does not empirically establish the effectiveness of the learning material, but is instead entirely reliant on parts from the

theory of didactics explored in this report. Presenting the material as an effective alternative approach may not be ethical in case it is not as useful to this group of students as presumed, or even make the subject more confusing. Therefore, the learning material should arguably be clearly presented as untested and/or experimental.

Furthermore, as the material developed is meant to help students understand the subject better, although not that likely, the risk for the material to, for instance, make it easier to pass the exam without understanding the material, should be considered. This could be an issue if the learning material focuses too much on the programmatic side without emphasising the concepts and physical aspects of control theory. It may also be important that it be emphasised that the learning material cannot replace any part of the targeted—or any other—course, and only exists as supplementary material.

It is also important that the material is available to everyone—not only a select few. This is assured by making all of the work available on GitHub and also written in English. The aim is also to have the course material published on the ERE103 website, which means it would be easily available for all students in the course.

6 Conclusions

In this report we have presented the learning material we have developed as a supplementary aid for the control theory course ERE103. The main idea in this learning material is the use of DSLs as a teaching aid.

The resulting material consists of four main sections; Prerequisites, the Laplace transform, Transfer functions, and Nyqvist diagrams. Each of these sections aim to rephrase complex mathematical constructs as programming exercises. The idea being that the students in question would be more familiar with this mode of thinking, and thus have an easier time understanding it.

The text was written in English, and associated DSLs were created using Haskell. English was chosen in order to reach a larger audience. While the intended audience is primarily Swedish, producing a Swedish text creates an inherent limitation to who can read and make use of it. While any programming language could theoretically create a DSL, the familiarity of the audience was deemed a high priority, so as to keep the emphasis of the student–text interactions on the contents of the text itself, rather than on learning another language.

Since no instances of the ERE103 course was held concurrently to this project, there were no opportunities to test the resulting material. Didactic theory and the results of prior projects using DSLs both support the effectiveness of this project. Nonetheless, it is an undesired loose end, and a future project can be performed that tests the effect it has on students learning. A continuation of the project that completes the learning material could be useful. Exploring similar domains could also yield further interesting results.

References

- [1] C. Wells, “Communicating mathematics: Useful ideas from computer science”, *The American mathematical monthly*, vol. 102, no. 5, pp. 397–408, 1995. DOI: [10.2307/2975030](https://doi.org/10.2307/2975030). [Online]. Available: <https://www.jstor.org/stable/2975030?origin=crossref&seq=1>.
- [2] G. J. Sussman and J. Wisdom, *Functional differential geometry*. MIT Press, 2013.
- [3] D. Gries and F. B. Schneider, *A logical approach to discrete math*. Springer Science & Business Media, 2013.
- [4] S. N. Walck, “Learn quantum mechanics with Haskell”, *Electronic Proceedings in Theoretical Computer Science*, 2016, ISSN: 2075-2180.
- [5] —, “Learn physics by programming in Haskell”, *Electronic Proceedings in Theoretical Computer Science*, vol. 170, no. Proc. TFPIE 2014, pp. 67–77, 2014, ISSN: 2075-2180.
- [6] P. Jansson and C. Ionescu. (2020). Domain specific languages of mathematics, [Online]. Available: <https://github.com/DSLsofMath/DSLsofMath/> (visited on 05/06/2020).
- [7] M. Fowler and R. Parsons, *Domain Specific Languages*. Pearson Education (US), 2010.
- [8] Chalmers University of Technology. (2018). Domain specific languages of mathematics, [Online]. Available: https://student.portal.chalmers.se/en/chalmersstudies/courseinformation/Pages/SearchCourse.aspx?course_id=27332&parsergrp=3 (visited on 05/06/2020).
- [9] P. Jansson, S. H. Einarsdóttir, and C. Ionescu, “Examples and results from a bsc-level course on domain specific languages of mathematics”, *Electronic Proceedings in Theoretical Computer Science*, vol. 295, pp. 79–90, 2019, ISSN: 2075-2180. DOI: [10.4204/eptcs.295.6](https://doi.org/10.4204/eptcs.295.6). [Online]. Available: <https://arxiv.org/abs/1908.01572v1>.
- [10] Chalmers University of Technology. (2018). Control theory, [Online]. Available: https://student.portal.chalmers.se/en/chalmersstudies/courseinformation/Pages/SearchCourse.aspx?course_id=27699&parsergrp=3 (visited on 05/06/2020).
- [11] J. Jonsson, P. Ngo, C. Rosvall, F. Lindahl, and J. Olsson, “Programmering som undervisningsverktyg för transformer, signaler och system. utvecklingen av läromaterialet TSS med DSL”, Bachelors’ thesis, 2016. [Online]. Available: <https://odr.chalmers.se/bitstream/20.500.12380/243894/1/243894.pdf>.
- [12] Chalmers University of Technology. (2018). Transforms, signals and systems, [Online]. Available: https://student.portal.chalmers.se/en/chalmersstudies/courseinformation/Pages/SearchCourse.aspx?course_id=28314&parsergrp=3 (visited on 05/06/2020).

- [13] J. Johansson, O. Lundström, E. Sjöström, and B. Werner, “Ett komplementerande läromaterial för datastudenter som lär sig fysik”, Bachelors’ thesis, 2018. [Online]. Available: <https://odr.chalmers.se/bitstream/20.500.12380/256122/1/256122.pdf>.
- [14] —, (2018). Learn you a physics for great good, [Online]. Available: <https://dslsofmath.github.io/BScProj2018/index.html>.
- [15] Chalmers University of Technology. (2018). Physics for engineers, [Online]. Available: https://student.portal.chalmers.se/en/chalmersstudies/courseinformation/Pages/SearchCourse.aspx?course_id=27951&parsergrp=3 (visited on 05/06/2020).
- [16] —, (Feb. 2020). Course evaluation minutes, [Online]. Available: https://webbpublicering360.portal.chalmers.se/Extern/Home/Download?recordnor=760464%5C%262020_04%5C%26909827_1_1.PDF%5C%26st (visited on 05/14/2020).
- [17] P. Hudak, “Building domain-specific embedded languages”, *ACM Comput. Surv.*, vol. 28, no. 4es, 196–es, Dec. 1996, ISSN: 0360-0300. DOI: [10.1145/242224.242477](https://doi.org/10.1145/242224.242477). [Online]. Available: <https://doi.org/10.1145/242224.242477>.
- [18] A. van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography”, *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000, ISSN: 0362-1340. DOI: [10.1145/352029.352035](https://doi.org/10.1145/352029.352035). [Online]. Available: <https://doi.org/10.1145/352029.352035>.
- [19] M. Mernik, J. Heering, and A. M. Sloane, “When and how to develop domain-specific languages”, *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005. DOI: [10.1145/1118890.1118892](https://doi.org/10.1145/1118890.1118892).
- [20] R. Esser and J. W. Janneck, “A framework for defining domain-specific visual languages”, in *Proceedings of the Workshop on Domain Specific Visual Languages at OOPSLA*, 2001.
- [21] C. Lejdfors, “Techniques for implementing embedded domain specific languages in dynamic languages”, Licentiate Thesis, Lund University, 2006. [Online]. Available: http://fileadmin.cs.lth.se/cs/Personal/Calle_Lejdfors/publications/lic.pdf.
- [22] B. Lennartson, *Reglerteknikens Grunder*. Studentlitteratur, 2005, vol. 4.
- [23] E. Fernandez-Cara and E. Zuazua, “Control theory: History, mathematical achievements and perspectives”, p. 62, 2003. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.5633&rep=rep1&type=pdf>.
- [24] S. P. Linder, D. Abbott, and M. Fromberger, “An instructional scaffolding approach to teaching software design”, *Journal of Computing Sciences in Colleges*, vol. 21, pp. 238–250, Jun. 2006.
- [25] L. S. Vygotskii, M. Cole, S. Stein, and A. Sekula, *Mind in society: the development of higher psychological processes*. Harvard University Press, 1978.
- [26] G. Imsen, *Elebens Värld*. Universitetforlaget, 2005, vol. 4.

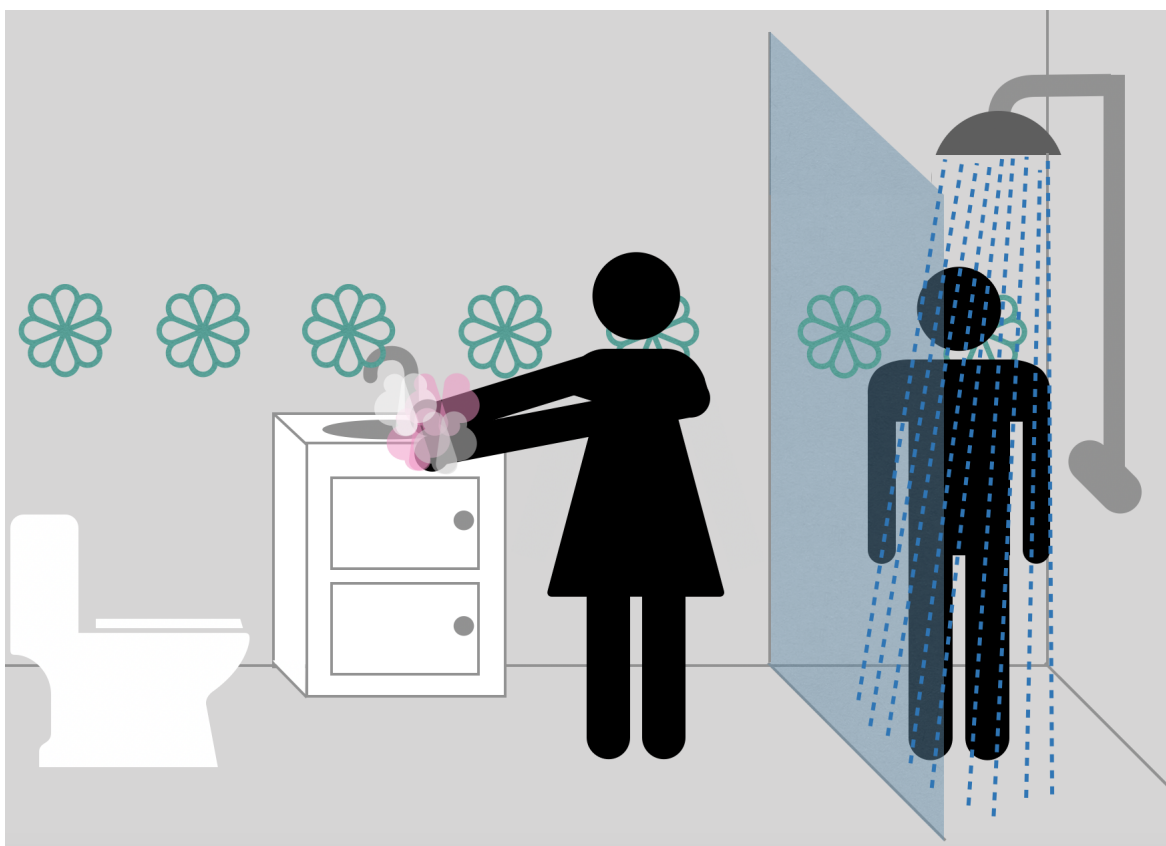
- [27] OECD, “Result from Talis 2013: Sweden”, 2014. [Online]. Available: <http://www.oecd.org/sweden/TALIS-2013-country-note-Sweden.pdf>.
- [28] M. O’Loughlin, “Rethinking science education: Beyond piagetian constructivism toward a sociocultural model of teaching and learning”, *Journal of Research in Science Teaching*, vol. 29, no. 8, pp. 791–820, 2007. DOI: [10.1002/tea.3660290805](https://doi.org/10.1002/tea.3660290805).
- [29] S. Hadjerrouit, “Constructivism as guiding philosophy for software engineering education”, *SIGCSE Bull.*, vol. 37, no. 4, pp. 45–49, Dec. 2005, ISSN: 0097-8418. DOI: [10.1145/1113847.1113875](https://doi.org/10.1145/1113847.1113875). [Online]. Available: <https://doi.org/10.1145/1113847.1113875>.
- [30] C. Ionescu and P. Jansson, “Domain-specific languages of mathematics: Presenting mathematical analysis using functional programming”, *Electronic Proceedings in Theoretical Computer Science*, vol. 230, pp. 1–15, 2016, ISSN: 2075-2180. DOI: [10.4204/eptcs.230.1](https://doi.org/10.4204/eptcs.230.1). [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.230.1>.
- [31] Wolfram Research, Inc., *Mathematica*, version 12.1, 2020. [Online]. Available: <https://www.wolfram.com/mathematica>.

A Learning material

Here follows a snapshot as of 2020-06-05 of the learning material that was developed during this project. Please note that this is a work still in progress and to see an up to date version please visit <https://github.com/simonhagglund/DATX02-dsl>.

Domain-specific languages of Control Theory

A supplementary learning material for ERE103



Jakob Fihlman

Simon Hägglund
Elin Ohlman

Christian Josefson
Tommy Räjert

Filip Nylander

Spring - 2020

Table of Contents

1	Introduction	3
1.1	Basic control theory	3
2	Prerequisites	4
2.1	Complex numbers	4
2.1.1	The Complex Number Concept	4
2.1.2	Basic Operators	5
2.1.3	Complex Geometry	6
2.1.4	Advanced Operators	8
2.2	Integrals	10
2.2.1	Setting up the Language	10
2.2.2	Brute Force Mathematics	11
2.2.2.1		13
2.2.3	An Elegant yet Primitive Solution	13
3	Laplace transform	14
3.1	The Expression datatype	15
3.1.1	Some conventions and notations	17
3.2	The definition of the Laplace transform and corresponding types	17
3.3	Common rules, how to use them	18
3.3.1	Superposition	18
3.3.2	Derivative rule	20
3.3.3	Integral rule	22
3.3.4	Convolution	22
3.3.5	Time shift	22
3.3.6	Exponential decay (frequency shift)	23
3.4	Inverse laplace transform	23
4	Transfer Functions and LTI Systems	23
4.1	Linear Time-Invariant Systems	23
4.2	Types	24
4.3	Linear Time-Invariant Systems – Part 2	26
4.4	Combining Transfer Functions	27
4.5	Finding Transfer Functions	28
4.6	Combining Systems	30
5	Stability	32
5.1	The Nyquist Criterion of Stability	33
5.1.1	Contours	34
5.1.2	Cauchy’s Argument Principle	34
5.1.3	Nyquist Criterion	35
A	Code	37

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.

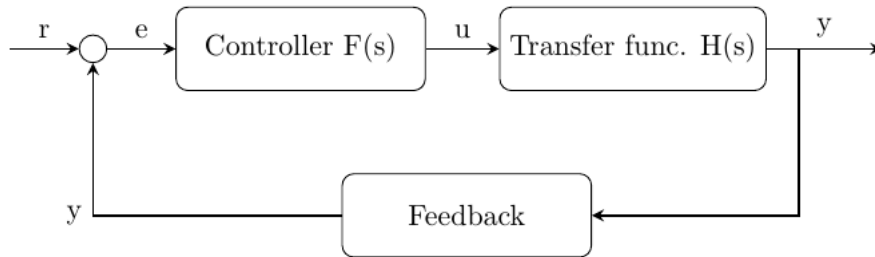


Figure 1: Feedback-loop

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 three 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 Nyquist 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:

```
1 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.

```
1 data Complex = Complex (R, R)
2   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:

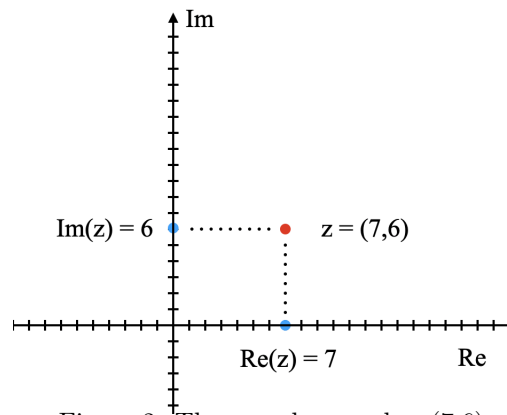


Figure 2: The complex number $(7,6)$

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:

```
1 whatsReal :: Complex -> R
2 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.

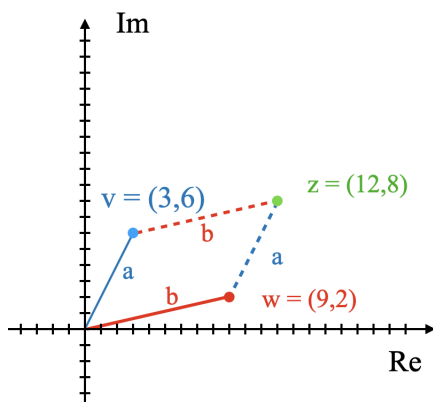


Figure 3: Addition of two complex numbers v and w , and the resulting number z .

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:

```
1 add :: Complex -> Complex -> Complex
2 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.

```
1 sub :: Complex -> Complex -> Complex
2 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:

```
1 absolute :: Complex -> R
2 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.

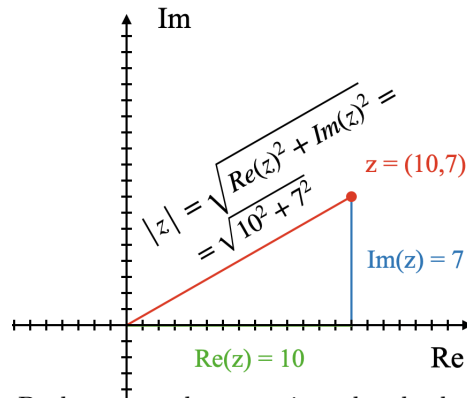


Figure 4: The Pythagorean theorem gives the absolute value of (10,7)

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.

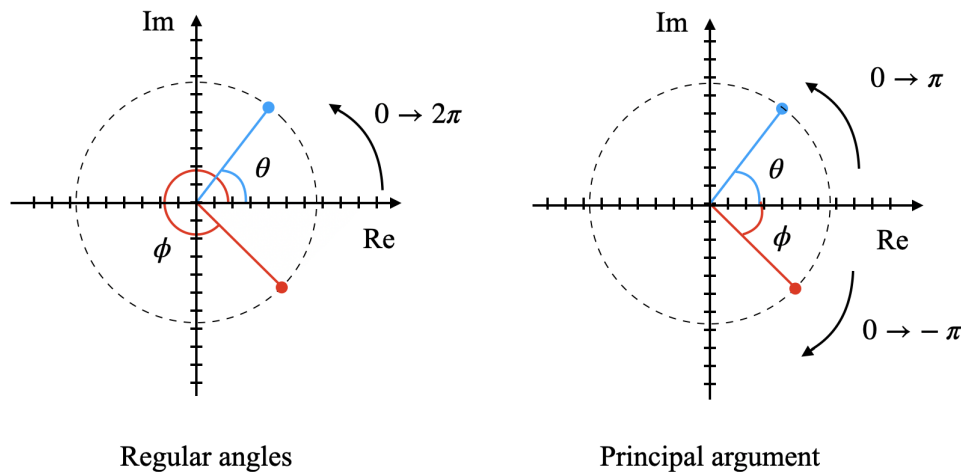


Figure 5: Angles vs arguments

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.

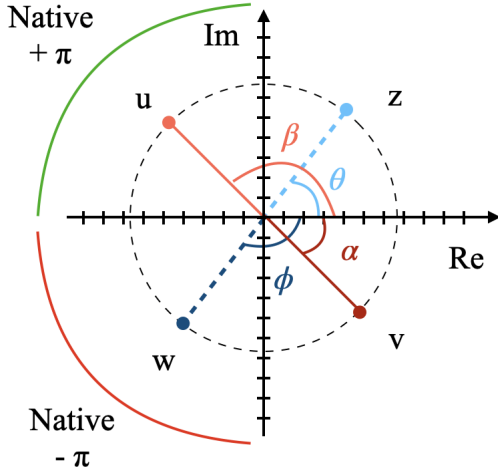
```

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

```

$$\tan(\theta) = \frac{\text{Im}(z)}{\text{Re}(z)} = \tan(\phi) = \frac{\text{Im}(w)}{\text{Re}(z)} = k$$

$$\tan(\alpha) = \frac{\text{Im}(u)}{\text{Re}(u)} = \tan(\beta) = \frac{\text{Im}(v)}{\text{Re}(v)} = l$$



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

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

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

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

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

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

Figure 6: Translating arcus tangents to principal arguments

```

4 | real < 0           = (atan (imaginary/real)) - pi
5 | 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 $+90^\circ$, -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:

$$\begin{array}{c}
 (1 + 2i) * (3 + i) = \\
 \begin{array}{cccc}
 1 & * & 3 & 1 & * & i & 2i & * & 3 & 2i & * & i \\
 \text{(green)} & & \text{(blue)} & \text{(green)} & & \text{(red)} & \text{(red)} & & \text{(blue)} & \text{(red)} & & \text{(red)}
 \end{array} \\
 = 3 + i + 6i + 2i^2
 \end{array}$$

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.

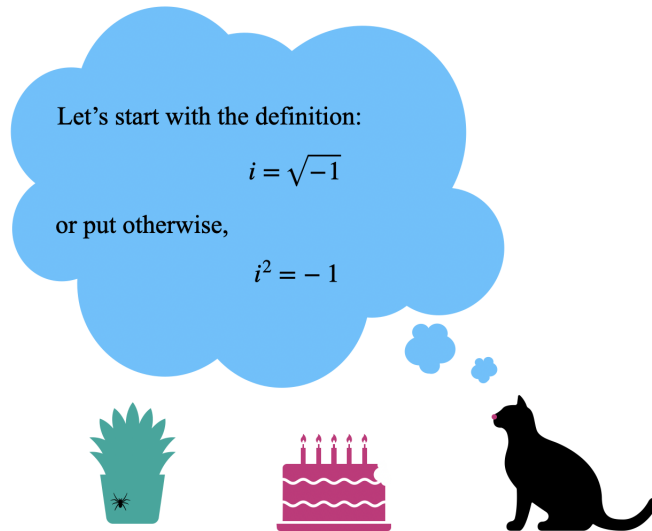


Figure 8: Comment regarding the definition of i that we brought up at the beginning of the section and fully expect you to have forgotten about by now

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.

```

1 multiply :: Complex -> Complex -> Complex
2 multiply (Complex (r1,i1)) (Complex (r2,i2))
3   = 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.

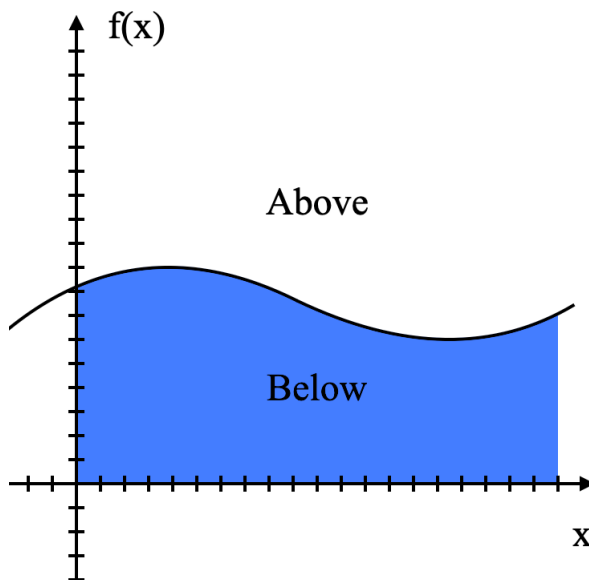


Figure 9

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:

```
1 data Funk =
2   Constant Double
3   | X Double
4   | XSquare Double
5   | Add Funk Funk
6   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; "aX²" can be 5X², 0.33X², etc. Finally, "Add" is used to string our words together into "sentences". As an example, the polynomial function "5 + 4X²" would in our DSL be written as

```
1 test :: Funk
2 test = Add test1 test2 where
3   test1 = Constant 5
4   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:

```
1 calculate :: Funk -> Double -> Double
2 calculate (Constant double) value = double
3 calculate (X double) value         = double * value
4 calculate (XSquare double) value   = double*(value^2)
5 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.

```

1  -- bfIntegral (brute force integral)
2  --input: function, startvalue, stopvalue
3  --output: result
4  bfIntegral :: Funk -> Double -> Double -> Double
5  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.

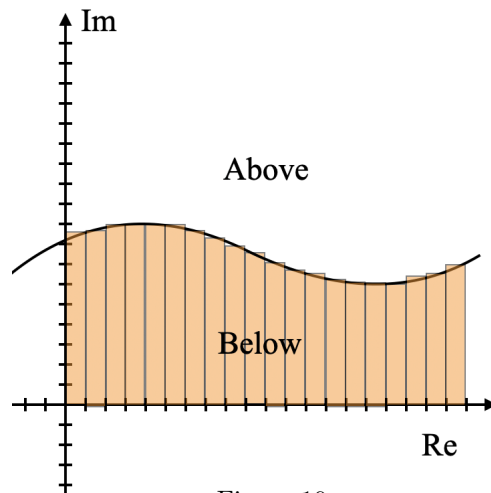


Figure 10

```

1  -- bfIntegral' (brute force integral alternative)
2  --input: function, start, end
3  --output: result
4  bfIntegral' :: Funk -> Double -> Double -> Double
5  bfIntegral' funk start end
6  | start >= end = 0
7  | 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:

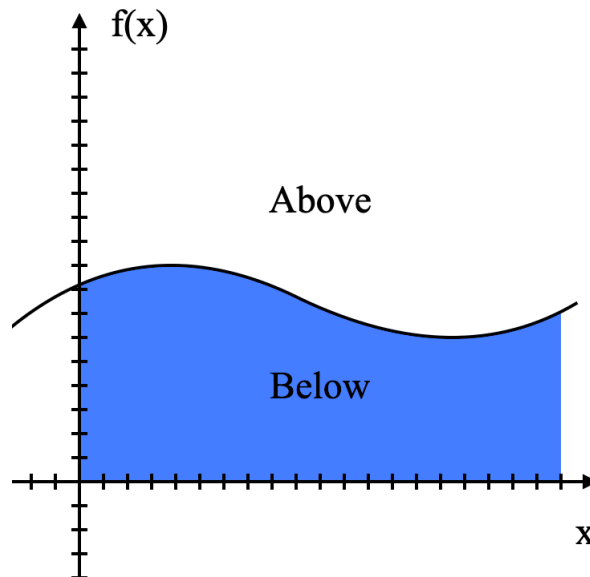


Figure 11

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 derivatation function on a micro-level, and instead limit ourselves to the more commonly used laws of derivation. Specifically, let's invert them:

```

1 primitivize :: Funk -> Double -> Funk
2 primitivize funk constant = Add (primitivize' funk) (Constant constant)
3
4 primitivize' :: Funk -> Funk
5 primitivize' (Constant double) = X double
6 primitivize' (X double)        = XSquare (double/2)
7 primitivize' (XSquare double)  = -- Your implementation of powers goes here; power=3, double/3
8 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:

```

1 primitiveTest :: Funk
2 primitiveTest = primitivize test 0
3 (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/DLSofMath/DLSofMath/>

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 ??.

```
1 module Laplace where
2   import ComplexNumbers
3   import FunNumInst
```

3.1 The Expression datatype

Like in section 2.2, we create a recursive datatype. We will not state it in its 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`.

```
1 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 `Expressions`. 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.

```
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:

```
1 testExpression :: Expression Complex
2 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 `Expressions` is to draw 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

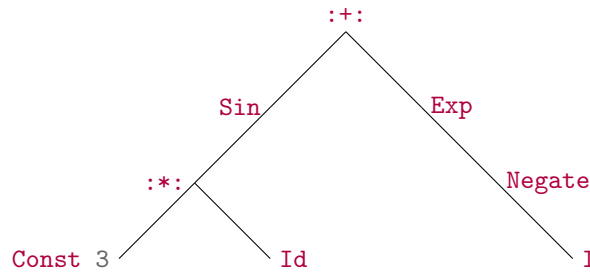


Figure 12: Sketch of a syntax tree; I imagine there being either small circle around each command or possibly just space (like in Communicating Mathematics ...)

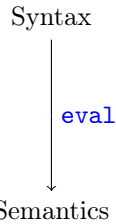


Figure 13: Diagram showing the syntactic and semantic levels, as well as the eval function between them. (The arrowhead is a little small)

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 **Expressions** represent? That's where the semantics come in, they are the meaning of our **Expressions**. 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.

```

1  shift :: Num a => a -> (a -> a) -> (a -> a)
2  shift tau f = \t -> f (t-tau)
3
4  eval :: (Num a, Floating a, Eq a) => Expression a -> a -> a
5  eval (Const c)           = \t -> c
6  eval Id                  = id
7  eval Pi                  = eval (Const pi)
8  eval (e1 :+: e2)         = eval e1 + eval e2
9  eval (e1 :*: e2)         = eval e1 * eval e2
10 eval (Shift tau e)        = shift tau (eval e) -- shifts e by tau, i.e. begins after tau seconds
11 eval (Negate e)          = negate (eval e)
12 eval (Exp e)              = exp      (eval e)
13 eval (Sin e)              = sin      (eval e)
14 eval (Cos e)              = cos      (eval e)
15 eval (Tan e)              = tan      (eval e)
16 eval (Asin e)             = asin     (eval e)
17 eval (Acos e)             = acos     (eval e)
18 eval (Atan e)             = atan     (eval e)
19 eval (Integral e e0)      = undefined -- TODO
20 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

```

1  -- TODO
2  tmp = eval testExpression
3  -- 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 primitive 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 \mathcal{L} :

$$\mathcal{L}\{f\} = s \mapsto \int_0^\infty e^{-st} f(t) dt.$$

There's a lot going on here. On the left hand side we have two symbols: \mathcal{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

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

or using `Expression`:

```
1 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

```
1 laplace' :: Expression R -> Expression Complex -> Expression Complex
2 laplace' f = Integ (Const 0) (Infinity) (exp *: f)
3           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 \mathcal{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

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

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

```

1  -- rules
2  laplace :: Expression Complex -> Expression Complex -- TODO: maybe instead do Time->Freq
3  laplace (e1 :+: e2) = laplace e1 :+: laplace e2 -- Superposition
4  laplace (Const a **: e) = Const a **: laplace e -- scaling
5  laplace (Derivative e e0) = Id **: laplace e :-: (Const e0) -- L {f'} = \s -> s * L {f} - f(0)
6  laplace (Integral e e0) = laplace e :/: Id -- L {int_0^t f(x) dx} = s -> L {f} /s
7  laplace (Shift tau e) = Exp (Id **: Const tau) **: laplace e -- TODO: double check signs
8  -- TODO: double check signs on next two
9  laplace (Exp (Negate (Const tau **: Id)) **: e) = Shift tau (laplace e)
10 laplace (Conv e1 e2) = laplace e1 **: laplace e2 -- convolution turns into multiplication
11 -- transforms
12 laplace (Impulse) = Const 1 -- L {delta(t)} = \s -> 1
13 laplace (Const 1) = Const 1 :/: Id -- L {a} = \s -> a/s; a = 1 => L {1} = 1/s
14 laplace (Id) = Const 1 :/: (Id **: Id) -- L {t} = \s -> 1/(s^2)
15 laplace (Exp (Negate (Const a **: Id))) = Shift (negate a) (Const 1 :/: Id) -- TODO: Check signs
16 laplace (Const 1 :-: Exp (Negate (Const a **: Id))) = Const a :/: (Id **: Shift a Id) -- TODO: Double
17
18 laplace (Exp (Negate (Const a)) :-: Exp (Negate (Const b))) =
19   Const (b - a) :/: (Shift (negate a) Id **: Shift (negate b) Id)
20 laplace (Id :-: (Const 1 :-: Exp (Negate (Const a) **: Id)) :/: Const a) =
21   Const a :/: (Id **: Id **: Shift (negate a) Id)
22 laplace (Const 1 :-: (Const 1 :+: Const a **: Id) **: Exp (Negate (Const a) **: Id)) =
23   Const (a*a) / (Id **: Shift (Negate a) (Id **: Id)) -- Const a **: Const a instead of Const (a*a)?
24 -- TODO: Doublecheck signs on the next two
25 laplace (Exp (Negate (Const a **: Id)) **: Sin (Const omega **: Id)) =
26   Const omega :/: (Shift a (Id **: Id) :+: (Const (omega*omega)))
27 laplace (Exp (Negate (Const a **: Id)) **: Cos (Const omega **: Id)) =
28   (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 $\mathcal{L}\{t \mapsto e^t + e^{-t}\}$, for example, knowing that

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

and that

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

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

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

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:

```
1 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

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

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:

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

i.e. the derivative of **e** with initial 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 differential equation.

```

1  laplace (Derivative e 1)
2  == {- derivative rule -}
3  Id *: laplace e :-: Const 1
4  == {- from differential equation -}
5  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 $\mathcal{L}\{f\} = s \mapsto sF(s) - 1$.

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

$$\begin{aligned} f &= f', & f(0) &= 1 \\ F &= s \mapsto sF(s) + 1 \end{aligned}$$

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 $\mathcal{L}\{t \mapsto e^{-at}\} = s \mapsto \frac{1}{s+a}$ (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

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

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

$$\mathcal{L} \left\{ t \mapsto \int_0^t x \mapsto f(x) dx \right\} = s \mapsto \frac{\mathcal{L}\{f\}}{s}.$$

Note that the integral is a function of t .

In our DSL, this is written:

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

3.3.4 Convolution

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

$$f \otimes g = \int_0^t f(\tau)g(t - \tau) d\tau.$$

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

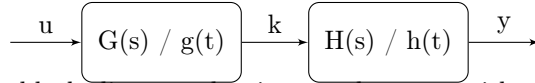


Figure 14: A simple block diagram showing a subsystem with transfer function $G(s)$ feeding into a subsystem with transfer function $H(s)$. $u(t)$ and $y(t)$ are the in- and output signals, respectively. $k(t)$ is the input of the H -system and output of the G -system. Adapted from 15

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

Mathematically, this means

$$\mathcal{L}\{f \otimes g\} = \mathcal{L}\{f\} \cdot \mathcal{L}\{g\} = F \cdot G$$

and in our DSL, this is written

```
1  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 $\mathcal{L}^{-1}\{F\}$, is given by

$$\mathcal{L}^{-1}\{F\}(s) = \frac{1}{2\pi} \lim_{T \rightarrow \infty} \int_{\gamma - iT}^{\gamma + iT} e^{st} F(s) ds.$$

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 it², since we can just use the rules to find the inverse instead!

4 Transfer Functions and LTI Systems

```
1 {- # LANGUAGE GADTs                #-}
2 {- # LANGUAGE StandaloneDeriving #-}
3
4 module LTlandTF where
5     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)
```

²However, 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:

$$\begin{aligned} \text{Time} &= \mathbb{R}^{>0}, \text{Signal} = \mathbb{R} \\ u, y : \text{Time} &\rightarrow \text{Signal} \end{aligned}$$

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:

$$\text{lti} : (\text{Time} \rightarrow \text{Signal}) \rightarrow (\text{Time} \rightarrow \text{Signal})$$

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 works well since *lti* is an endofunction (i.e. a function of type $a \rightarrow a$), 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:

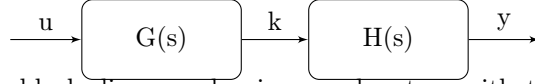


Figure 15: A simple block diagram showing a subsystem with transfer function $G(s)$ feeding into a subsystem with transfer function $H(s)$. $u(t)$ and $y(t)$ are the in- and output signals, respectively. $k(t)$ is the input of the H -system and output of the G -system.

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:

$$\begin{aligned}
\text{Time} &= \mathbb{R}^{\geq 0}, \text{ Signal} = \mathbb{R} \\
u, g, k, h, y &: \text{Time} \rightarrow \text{Signal}
\end{aligned}$$

```

6 type Time = Real
7 type Signal = Real
8 u, g, k, h, y :: Time -> Signal

```

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

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

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 $Y(s)/U(s) = G(s) \cdot H(s)$.

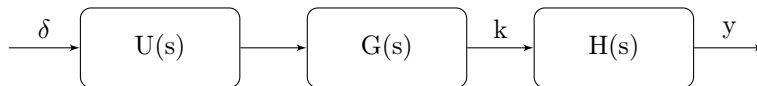


Figure 16: Extended version of the block diagram shown in figure 15. The input signal $u(t)$ has been replaced by its transfer function with δ as input, acting as an actuator for the system. This is identical to the previous representation, but highlights how the input can be shown as a system with a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

```

9 data Signal a where
10   -- Amp -> Freq -> Time-shift -> (a -> a)
11   Sin  :: Num a => a -> a -> a -> Signal a
12   Sum  :: Signal a -> Signal a -> Signal a
13   Scale :: a -> Signal a -> Signal a
14   Deriv :: Signal a -> Signal a
15   Integ :: Signal a -> Signal a
16   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.

```

17 simplifySignal :: (Num a, Eq a, Floating a, Ord a) => Signal a -> Signal a
18 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:

```

19 simplifySignal (Deriv (Sin a f s)) = Sin (a * f) (-f) (pi/2 - s)
20 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:

```

21 simplifySignal (Sum (Sin a1 f1 s1)
22                   (Sin a2 f2 s2)) | f1 == f2 = Sin a f1 s where
23   a = sqrt $ q1 ^ 2 + q2 ^ 2
24   s = atan $ q2 / q1
25   q1 = a1 * cos s1 + a2 * cos s2
26   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:

```

27 simplifySignal other = let ss = simplifySignal in case other of
28   Scale k (Sum o1 o2)      -> Sum (ss $ Scale k o1) (ss $ Scale k o2)
29   Deriv  (Sum o1 o2)      -> Sum (ss $ Deriv  o1) (ss $ Deriv  o2)
30   Integ  (Sum o1 o2)      -> Sum (ss $ Integ  o1) (ss $ Integ  o2)
31   Scale k a                -> ss $ Scale k $ ss a
32   Deriv  a                 -> ss $ Deriv  $ ss a
33   Integ  a                 -> ss $ Integ  $ ss a
34   Sum a b | ss a /= a || ss b /= b -> ss $ Sum (ss a) (ss b)
35   Sum a@(Sum a1 a2) b@(Sum _ _) -> ss $ Sum (ss $ Sum a1 b) (ss a2)
36   Sum a b | maxfreq a < maxfreq b -> ss $ Sum (ss b) (ss a)
37   Sum (Sum c a@(Sin _ af _)) b@(Sin _ bf _)
38       | af == bf -> ss $ Sum (ss c) (ss $ Sum a b)
39   _ -> 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:

```

40 evalSignal :: (Num a, Floating a, Eq a, Ord a) => Signal a -> (a -> a)
41 evalSignal (Sin a f s) t = a * sin (f * t + s)
42 evalSignal (Sum a b) t = evalSignal a t + evalSignal b t
43 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`:

```

44 evalSignal sig@(Deriv a) t = evalSignal (simplifySignal sig) t
45 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:

$$(\text{dirac } |*|) == (\text{id} :: (\text{Time} \rightarrow \text{Signal}) \rightarrow (\text{Time} \rightarrow \text{Signal}))$$

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:

```
46 data LCDE a where
47   LCDE :: ([a],[a]) -> LCDE a
48
49   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).

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

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).

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

$$C = \sum_{n=0}^m c_n s^n \quad (1)$$

```

50 data Transformed a where
51   Transformed :: ([a],[a]) -> [a] -> Transformed a
52
53   --          equation f~i(0) g~j(0) L{equation}
54   laplace :: Num a => LCDE a -> [a] -> [a] -> Transformed a
55   laplace (LCDE (lhs, rhs)) f0 g0 = transformed where
56     transformed = Transformed (lhs, rhs) (zipWithL 0 (-) lc rc)
57     lc = map sum [zipWith (*) f0 (drop n lhs) | n <- [1..length lhs-1]]
58     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 $\text{LCDE (lhs, rhs)} \mapsto \text{Transformed (lhs, rhs) } _$. 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:

```

59 data Transformed a where
60   Transformed :: ([a],[a]) -> Transformed a deriving Eq
61
62   laplace :: Num a => LCDE a -> Transformed a
63   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:

```

64 lcdeTrans1 = laplace lcde
65 lcdeTrans2 = Transformed ([9,6,1], [3,2])
66 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.

```

67 data RatExpr a where
68   RatExpr :: Num a => ([a],[a]) -> RatExpr a

```


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

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

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

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

```
69 solve2RatExpr :: Num a => Transformed a -> RatExpr a
70 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:

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

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

```
73 evalRE :: Fractional a => RatExpr a -> a -> a
74 evalRE (RatExpr (den,num)) s = numerator / denominator where
75     numerator   = unravel num
76     denominator = unravel den
77     unravel []  = 0
78     unravel list = head list + s * unravel (tail list)
79
80 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:

```
81 de = LCDE ([2,3],[-1,0,1]) :: Fractional a => LCDE a
82 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"**).

```
83 data LTI where
84     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:

```
85  (:->) :: 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 with a single operator to not have to deal with systems of several inputs or output:

```
86  (:<>) :: 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 dictionary³ of transfer functions and evaluates the LTI data type to its transfer function.

```
87  evalLTI :: (String -> (C -> C)) -> LTI -> (C -> C)
88  evalLTI dict (TF s) = dict s
89  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`.

```
90  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.

```
91  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:

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

```

92 data LTI where
93   TF      :: String -> LTI
94   (:->)   :: LTI -> LTI -> LTI
95   (:<>)   :: LTI -> (LTI, LTI) -> LTI
96   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,

```

97 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:

```

98 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)!

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

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:

```
1  --                                zeros poles
2  data TransferFunc = TransferFunc [C] [C]
```

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

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 $\backslash t \rightarrow \exp (t \cdot b x)$ 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 argument 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:

5.1.1.1 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 $d \rightarrow 0^+$ `ucContour` tends to an *approximation* of the unit circle contour.

```
3 ucContour :: R -> [C]
4 ucContour d = [C (cos (2*pi*k//n)) (sin (2*pi*k//n)) | k<-[0..n]] where
5   n = floor (2*pi/d)
6   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):

```
7 (*&)amp; :: R -> C -> C
8 c *& C r i = C (c * r) (c * i)
9 infixl 7 *& -- same as *
10
11 subdiv :: R -> (C, C) -> [C]
12 subdiv d (a, b) = [ a + n *& seg | n <- [1..nseg]] where
13   len = re.abs $ b - a
14   nseg = fromIntegral.floor $ len / d
15   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.

```
16 contourAng :: R -> [C] -> [C]
17 contourAng dist points = concatMap (subdiv dist) $ pairs ps where
18   ps = points ++ [head points]
19   pairs (a:b:cs) = (a,b) : pairs (b:cs)
20   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 $F(s)=(s-z_0)$, which has a zero at z_0 . If we map F onto a contour C surrounding z_0 in the complex plane, this is naturally equivalent to translating C with $-z_0$. Since C is defined to surround z_0 , the translated contour

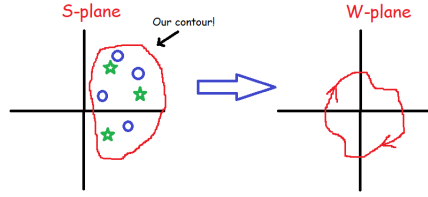


Figure 17: Mapping the values in a contour surrounding three zeroes and four poles in the s -plane (for a transfer function) is mapped to a contour surrounding the origin once. (This mapping is sometimes said to be in the “ W -plane.”)

will surround $z_0 + (-z_0) = 0$ (i.e. the origin). If we add a zero to the function ($F_2(s) = (s - z_0)(s - z_1)$) and let C surround both of these, $\text{map } f \text{ } c$ will be the element-wise product of the contours translated by $-z_0$ and $-z_1$, 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 surround the origin once, their element-wise product should do so twice. If, instead C surrounds only z_0 , when mapping $(s - z_1)$ on C , this translates C to surround $(z_0 - z_1)$ strictly without surrounding the origin.

In the general case, let $F_3(s) = (s - z_0)g(s)$, with a zero at z_0 , for some function g . In this case, the contour is translated separately by $-z_0$ 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 , $\text{map } f \text{ } c$ 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 $\text{map } p \text{ } c$ is reciprocated, and is as a consequence flipped over the real line⁴. 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 $P(s) = 1 - G(s) = 1 - \sum_x \prod (s - p_x)$, 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 $c = a^2 - b^2$

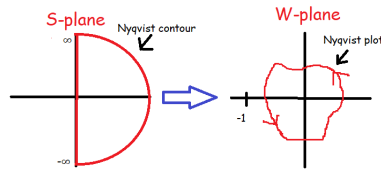


Figure 18: Placing a contour that surrounds all of the right-hand side of the complex plane, and mapping with a (transfer) function gives us the so-called Nyquist plot. If this surrounds the origin, the system the transfer function describes is stable.

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 $C 0 (-\text{Infinity})$ to $C 0 (\text{Infinity})$ and looking only at whether $\text{map } g \ c$ 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

```
1  --      null-val zip-function   left   right   zipped
2  zipWithL :: a -> (a -> a -> b) -> [a] -> [a] -> [b]
3  zipWithL i f (l:left) (r:right) = f l r : zipWithL i f left right
4  zipWithL _ _ []           []           = []
5  zipWithL i f []           (r:right) = f i r : zipWithL i f []   right
6  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]`.

```
1  maxfreq :: (Ord a, Num a) => Signal a -> a
2  maxfreq (Sin _ f _) = f
3  maxfreq (Scale _ a) = maxfreq a
4  maxfreq (Deriv _ a) = maxfreq a
5  maxfreq (Integ _ a) = maxfreq a
6  maxfreq (Sum a b) | af < bf = bf
7                    | otherwise = af where
8                    af = maxfreq a
9                    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.

```
1  {- DSL for complex numbers. Adapted with minor modifications from
2     https://github.com/DSLsofMath/BScProj/ -}
3  module ComplexNumbers where
4  import GHC.Real
5
6  type R = Double
7
8  -- Komplexa tal kan ses som ett par av reella värden.
9
10 data Complex = Complex (R, R)
11   deriving Eq
12
13 -- Där första komponenten representerar realdelen och den andra komponenten
14 -- imaginärdelen.
15
16 realPart :: Complex -> R
17 realPart (Complex (re, im)) = re
18
19 imPart :: Complex -> R
20 imPart (Complex (re, im)) = im
21
22 conjugate :: Complex -> Complex
23 conjugate z = Complex ((realPart z), (negate (imPart z)))
24
```



```

25 {-
26     Argumentet av ett komplext tal.
27     Bökigt värre på grund av tråkiga kvadranter och bös,
28     men genom att använda funktionen atan2 så blir det mycket snyggare!
29 -}
30
31 arg :: Complex -> R
32 arg z = atan2 (imPart z) (realPart z)
33
34 {-
35     Utan atan2 hade vi fått skriva så här:
36
37     arg :: Complex -> R
38     arg z | (imPart z) < 0 && (realPart z) < 0 = atan (imPart z / realPart z) - pi
39           | (realPart z) < 0                  = atan (imPart z / realPart z) + pi
40           | otherwise                         = atan (imPart z / realPart z)
41
42 -}
43
44 scale :: R -> Complex -> Complex
45 scale a z = Complex ((a * realPart z), (a * imPart z))
46
47 {-
48     j är det komplexa talet med realdelen 0 och imaginärdelen 1
49     Många matematiktexter kallar detta talet också för `i`
50 -}
51
52 j :: Complex
53 j = Complex (0, 1)
54
55 -- TODO: Försök få read . printComplex = id genom att göra printComplex
56 -- enklare
57 -- TODO: "name and reuse" something like showIm im = show im ++ "j" to make it
58 -- easy to change the spacing (or change "j" to "i", or the type from String
59 -- to ShowS) in only one place
60
61 instance Show Complex where
62     show = printComplex
63
64 printComplex :: Complex -> String
65 printComplex z
66 | r == 0    = imj
67 | im == 0   = show r
68 | im < 0    = show r ++ "-" ++ show (abs im) ++ "j"
69 | otherwise = show r ++ "+" ++ imj
70   where im = imPart z
71         r  = realPart z
72         imj
73         | im == 0 = "0"
74         | otherwise = show im ++ "j"

```

```

75
76 -- / Skapar ett komplext tal utifrån en vinkel.
77 euler :: R -> Complex
78 euler phi = Complex ((cos phi), (sin phi))
79
80 instance Num Complex where
81 -- Plus är förhållandevist trivialt
82   z + w = Complex ((realPart z + realPart w), (imPart z + imPart w))
83
84 -- Multiplikationen följer ifrån att vi multiplicerar komponenterna
85 -- parvis med varandra (i likhet med (a+b) * (c+d))
86   z * w = Complex ((realZ*realW - imZ*imW), (realZ*imW + realW*imZ))
87   where realZ = realPart z
88         realW = realPart w
89         imZ   = imPart z
90         imW   = imPart w
91
92 -- Negationen av ett komplext tal är ett komplext tal där båda komponenter är negerade
93
94   negate z = Complex ((negate $ realPart z), (negate $ imPart z))
95
96 -- Den naturliga generaliseringen av signum från reella till komplexa tal:
97
98   signum z = z / abs z
99
100 -- Absolutbeloppet av ett komplext tal är pythagoras sats på dess komponenter.
101
102   abs z = Complex (hyp, 0)
103   where hyp = sqrt (r*r + im*im)
104         r    = realPart z
105         im    = imPart z
106
107 -- Heltal är bara komplexa tal utan imaginärdel
108
109   fromInteger n = Complex (fromInteger n, 0)
110
111 instance Fractional Complex where
112 -- Division utförs genom att man förlänger hela bråket med nämnarens konjugat
113   z / w = Complex (realPart zw' / realPart ww', imPart zw' / realPart ww')
114   where zw' = z * (conjugate w)
115         ww' = w * (conjugate w)
116
117 -- En kvot av två heltal är helt reell och därför kommer imaginärdelen vara 0
118   fromRational z = Complex (re, 0)
119   where re = fromInteger (numerator z) / fromInteger (denominator z)
120
121 instance Floating Complex where
122 -- Det komplexa talet pi är ett tal med realdelen pi och imaginärdelen 0
123   pi = Complex (pi, 0)
124
125 {-

```

```

126 Potenslagarna ger att  $e^{(a+bj)} \Leftrightarrow e^a * e^{bj}$  och
127 Eulers formel ger att  $e^{bj} \Leftrightarrow \cos b + j \sin b$ 
128 -}
129 exp z = scale (exp (realPart z)) (euler (imPart z))
130
131 {-
132 Eulers formel ger att man kan skriva om  $\sin x$  som  $(e^{jx} - e^{-jx})/2j$ 
133 Eftersom vi definierat exponentialfunktionen för komplexa tal kan vi använda
134 eulers formel som vår sinus implementation för komplexa tal
135 -}
136 sin z = (exp (j*z) - exp (-(j*z)))/(scale 2 j)
137
138 -- Eulers formel ger att man kan skriva om  $\cos x$  som  $(e^{jx} + e^{-jx})/2$ 
139 cos z = (exp (j*z) + exp (-(j*z)))/2
140
141 cosh z = Complex (cosh (realPart z) * cos (imPart z),
142                  sinh (realPart z) * sin (imPart z))
143
144 sinh z = Complex (sin (realPart z) * cosh (imPart z),
145                  cos (realPart z) * sinh (imPart z))
146
147 {-
148 Eulers formel ger att  $z = re^{(j*\phi)}$  och eftersom log och exponentialfunktionen
149 är varandras inverser och logaritmlagarna ger ex.  $\log(a*b) \Leftrightarrow \log a + \log b$ .
150 Därför kan vi skriva  $\log z = \log r + \log(e^{(j*\phi)}) = \log r + j*\phi$ 
151 -}
152 log z = Complex (logR (abs z), arg z)
153   where logR = log . realPart
154
155 {-
156 Funktioner vi troligen / förhoppningsvis inte kommer behöva och som vi därför lämnar
157 odefinierade än så länge.
158 -}
159 atanh = undefined
160 atan  = undefined
161 asinh = undefined
162 asin  = undefined
163 acosh = undefined
164 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.

```

1 data Expression a where
2   Const      :: a -> Expression a
3   Id         :: Expression a
4   Impulse    :: Expression a
5   Pi         :: Expression a
6   (:+ :)     :: Expression a -> Expression a -> Expression a
7   (:* :)     :: Expression a -> Expression a -> Expression a
8   (:~ :)     :: Expression a -> Expression a -> Expression a

```

```

9  (./:)      :: Expression a -> Expression a -> Expression a
10 Conv      :: Expression a -> Expression a -> Expression a
11 Shift     :: a -> Expression a
12 Negate    :: Expression a
13 Exp       :: Expression a
14 Sin       :: Expression a
15 Cos       :: Expression a
16 Tan       :: Expression a
17 Asin      :: Expression a
18 Acos      :: Expression a
19 Atan      :: Expression a
20 Integral  :: Expression a -> a -> Expression a -- function and initial value, i.e.  $\int_{y_0}^t f(x)$ 
21 Derivative :: Expression a -> a -> Expression a -- function and initial value, i.e.  $f'$  and  $f(0)$ 
22 deriving (Eq, Show)

```

Code 6: Expression datatype.

```

1  shift :: Num a => a -> (a -> a) -> (a -> a)
2  shift tau f = \t -> f (t-tau)
3
4  eval :: (Num a, Floating a, Eq a) => Expression a -> a -> a
5  eval (Const c)      = \t -> c
6  eval Id             = id
7  eval Impulse        = undefined -- TODO
8  eval Pi             = eval (Const pi)
9  eval (e1 :+: e2)     = eval e1 + eval e2
10 eval (e1 **: e2)     = eval e1 * eval e2
11 eval (e1 :-: e2)     = eval e1 - eval e2
12 eval (e1 ./: e2)     = eval e1 / eval e2
13 eval (DefInteg e1 e2 e3) = undefined -- TODO
14 eval (Conv          e1 e2) = undefined -- TODO
15 eval (Shift tau e)    = shift tau (eval e) -- shifts e by tau, i.e. begins after tau seconds
16 eval (Negate e)       = negate (eval e)
17 eval (Exp e)          = exp      (eval e)
18 eval (Sin e)          = sin      (eval e)
19 eval (Cos e)          = cos      (eval e)
20 eval (Tan e)          = tan      (eval e)
21 eval (Asin e)         = asin     (eval e)
22 eval (Acos e)         = acos     (eval e)
23 eval (Atan e)         = atan     (eval e)
24 eval (Integral e e0)  = undefined -- TODO
25 eval (Derivative e e0) = undefined -- TODO

```

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

```

1  -- rules
2  laplace :: Expression Complex -> Expression Complex -- TODO: maybe instead do Time->Freq
3  laplace (e1 :+: e2) = laplace e1 :+: laplace e2 -- Superposition
4  laplace (e1 :-: e2) = laplace e1 :+: Negate (laplace e2) -- special case of superposition

```

```

5 laplace (Const a :: e) = Const a :: laplace e -- scaling
6 laplace (Negate e) = Negate (laplace e) -- can be seen as scaling with -1 as well
7 laplace (Derivative e e0) = Id :: laplace e :: (Const e0) -- L {f'} = \s -> s * L {f} - f(0)
8 laplace (Integral e e0) = laplace e :: Id -- L {int_0^t f(x) dx} = s -> L {f} / s
9 laplace (Shift tau e) = Exp (Id :: Const tau) :: laplace e -- TODO: double check signs
10 -- TODO: double check signs on next two
11 laplace (Exp (Const tau :: Id) :: e) = Shift (negate tau) (laplace e) -- semantically same as next
12 laplace (Exp (Negate (Const tau :: Id)) :: e) = Shift tau (laplace e)
13 laplace (Conv e1 e2) = laplace e1 :: laplace e2 -- convolution turns into multiplication
14 -- transforms
15 laplace (Impulse) = Const 1 -- L {delta(t)} = \s -> 1
16 laplace (Const a) = Const a :: Id -- L {a} = \s -> a/s; a = 1 => L {1} = 1/s
17 laplace (Id) = Const 1 :: (Id :: Id) -- L {t} = \s -> 1/(s^2)
18 laplace (Exp (Negate (Const a :: Id))) = Shift (negate a) (Const 1 :: Id) -- TODO: Check signs
19 laplace (Exp (Const a :: Id)) = Shift a (Const 1 :: Id) -- TODO: Check signs
20 laplace (Const 1 :: Exp (Negate (Const a :: Id))) = Const a :: (Id :: Shift a Id) -- TODO: Doubl
21
22 laplace (Exp Id) = laplace (Exp (Const 1 :: Id))
23 -- ... some more special cases, not sure we care?
24 -- TODO: Doublecheck signs on the next two
25 laplace (Exp (Negate (Const a :: Id)) :: Sin (Const omega :: Id)) =
26   Const omega :: (Shift a (Id :: Id) :: (Const (omega*omega)))
27 laplace (Exp (Negate (Const a :: Id)) :: Cos (Const omega :: Id)) =
28   (Shift a Id) :: (Shift a (Id :: Id) :: (Const (omega*omega)))

```

Code 8: Definition of `laplace`, the Laplace transform of an `Expression`.

