Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Introduction / About this book

[src: Introduction/About.lhs] Previous: Table of contents Table of contents Next: So what's a DSL?

About this book

You've arrived at **Learn You a Physics for Great Good**, the #1 stop for all your beginner-to-intermediate needs of learning **Physics** through the use of **Domain Specific Languages** (DSLs).

This book was written as a BSc thesis project at Chalmers University of Technology as an offshoot of a bachelor's level elective course *Domain Specific Languages of Mathematics*. The goal of the the project and the reason for the existence of this book, is to help (primarily CS) students learn physics better. We think that the use of domain specific languages to teach the subject will help set these students in the right mindset to learn physics efficiently and properly. An observed problem has been that students base their mental models completely or partially on intuition and incorrect assumptions, instead of on definitions and proved theorems where validity is ensured. This may be a bad habit stemming from the way math and physics is taught in earlier stages of the education, and we think that DSLs will inherently force students into the right mindset for learning this subject well! Further, many textbooks on physics are incredibly thick and boring, so we've decided to emulate the great and good Learn You a Haskell for Great Good in order to provide som comic relief in between all the dramatic definitions.

In this book, we will study physics through the lens of DSLs. We will need to pay close attention to definitions, throughly ponder any non-trivial syntax, and verify (via tests or

proofs) that the things we do are actually correct. The areas covered include such things as: dimensional analysis, vectors, calculus, and more!

The book is aimed at you who have some knowledge of Haskell. If you know what a `class` and an `instance` is, you're probably good to go! Even if you don't we're sure you could pick it up as we go. We believe in you!

If you wonder about the weird code blocks at the start of many chapters,

```
module Introduction.About where
```

they are there to declare the chapters as Haskell modules. All chapters are written in *Literate Haskell*, and can be used with GHC/GHCi directly as source code. Therefore, you may choose to read each chapter as documented source code, rather than text with examples.

[src: Introduction/About.lhs] Previous: Table of contents Table of contents Next: So what's a DSL?

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Introduction / So what's a DSL?

[src: Introduction/WhatIsADsl.lhs] Previous: About this book Table of contents Next: Getting started

```
module Introduction.WhatIsADsl where
```

So what's a DSL?

In general, a domain specific language is simply a computer language for a specific domain. It's NOT a synonym for jargon! DSLs can be specialized for markup, like *HTML*; for modelling, like *EBNF*; for shell scripting, like *Bash*; and more.

The languages we will construct will mostly concern modelling of physics and mathematics. We will create data structures in Haskell to represent the same physics calculations that we write on paper, in such a way that we can write functions to, for example, analyze the expressions for validity.

Look, we'll demonstrate. Let's say we want to model a language that is a subset to the common algebra we're all familiar with. Our language will consist expressions of a single variable and addition. For example, the following three expressions are all valid in such a language:

$$x + x$$

$$x$$

$$(x + x) + (x + x)$$

When implementing a DSL, we typically start with modelling the syntax. Let's first declare a data type for the language

```
data Expr
```

Then we interpret the textual description of the language. "Our language will consist of expressions of a single variable and addition". Ok, so an expression can be one of two things then: a single variable

```
  = X
```

or two expressions added together.

```
  | Add Expr Expr
```

And that's it, kind of! However, a DSL without any associated functions for validation, symbolic manipulation, evaluation, or somesuch, is really no DSL at all! We must DO something with it, or there is no point!

One thing we can do with expressions such as these, is compare whether two of them are equal. Even without using any numbers, we can test this by simply counting the xs! If both expressions contain the same number of xs, they will be equal!

```
eq :: Expr -> Expr -> Bool
eq e1 e2 = count e1 == count e2
  where count X           = 1
        count (Add e1 e2) = count e1 + count e2
```

We can now test whether our expressions are equal.

```
ghci> eq (Add (Add X X) X) (Add X (Add X X))
True
```

And NOW that's it (if we want to stop here)! This is a completely valid (but boring) DSL. We've modelled the syntax, and added a function that operates symbolically on our language. This is a very small and simple DSL, and you've likely done something similar before without even realizing you were constructing a DSL. It can really be that simple

In other DSLs we may also look at evaluation and the semantics of a language, i.e. what is the actual type of the result when we *evaluate* or "compute" our expressions.

Throughout this book we'll construct and use DSLs in many different ways. Different parts of the physics we look at will require different DSL treatments, basically. There is no *one* model to use for all DSLs.

[src: Introduction/WhatIsADsl.lhs] Previous: About this book Table of contents Next: Getting started

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Introduction / Getting started

[src: Introduction/GettingStarted.lhs]Previous: So what's a DSL?Table of contentsNext: Introduction

```
module Introduction.GettingStarted where
```

What you need to dive in

To just follow along and implement the same stuff we do, a single document loaded into GHCi will do. For this, you just need to install the Haskell Platform.

If you want to automatically install any required dependencies, like *Hatlab* which will be used later on, install Stack. Stack is a build system that will automatically get dependencies and build the project for you. If you want to build the code the same way we do, this is what you'll need. With Stack installed and our repository cloned, enter the `Physics` directory and type `stack build`. Stack should now download and compile all necessary dependencies, such that they are avaiable when you load a module in GHCi.

If you need to sharpen up your skills

If you need to freshen up on your Haskell, Learn You a Haskell for Great Good is a great book that covers all the important stuff in a humorous manner that really holds your attention.

If you still don't really get what DSLs are all about, try reading the "notes" (almost a full book really) for the course on DSLs of math at Chalmers. If you're studying at Chalmers, it is even better to actually take the course!

If there's some part of the material that you still think is unclear (or maybe even wrong? Blasphemy!), please open an issue on the github repo!. We'll look into it!

[src: Introduction/GettingStarted.lhs]Previous: So what's a DSL?Table of contentsNext: Introduction

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Calculus / Introduction

[src: Calculus/Intro.lhs] Previous: Getting started Table of contents Next: Function expressions

```
module Calculus.Intro where
```

What is calculus?

Plain equations where no values change over time are all fine and well. The importance of being able to solve basic problems like “If Jenny has 22 apples, and Richard has 18 apples: how many apples do they have together?” cannot be understated, but they’re not especially fun!

“An unstoppable car has an constant velocity of 141.622272 km/h. How many kilometers has it driven after a day?”. To solve more interesting problems like this, we need calculus.

Calculus is the study of stuff that continuously change over time (or some other continuous variable). For example, a distance that changes over time is equivalent to a velocity. You can have rates of changes with respect to other units, like length, as well, but those are not as common.

There are two major branches of calculus, differential calculus and integral calculus. Differential calculus is all about those rates of changes and graph slopes. Differences, differentials, derivatives, and the like. Integral calculus, on the other hand, is all about accumulation and areas. Sums, integrals, and such.

In this chapter we'll expore the syntax of diffences, the problem with differentials, symbolic differentiation, and numeric and symbolic integration.

[src: Calculus/Intro.lhs] Previous: Getting started Table of contents Next: Function expressions

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Calculus / Function expressions

[src: Calculus/FunExpr.lhs] Previous: Introduction Table of contents Next: Differential calculus

```haskell
module Calculus.FunExpr (FunExpr (..), RealNum, RealFunc) where

import Test.QuickCheck
```

Semantics, syntax, and stuff

What is a value in calculus? What kind of values do functions in calculus operate on and produce?

Let's look at derivatives to get an idea of what the semantic value of calculus is.

$$\frac{dx^2}{dx} = 2x$$

$$\frac{df(x)}{dx} = f'(x)$$

Hmm, these examples left me more confused than before. The differentiation function seems to take an expression as an argument, and return the differentiated expression, with regards to a variable. But what is an expression represented as a semantic value? It's not a number yet, the variable in the body needs to be substituted first in order for the expression to be computable. Is it some kind of function then? Well, yes it is! If we reinterpret the differentiation expressions above, it makes more sense.

$$\frac{dx^2}{dx} = 2x$$

can be written as

$$D(x^2) = 2x \text{ with regards to } x$$

which is really equivalent to

$$D(x \mapsto x^2) = x \mapsto 2x$$

or with more descriptive function names

$$D(square) = double$$

.

So the type of unary real functions seems like a great fit for a semantic value for calculus, and it is! Great! But... how do we represent a real number in Haskell? There is no `Real` type to use. Well, for the sake of simplicity we can just say that a real number is basically the same as a `Double`, and really it is (basically). The problem with `Double` is that it's of finite precision, so rounding errors may occur. We'll have to keep that in mind when doing calculations!

```
type RealNum = Double
type RealFunc = RealNum -> RealNum
```

Now, to the syntax. We've concluded that real functions are really what calculus is all about, so let's model them. We create a data type `FunExpr` that will represent symbolic expressions of functions in our language.

`FunExpr` is very expressive!

```
data FunExpr
```

First of all, there are the elementary functions. We can't have them all, that would get too repetitive to implement, but we'll put in all the fun ones.

```
    = Exp | Log | Sin | Cos | Asin | Acos
```

Then, there are the arithmetic operators. "But wait", you say, "Aren't arithmetic operators used to combine expressions, not functions?". We hear you friend, but we will do it anyways. We could make a `Lambda` constructor for "VAR $\mapsto$ EXPR" expressions and define the arithmetic operators for the expression type, but this would make our language much more complicated! Instead, we'll restrain ourselves to single variable expressions, which can be represented as compositions of unary functions, and define the arithmeric operators for the functions instead.

$$f\ OP_{r \to r}\ g = x \mapsto (f(x)\ OP_r\ g(x))$$

```
  | FunExpr :+ FunExpr
  | FunExpr :- FunExpr
  | FunExpr :* FunExpr
  | FunExpr :/ FunExpr
  | FunExpr :^ FunExpr
```

And then there's that single variable. As everything is a function expression, the function that best represents "just a variable" would be $x \mapsto x$, which is the same as the id function.

```
  | Id
```

In a similar vein, the constant function. $const(c) = x \mapsto c$

```
  | Const RealNum
```

Then there's function composition. If you didn't already know it, it's defined as

$$f \circ g = x \mapsto f(g(x))$$

```
  | FunExpr :. FunExpr
```

Finally, the real heroes: the functions of difference, differentiation, and integration! They will be well explored later. But for now, we just define the syntax for them as

```
  | Delta RealNum FunExpr
  | D FunExpr
  | I FunExpr
```

Even more finally, we add a `deriving` modifier to automatically allow for syntactic equality tests between `FunExpr`s.

```
  deriving Eq
```

Nice! This syntax tree will allow us to do symbolically (at the syntax level) what we otherwise would have to do numerically (at the semantics level).

Before we move on, we just have to fix one thing: the operator precedence! If we don't do anything about it, this will happen

```
ghci> Id :+ Id :* Id == (Id :+ Id) :* Id
True
```

Now this is obviously wrong. *Plus* doesn't come before *times*, unless I accidentaly switched timelines in my sleep. To fix this, we have to fix the fixity. `infixl` allows us to make an operator left-associative, and set the precedence.

```
-- Medium precedence
infixl 6 :+
infixl 6 :-
-- High precedence
infixl 7 :*
infixl 7 :/
-- Higher precedence
infixl 8 :^
-- Can't go higher than this precedence
infixl 9 :.
```

A structure with class

Now that we've defined the basic structure of our language, we can instantiate some useful classes. There are three in particular we care for: `Show`, `Num`, and `Arbitrary`.

Try modifying `FunExpr` to derive `Show`, so that our expressions can be printed.

```
  deriving (Eq, Show)
```

Consider now how GHCi prints out a function expression we create

```
ghci> carAccel = Const 20
ghci> carSpeed = Const 50 :+ carAccel :* Id
ghci> carPosition = Const 10 :+ carSpeed :* Id
ghci> carPosition
Const 10.0 :+ (Const 50.0 :+ Const 20.0 :* Id) :* Id
```

Well that's borderline unreadable. Further, the grokability of a printed expression is very inversely proportional to the size/complexity of the expression, as I'm sure you can imagine.

So if the derived `Show` is bad, we'll just have to make our own `Show`!

```
showFe :: FunExpr -> String
showFe Exp = "exp"
showFe Log = "log"
showFe Sin = "sin"
showFe Cos = "cos"
showFe Asin = "asin"
showFe Acos = "acos"
showFe (f :+ g) = "(" ++ showFe f ++ " + " ++ showFe g ++ ")"
showFe (f :- g) = "(" ++ showFe f ++ " - " ++ showFe g ++ ")"
showFe (f :* g) = "(" ++ showFe f ++ " * " ++ showFe g ++ ")"
showFe (f :/ g) = "(" ++ showFe f ++ " / " ++ showFe g ++ ")"
showFe (f :^ g) = "(" ++ showFe f ++ "^" ++ showFe g ++ ")"
showFe Id = "id"
showFe (Const x) = showReal x
showFe (f :. g) = "(" ++ showFe f ++ " . " ++ showFe g ++ ")"
showFe (Delta h f) = "(delta_" ++ showReal h ++ " " ++ showFe f ++ ")"
showFe (D f) = "(D " ++ showFe f ++ ")"
showFe (I f) = "(I " ++ showFe f ++ ")"

showReal x = if isInt x then show (round x) else show x
  where isInt x = x == fromInteger (round x)


instance Show FunExpr where
  show = showFe
```

Not much to explain here. It's just one way to print our syntax tree in a more readable way. What's interesting is how we can now print our expressions in a much more human friendly way!

```
ghci> carPosition
(10 + ((50 + (20 * id)) * id))
```

Still a bit noisy with all the parentheses, but much better!

Another class we can instantiate for our `FunExpr` is `Num`. This class allows us to make use of the native Haskell functions and operators for numeric operations, instead of writing our own constructors. This sometimes improves the readability of the code.

```
instance Num FunExpr where
  negate e    = Const 0 :- e
  (+)         = (:+)
  (*)         = (:*)
  fromInteger = Const . fromInteger
  abs         = undefined
  signum      = undefined
```

Third but not least, we'll instantiate `Arbitrary`. This class is associated with the testing library *QuickCheck*, and describes how to generate arbitrary values of a type for use when testing logical properties with `quickCheck`. For example, a property function could be formulated that states that the `:*` constructor of `FunExpr` is associative.

The implementation itself is not very interesting. We generate a function expression that tends to contain mostly elementary functions, arithmetic operations, and a generous dose of constants; with a light sprinkle of differences. We won't include derivatives as all elementary functions have elementary derivatives anyways, and integrals may cause cause approximation errors if we have to numerically compute them at evaluation.

```
instance Arbitrary FunExpr where
  arbitrary =
```

`frequency` "chooses one of the given generators, with a weighted random distribution". By assigning probabilities of generating certain functions more often than others, we can restrain the growth of the generated expressions in complexity.

```
      frequency
        [ (10, genElementary)
        , (10, genBinaryOperation)
        , (10, return Id)
        , (20, fmap Const arbitrary)
        , (10, genBinaryApp (:.))
        , (5 , genBinaryApp Delta) ]
    where genElementary = elements [Exp, Log, Sin, Cos, Asin, Acos]
```

```
genBinaryApp op = fmap (\(f, g) -> f `op` g) arbitrary
genBinaryOperation =     elements [(:+), (:-), (:*), (:/), (:^)]
                     >>= genBinaryApp
```

[src: Calculus/FunExpr.lhs] Previous: Introduction Table of contents Next: Differential calculus

Learn You a Physics for Great Good!

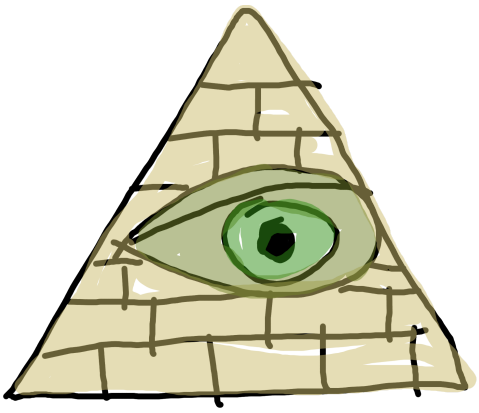
>>> WORK IN PROGRESS <<<

Calculus / Differential calculus

[src: Calculus/DifferentialCalc.lhs] Previous: Function expressions | Table of contents | Next: Integral calculus

```
module Calculus.DifferentialCalc (deriveApprox, derive, simplify) where

import Calculus.FunExpr
```

Deep, dark, differences



A *difference* is, in it's essence, quite simply the result of applying the operation of subtraction to two real number terms.

$$minuend - subtrahend = difference$$

Nice, great job, we're done here, let's move on.

...

Just kidding, of course there's more to it than that.

In calculus, the term *difference* carries more meaning than usual. More than just a subtraction of arbitrary values, differences lie at the heart of calculations regarding rate of change, both average and instantaneous.

Quotients of differences of functions of the same time describe the average rate of change over the time period. For example, an average velocity can be described as the difference quotient of difference in position and difference in time.

$$v_{avg} = \frac{p_2 - p_1}{t_2 - t_1}$$

where p_n is the position at time t_n.

In the context of calculus, we use a special syntax for differences: the delta operator! With this, the previous definition can be rewritten as

$$v_{avg} = \frac{p_2 - p_1}{t_2 - t_1} = \frac{\Delta p}{\Delta t}$$

.

This is the informal definition of the delta operator used in *University Physics*:

$$\Delta x = x_2 - x_1$$

Ok, so it's a difference. But what does x_2 and x_1 mean, and what do they come from? x_2 and x_1 are not explicitly bound anywhere, but it seems reasonable to assume that $x_i \in \mathbb{R}$ or equivalently, that x is a function with a subscript index as an argument, that returns a $\mathbb{R}$.

Further, the indices $1, 2$ should not be thought of as specific constants, but rather arbitrary real number variables identified by these integers. Lets call them a, b instead, to make it clear that they are not constants.

$$\Delta x = x_b - x_a$$

Now a, b are implicitly bound. We make the binding explicit.

$$(\Delta x)(a, b) = x_b - x_a$$

We compare this to the more formal definition of *forward difference* on wikipedia:

$$\Delta_h[f](x) = f(x + h) - f(x)$$

The parameter bindings are a bit all over the place here. To more easily compare to our definition, let's rename x to a and f to x, and change the parameter declaration syntax:

$$(\Delta x)(h)(a) = x(a+h) - x(a)$$

This is almost identical to the definition we arrived at earlier, with the exception of expressing b as $a + h$. Why is this? Well, in calculus we mainly use differences to express two things, as mentioned previously. Average rate of change and instantaneous rate of change.

Average rate of change is best described as the difference quotient of the difference in y-axis value over an interval of x, and the difference in x-axis value over the same interval.

$$\frac{y(x_b) - y(x_a)}{x_b - x_a}$$

.

In this case, the x's can be at arbitrary points on the axis, as long as $b > a$. Therefore, the definition of difference as $(\Delta x)(a, b) = x_b - x_a$ seems a good fit. Applied to average velocity, our difference quotient

$$v_{avg} = \frac{\Delta p}{\Delta t}$$

will expand to

$$v_{avg}(t_2, t_1) = \frac{(\Delta p)(t_2, t_1)}{(\Delta t)(t_2, t_1)}$$

for $t_2 > t_1$.

Instantaneous rate of change is more complicated. At its heart, it too is defined in terms of differences. However, we are no longer looking at the average change over an interval delimited by two points, but rather the instantaneous change in a single point.

Of course, you can't have a difference with only one point. You need two points to look at how the function value changes between them. But what if we make the second point reeeeeeeeealy close to the first? That's basically the same as the difference in a single point, for all intents and purposes. And so, for instantaneous rate of change, the definition of difference as $(\Delta x)(h)(a) = x(a+h) - x(a)$ will make more sense, for very small h's. Applied to instantaneous velocity, our difference quotient

$$v_{inst} = \frac{\Delta p}{\Delta t}$$

for very small Δt, will expand to

$$v_{inst}(h, x) = \frac{(\Delta p)(h, x)}{(\Delta t)(h, x)}$$

for very small h.

As h gets closer to 0, our approximation of instantaneous rate of change gets better.

And so, we have a method of computing average rate of change, and instantaneous rate of change (numerically approximatively). In Haskell, we can make shallow embeddings for differences in the context of rate of change as velocity.

Average velocity is simply

```
v_avg pos t2 t1 = (pos(t2) - pos(t1)) / (t2 - t1)
```

which can be used as

```
ghci> v_avg (\x -> 5*x) 10 0
5.0
```

And instantaneous velocity is

```
v_inst pos h t = (pos(t + h) - pos(t)) / ((t + h) - t)
```

which can be used as

```
ghci> carSpeed t = v_inst (\x -> x + sin(x)) 0.00001 t
ghci> carSpeedAtPointsInTime = map carSpeed [0, 1, 2, 3]
ghci> carSpeedAtPointsInTime
[1.9999999999833333,1.5402980985023251,0.5838486169602084,1.0006797790330592e-2]
```

We'd also like to model one of the versions of the delta operator, finite difference, in our syntax tree. As the semantic value of our calculus language is the unary real function, the difference used for averages doesn't really fit in well, as it's a binary function (two arguments: t_2 and t_1). Instead, we'll use the version of delta used for instants, as it only takes a single point in time as an argument (assuming h is already given).

The constructor in our syntax tree is therefore

```
    | Delta RealNum FunExpr
```

where the first argument is h, and the second is the function.

Derivatives

The *derivative* of a function is, according to wikipedia, "the slope of the tangent line to the graph of [a] function at [a] point" and can be described as the "instantaneous rate of change", and *differentiation* is the method of finding a derivative for a function.

…

Wait, didn't we just look at instantaneous rates of changes in the previous section on differences? Well yes, and the difference quotient for a function at a point with a very small step h is indeed a good way to numerically approximate the derivative of a function. From what we found then, we can derive a general expression for instantaneous rate of change

$$\frac{(\Delta f)(h,x)}{(\Delta id)(h,x)} = \frac{f(x+h)-f(x)}{h}$$

for very small h.

But what if we don't want just a numerical approximation, but THE derivative of a function at any arbitrary point? What if we make h not just very small, but *infinitley* small?

Introducing *infinitesimals*! From the wikipedia entry on *Leibniz's notation*

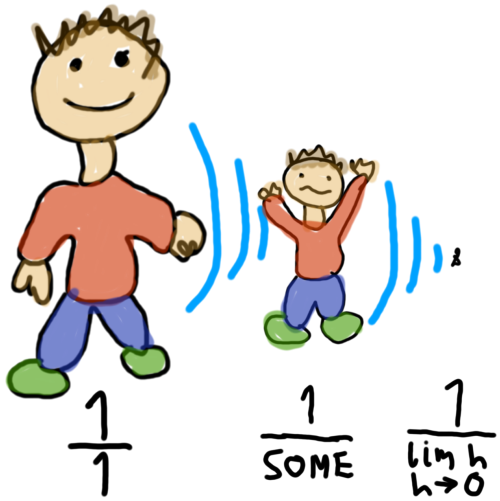
> In calculus, Leibniz's notation, named in honor of the 17th-century German philosopher and mathematician Gottfried Wilhelm Leibniz, uses the symbols dx and dy to represent infinitely small (or infinitesimal) increments of x and y, respectively, just as Δx and Δy represent finite increments of x and y, respectively.

So there's a special syntax for differences where the step h is infinitely small, and it's called Leibniz's notation. We could naively interpret the above quote in mathematical terms as

$$dx = lim_{\Delta x \to 0} \Delta x$$

such that

$$\forall y(x).\, D(y) = \frac{dy}{dx} = \frac{lim_{\Delta y \to 0} \Delta y}{lim_{\Delta x \to 0} \Delta x}$$



where D is the function of differentiation. However, this doesn't quite make sense as $lim_{x \to 0} x = 0$ and we'd be dividing by 0. The wording here is important: infinitesimals aren't 0, but *infinitely* small! The result is that we can't define infinitesimals in terms of limits, but have to treat them as an orthogonal concept.

Except for this minor road bump, this definition of derivatives is very appealing, as it suggests a very simple and intuitive transition from finite differences to infinitesimal differentials. Also, it suggests the possibility of manipulating the infinitesimals of the derivative algebraically, which might be very useful. However, this concept is generally considered too imprecise to be used as the foundation of calculus.

A later section on the same wikipedia entry elaborates a bit:

> Leibniz's concept of infinitesimals, long considered to be too imprecise to be used as a foundation of calculus, was eventually replaced by rigorous concepts developed by Weierstrass and others. Consequently, Leibniz's quotient notation was re-interpreted to stand for the limit of the modern definition. However, in many instances, the symbol did seem to act as an actual quotient would and its usefulness kept it popular even in the face of several competing notations.

What is then the "right" way to do derivatives? As luck would have it, not much differently than Leibniz's suggested and how we interpreted it above! The intuitive idea can be turned into a precise definition by defining the derivative to be the limit of difference quotients of real numbers. Again, from wikipedia - Leibniz's notation:

> In its modern interpretation, the expression dy/dx should not be read as the division of two quantities dx and dy (as Leibniz had envisioned it); rather, the

> whole expression should be seen as a single symbol that is shorthand for
>
> $$D(y) = lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$

which, when y is a function of x, and x is the id function for real numbers (which it is in the case of time), is:

$$\begin{aligned}
D(y) &= lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \\
&= a \mapsto lim_{\Delta x \to 0} \frac{(\Delta y)(\Delta x, a)}{\Delta x} \\
&= a \mapsto lim_{h \to 0} \frac{y(a + (\Delta x)(h, a)) - y(a)}{(\Delta x)(h, a)} \\
&= a \mapsto lim_{h \to 0} \frac{y(a + ((a + h) - a)) - y(a)}{(a + h) - a} \\
&= a \mapsto lim_{h \to 0} \frac{y(a + h) - y(a)}{h}
\end{aligned}$$

There, the definition of derivatives! Not too complicated, was it? We can write a simple numerically approximative according to the definition like

```
deriveApprox f h x = (f(x + h) - f(x)) / h
```

Only when h is infinitely small is `deriveApprox` fully accurate. However, as we can't really represent an infinitely small number in finite memory, the result will only be approximative, and the approximation will (in most cases) get better as h gets smaller. For example, let's calculate the slope of $f(x) = x^2$ at $x = 3$. As the slope of this parabola is calculated as $k = 2x$, we expect the result of `deriveApprox` to approach $k = 2x = 2 * 3 = 6$ as h gets smaller

```
ghci> deriveApprox (\x -> x^2) 5    3
11
ghci> deriveApprox (\x -> x^2) 1    3
7
ghci> deriveApprox (\x -> x^2) 0.5  3
6.5
ghci> deriveApprox (\x -> x^2) 0.1  3
6.100000000000012
ghci> deriveApprox (\x -> x^2) 0.01 3
6.009999999999849
```

Ok, that works, but not well. By making use of that fancy definition for derivatives that we derived earlier, we can now derive things symbolically, which implies provable 100% perfect accuracy, no numeric approximations!

We define a function to symbolically derive a function expression. `derive` takes a function expression, and returns the differentiated function expression.

```haskell
derive :: FunExpr -> FunExpr
```

Using only the definition of derivatives, we can derive the definitions of `derive` for the various constructors in our syntax tree.

For example, how do we derive `f :+ g`? Let's start by doing it mathematically.

$$
\begin{aligned}
D(f+g) =& a \mapsto lim_{h \to 0} \frac{(f+g)[a+h] - (f+g)[a]}{h} \\
& \{ \text{ Addition of functions } \} \\
=& a \mapsto lim_{h \to 0} \frac{f(a+h) + g(a+h) - (f(a) + g(a))}{h} \\
=& a \mapsto lim_{h \to 0} \frac{f(a+h) + g(a+h) - f(a) - g(a)}{h} \\
=& a \mapsto lim_{h \to 0} (\frac{f(a+h) - f(a)}{h} + \frac{g(a+h) - g(a)}{h}) \\
=& a \mapsto ((lim_{h \to 0} \frac{f(a+h) - f(a)}{h}) + (lim_{h \to 0} \frac{g(a+h) - g(a)}{h})) \\
=& (a \mapsto lim_{h \to 0} \frac{f(a+h) - f(a)}{h}) + (a \mapsto lim_{h \to 0} \frac{g(a+h) - g(a)}{h}) \\
& \{ \text{ Definition of derivative } \} \\
=& D(f) + D(g)
\end{aligned}
$$

Oh, it's just the sum of the derivatives of both functions! The Haskell implementation is then trivially

```haskell
derive (f :+ g) = derive f :+ derive g
```

Let's do one more, say, sin. We will make use of the trigonometric identity of sum-to-product

$$\sin\theta - \sin\varphi = 2\sin\left(\frac{\theta-\varphi}{2}\right)\cos\left(\frac{\theta+\varphi}{2}\right)$$

And the limit

$$\lim_{x\to 0}\frac{sinx}{x} = 1$$

the proof of which is left as an exercise to the reader

Exercise. Prove the limit $\lim_{x\to 0}\frac{sinx}{x} = 1$

▼ **Hint**

This limit can be proven using the unit circle and squeeze theorem

Then, the differentiation

$$\begin{aligned}
D(sin) =& a \mapsto lim_{h\to 0}\frac{sin(a+h)-sin(a)}{h} \\
& \{\text{ trig. sum-to-product }\} \\
=& a \mapsto lim_{h\to 0}\frac{2\sin\left(\frac{a+h-a}{2}\right)\cos\left(\frac{a+h+a}{2}\right)}{h} \\
=& a \mapsto lim_{h\to 0}\frac{2\sin\left(\frac{h}{2}\right)\cos\left(\frac{2a+h}{2}\right)}{h} \\
=& a \mapsto lim_{h\to 0}\frac{2\sin\left(\frac{h}{2}\right)\cos\left(\frac{2a+h}{2}\right)}{h} \\
=& a \mapsto lim_{h\to 0}\frac{\sin\left(\frac{h}{2}\right)}{\frac{h}{2}}\cos\left(\frac{2a+h}{2}\right) \\
& \{h \text{ approaches } 0\} \\
=& a \mapsto 1\cos\left(\frac{2a+0}{2}\right) \\
=& a \mapsto \cos(a) \\
=& \cos
\end{aligned}$$

Again, trivial definition in Haskell

```
derive Sin = Cos
```

Exercise. Derive the rest of the cases using the definition of the derivative

▼ Solution

```
derive Exp = Exp
derive Log = Const 1 :/ Id
derive Cos = Const 0 :- Sin
derive Asin = Const 1 :/ (Const 1 :- Id:^(Const 2)):^(Const 0.5)
derive Acos = Const 0 :- derive Asin
derive (f :- g) = derive f :- derive g
derive (f :* g) = derive f :* g :+ f :* derive g
derive (f :/ g) = (derive f :* g :- f :* derive g) :/ (g:^(Const 2))
derive (f :^ g) =    f:^(g :- Const 1)
                  :* (g :* derive f :+ f :* (Log :. f) :* derive g)
derive Id = Const 1
derive (Const _) = Const 0
```

For a function composition $f \circ g$ we have to handle the case where f is a constant function, as we may get division by zero if we naively apply the chain rule. We'll make use of the `simplify` function, which we'll define later, to reduce compositions of constant functions to just constant functions.

```
derive (f :. g) =
  case simplify f of
    Const a -> Const 0
    sf      -> (derive sf :. g) :* derive g
derive (Delta h f) = Delta h (derive f)
derive (D f) = derive (derive f)
```

Oh right, I almost forgot: Integrals. How are you supposed to know how to derive these bad boys when we haven't even covered them yet! We'll prove why this works later, but for now, just know that another name for integral is *Antiderivative*...

```
derive (I f) = f
```

Keep it simple

So we've got our differentiation function, great! Let's try it out by finding the derivative for a simple function, like $f(x) = sin(x) + x^2$, which should be $f'(x) = cos(x) + 2x$:

```
ghci> f = Sin :+ Id:^(Const 2)
ghci> derive f
(cos + ((id^(2 - 1)) * ((2 * 1) + ((id * (log . id)) * 0))))
```

Oh… that's not very readable. If we simplify it manually we get that the result is indeed as expected

```
(cos + ((id^(2 - 1)) * ((2 * 1) + ((id * (log . id)) * 0))))
cos + (id^1 * (2 + (id * (log . id) * 0)))
cos + (id * (2 + 0))
cos + 2*id
```

But still, we shouldn't have to do that manually! Let's have Mr. Computer help us out, by writing a function to simplify expressions.

We'll write a `simplify` function which will reduce an expression to a simpler, equivalent expression. Sounds good, only… what exactly does "simpler" mean? Is 10 simpler than $2 + 2 * 4$? Well, yes obviously, but there are other expressions where this is not the case. For example, polynomials have two standard forms. The first is the sum of terms, which is appropriate when you want to add or subtract polynomials. The other standard form is the product of irreducible factors, which is a good fit for when you want to divide polynomials.

So, our `simplify` function will not guarantee that every expression is reduced to *its most simple* form, but rather that many expressions will be reduced to *a simpler form*. As an exercise, you can implement more reduction rules to make expressions simpler to you. For example, the trigonometric identities like $sin(\theta + \frac{\pi}{2}) = cos(\theta)$.

```
simplify :: FunExpr -> FunExpr
```

The elementary functions by themselves are already as simple as can be, so we don't have to simplify those. When it comes to the arithmetic operations, most interesting is the cases of one operand being the identity element.

For addition and subtraction, it's 0

```
simplify (f :+ g) = case (simplify f, simplify g) of
    (Const 0, g') -> g'
    (f', Const 0) -> f'
    (Const a, Const b) -> Const (a + b)
    (f', g') | f' == g' -> simplify (Const 2 :* f')
    (Const a :* f', g') | f' == g' -> simplify (Const (a + 1) :* f')
    (f', Const a :* g') | f' == g' -> simplify (Const (a + 1) :* f')
    (Const a :* f', Const b :* g') | f' == g'
      -> simplify (Const (a + b) :* f')
    (f', g') -> f' :+ g'
simplify (f :- g) = case (simplify f, simplify g) of
    (f', Const 0 :- g') -> simplify (f' :+ g')
    (f', Const 0) -> f'
    (Const a, Const b) -> if a > b
                          then Const (a - b)
                          else Const 0 :- Const (b - a)
    (f', g') | f' == g' -> Const 0
    (Const a :* f', g') | f' == g' -> simplify (Const (a - 1) :* f')
    (f', Const a :* g') | f' == g'
      -> Const 0 :- simplify (Const (a - 1) :* f')
    (Const a :* f', Const b :* g') | f' == g'
      -> simplify ((Const a :- Const b) :* f')
    (f', g') -> f' :- g'
```

For multiplication and division, the identity element is 1, but the case of one operand being 0 is also interesting

```
simplify (f :* g) = case (simplify f, simplify g) of
    (Const 0, g') -> Const 0
    (f', Const 0) -> Const 0
    (Const 1, g') -> g'
    (f', Const 1) -> f'
    (Const a, Const b) -> Const (a * b)
    (f', Const c) -> Const c :* f'
    (f', g') | f' == g' -> f' :^ Const 2
    (Const a, Const b :* g') -> simplify (Const (a*b) :* g')
    (Const a :* f', g') -> simplify (f' :* (Const a :* g'))
    (fa, g' :/ fb) | fa == fb -> g'
    (f', g') -> f' :* g'
simplify (f :/ g) = case (simplify f, simplify g) of
    (Const 0, g') -> Const 0
    (f', Const 1) -> f'
    (f', g') | f' == g' -> Const 1
    (f', g') -> f' :/ g'
```

Exponentiation is not commutative, and further has no (two-sided) identity element. However, it does have an "asymmetric" identity element: the right identity 1!

```
simplify (f :^ g) = case (simplify f, simplify g) of
    (f', Const 1) -> f'
    (f', g') -> f' :^ g'
```

Exercises. Look up (or prove by yourself) more identities (of expressions, not identity elements) for exponentiation and implement them.

▼ Solution

> For example, there is the identity of negative exponents. For any integer n and nonzero b
>
> $$b^{-n} = \frac{1}{b^n}$$

Intuitively, the identity function is the identity element for function composition

```
simplify (f :. g) = case (simplify f, simplify g) of
    (Const a, _) -> Const a
    (Id, g')     -> g'
    (f', Id)     -> f'
    (f', g')     -> f' :. g'


simplify (Delta h f) = Delta h (simplify f)
simplify (D (I f')) = simplify f'
simplify (D f) = D (simplify f)
simplify (I f) = I (simplify f)
simplify f = f
```

With this new function, many expressions become much more readable!

```
ghci> f = Sin :+ Id:^(Const 2)
ghci> derive f
(cos + ((id^(2 - 1)) * ((2 * 1) + ((id * (log . id)) * 0))))
ghci> simplify (derive f)
(cos + (2 * id))
```

A sight for sore eyes!

Exercise. Think of more ways an expression can be “simplified”, and add your cases to the implementation.

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Calculus / Integral calculus

[src: Calculus/IntegralCalc.lhs] Previous: Differential calculus Table of contents Next: Plotting graphs

```
module Calculus.IntegralCalc (integrateApprox, integrate, eval) where

import Calculus.FunExpr
import Calculus.DifferentialCalc
```

Integrals - An integral part of calculus

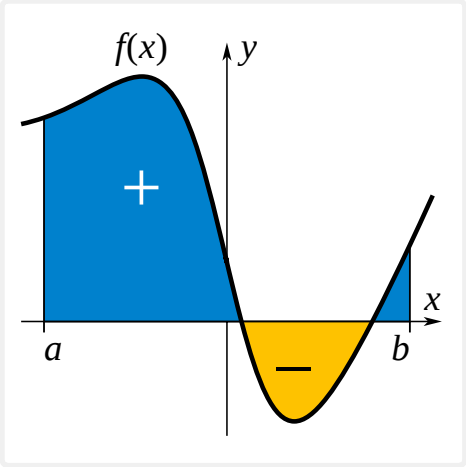
Integrals are functions used to describe area, volume, and accumulation in general. The operation of integration is the second fundamental operation of calculus, and the inverse of differentiation. Whereas derivatives are used to describe the rate of change in an instant, integrals are used to describe the accumulation of value over time.

Recall how we used derivatives before. If we know the distance traveled of a car and the time it took, we can use differentiation to calculate the velocity. Similarly but reversely, if we know the velocity of the car and the time it travels for, we can use integration to calculate the distance traveled.

$$x_{traveled} = \int_{t_0}^{t_1} v(t)dt$$

.

Ok, let's dive into this! We need to grok the syntax and find a rigorous, modelable definition of what *exactly* an integral is. We ask our kind friend Wikipedia for help. From the entry on *Integral*:

> Given a function f of a real variable x and an interval $[a, b]$ of the real line, the definite integral
>
> $$\int_a^b f(x)dx$$
>
> 
>
>
> A definite integral of a function can be represented as the signed area of the region bounded by its graph. (C) KSmrq
>
> is defined informally as the signed area of the region in the xy-plane that is bounded by the graph of f, the x-axis and the vertical lines $x = a$ and $x = b$. The area above the x-axis adds to the total and that below the x-axis subtracts from the total.
>
> Roughly speaking, the operation of integration is the inverse of differentiation. For this reason, the term integral may also refer to the related notion of the antiderivative, a function F whose derivative is the given function f. In this case, it is called an indefinite integral and is written:
>
> $$F(x) = \int f(x)dx$$

Ok, so first of all: confusion. Apparently there are two different kinds of integrals, *definite integrals* and *indefinite integrals*?

Let's start with defining *indefinite* integrals. *Wikipedia - Antiderivative* tells us that the *indefinite* integral, also known as the *antiderivative*, of a function f is equal to a differentiable function F such that $D(F) = f$. It further tells us that the process of finding the antiderivative is called *antidifferentiation* or *indefinite integration*.

The same article then brings further clarification

> Antiderivatives are related to definite integrals through the fundamental theorem of calculus: the definite integral of a function over an interval is equal to the difference between the values of an antiderivative evaluated at the endpoints of the interval.

So indefinite integrals are the inverse of derivatives, and definite integrals are just the application of an indefinite integral to an interval. If we look back at the syntax used, this makes sense. $\int f(x)dx$ is the indefinite integral. A function not applied to anything. $\int_a^b f(x)dx$ is the definite

integral. The difference of the indefinite integral being applied to the endpoints of the interval $[a, b]$.

To simplify a bit, we see that just as with derivatives the x's everywhere are just there to confuse us, so we remove them.

$$\int f(x)dx$$

should really just be

$$\int f$$

.

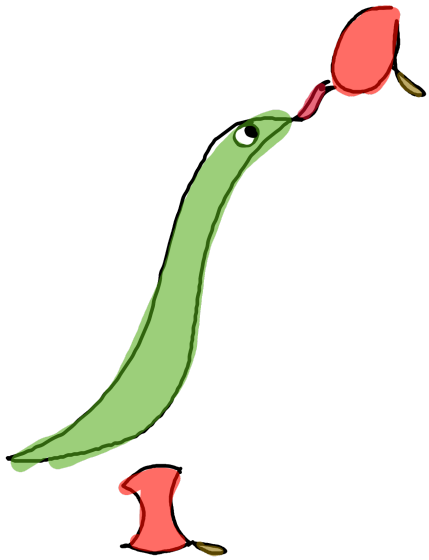
Next, the definition of definite integrals implies that we can write

$$\int_a^b f(x)dx$$

as

$$(\int f)[b] - (\int f)[a]$$

.



Only one of the two kinds of integral are fit to directly model in the syntax tree of our language. As `FunExpr` represents functions, it has to be the indefinite integral, which is a function unlike the definite integral which is a real value difference.

A thing to note is that while we may sometimes informally speak of the indefinite integral as a single unary function like any other, it's actually a set of functions, and the meaning of $F(x) = \int f(x)dx$ is really ambiguous. The reason for this is that for some function f, there is not just one function F such that $D(F) = f$. A simple counterexample is

$$D(x \mapsto x + 2) = 1 \text{ and } D(x \mapsto x + 3) = 1$$

The fact that adding a constant to a function does not change the the derivative, implies that the indefinite integral of a function is really a set of functions where the constant term differs.

$$\int f = \{F + const\ C \mid C \in \mathbb{R}\}$$

We don't want sets though. We want unary real functions (because that's our the type of our semantics!). So, we simply say that when integrating a function, the constant term will always be 0 in order to nail the result down to a single function! I.e. $(If)0 = 0$

```
    | I FunExpr
```

Actually integrating with my man, Riemann

We've analyzed *what* an integral is, and we can tell if a function is the antiderivative of another. For example, x^2 is an antiderivative of $2x$ because $D(x^2) = 2x$. But *how* do we find integrals in the first place?

We start our journey with a familiar name, Leibniz. He, and also but independently Newton, discovered the heart of integrals and derivatives: The *fundamental theorem of calculus*. The definitions they made were all based on infinitesimals which, as said earlier, was considered too imprecise. Later, Riemann rigorously formalized integration using limits.

There exist many formal definitions of integrals, and they're not all equivalent. They each deal with different cases and classes of problems, and some remain in use mostly for pedagogical purposes. The most commonly used definitions are the Riemann integrals and the Lebesgue integrals.

The Riemann integral was the first rigorous definition of the integral, and for many practical applications it can be evaluated by the fundamental theorem of calculus or approximated by numerical integration. However, it is a deficient definition, and is therefore unsuitable for many theoretical purposes. For such purposes, the Lebesgue integral is a better fit.

All that considered, we will use Riemann integrals. While they may be lacking for many purposes, they are probably more familiar to most students (they are to me!), and will be sufficient for the level we're at.

If we look back at the syntax of definite integrals

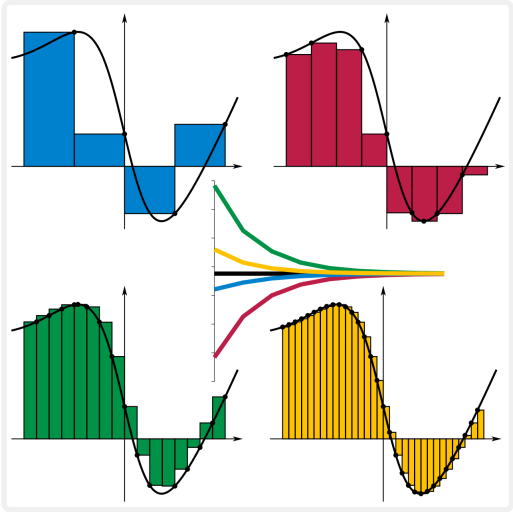
$$\int_a^b f(x)dx$$

the application of f and the dx part actually implies the definition of the Riemann integral. We can read it in english as "For every infinitesimal interval of x, starting at a and ending at b, take the value of f at that x (equiv. to taking the value at any point in the infinitesimal interval), and

calculate the area of the rectangle with width dx and height $f(x)$, then sum all of these parts together.".

As we're dealing with an infinite sum of infinitesimal parts: a limit must be involved. a and b are the lower and upper limits of the sum. Our iteration variable should increase with infinitesimal dx each step. Each step we add the area of the rectangle with height $f(x')$, where x' is any point in $[x, x + dx]$. As $x + dx$ approaches x when dx approaches zero, $x' = lim_{dx \to 0} x + dx = x$.

$$\int_a^b f = \int_a^b f(x)dx = lim_{dx \to 0} \sum_{x=a,a+dx,a+2dx,...}^{b} A(x, dx) \text{ where } A(x, dx) = f(x) * dx$$



Smaller dx result in better approximations. (C) KSmrq

Based on this definition, we could implement a function in Haskell to compute the numerical approximation of the integral by letting dx be a very small, but finite, number instead of being infinitesimal. The smaller our dx, the better the approximation

```
integrateApprox :: RealFunc -> RealNum -> RealNum ->
RealNum -> RealNum
integrateApprox f dx a b =
```

b must be greater than a for a definite integral to make sense, but if that's not the case, we can just flip the order of a and b and flip the sign of the area.

```
  let area x = f x * dx
  in if b >= a
     then sum (fmap area (takeWhile (<b) [a + 0*dx, a + 1*dx ..]))
     else -sum (fmap area (takeWhile (>b) [a - 0*dx, a - 1*dx ..]))
```

For example, let's calculate the area of the right-angled triangle under $y = x$ between $x = 0$ and $x = 10$. As the area of a right-angled triangle is calculated as $A = \frac{b*h}{2}$, we expect the result of to approach $A = \frac{b*h}{2} = \frac{10*10}{2} = 50$ as dx gets smaller

```
λ integrateApprox (\x -> x) 5    0 10
25
λ integrateApprox (\x -> x) 1    0 10
45
λ integrateApprox (\x -> x) 0.5  0 10
47.5
λ integrateApprox (\x -> x) 0.1  0 10
49.50000000000013
```

```
λ integrateApprox (\x -> x) 0.01 0 10
50.04999999999996
```

Great, it works for numeric approximations! This can be useful at times, but not so much in our case. We want closed expressions to use when solving physics problems, regardless of whether there are computations or not!

To find some integrals, making simple use of the fundamental theorem of calculus, i.e. $D(\int f) = f$, is enough. That is, we "think backwards". For example, we can use this method to find the integral of cos.

Which function derives to cos? Think, think, think ... I got it! It's sin, isn't it?

$$D(sin) = cos \implies \int cos = sin + constC$$

So simple! The same method can be used to find the integral of polynomials and some other simple functions. Coupled with some integration rules for products and exponents, this can get us quite far! But what if we're not superhumans and haven't memorized all the tables? What if we have to do integration without a cheat sheet for, like, an exam? In situations like these we make use of the definition of the Riemann integral, like we make use of the definition of differentiation in a previous chapter. As an example, let us again integrate cos, but now with this second method. Keep in mind that due to the technical limitations of Riemann integrals, not all integrals may be found this way.

Using the trigonometric identity of $\lim_{x \to 0} \frac{sinx}{x} = 1$ we find

$$\int_a^b cos$$

$$\{ \text{Riemann integral} \}$$

$$= \lim_{dx \to 0} \sum_{x=a,a+dx,a+2*dx,\ldots}^{b} cos(x) * dx$$

$$= \lim_{dx \to 0} dx * \sum_{x=a,a+dx,a+2*dx,\ldots}^{b} cos(x)$$

$$= \lim_{dx \to 0} dx * (cos(a) + cos(a+dx) + cos(a+2*dx) + \ldots + cos(a + \frac{b-a}{dx} * dx))$$

$$\{ \text{Sums of cosines with arguments in arithmetic progression} \}$$

$$= \lim_{dx \to 0} dx * \frac{sin(\frac{(\frac{b-a}{dx}+1)dx}{2}) * cos(a + \frac{\frac{b-a}{dx}dx}{2})}{sin(dx/2)}$$

$$= \lim_{dx \to 0} dx * \frac{sin(\frac{b-a+dx}{2}) * cos(\frac{a+b}{2})}{sin(dx/2)}$$

$$\{ \text{Trig. product-to-sum ident.} \}$$

$$= \lim_{dx \to 0} dx * \frac{sin(\frac{b-a+dx}{2} + \frac{a+b}{2}) + sin(\frac{b-a+dx}{2} - \frac{a+b}{2})}{2sin(dx/2)}$$

$$= \lim_{dx \to 0} dx * \frac{sin(b+dx/2) + sin(-a+dx/2)}{2sin(dx/2)}$$

$$= \lim_{dx \to 0} \frac{sin(b+dx/2) + sin(-a+dx/2)}{\frac{sin(dx/2)}{dx/2}}$$

$$\{dx \to 0\}$$

$$= \frac{sin(b+0/2) + sin(-a+0/2)}{1}$$

$$= sin(b) + sin(-a)$$

$$= sin(b) - sin(a)$$

The definition of definite integrals then give us that

$$\int_a^b cos = sin(b) - sin(a) \wedge \int_a^b f = F(b) - F(a) \implies F = sin$$

The antiderivative of cos is sin (again, as expected)!

Let's implement these rules as a function for symbolic (indefinite) integration of functions. We'll start with the nicer cases, and progress to the not so nice ones.

`integrate` takes a function to symbolically integrate, and returns the antiderivative where $F(0) = 0$.

Important to note is that not all functions are integrable. Unlike derivatives, some antiderivatives of elementary functions simply cannot be expressed as elementary functions themselves, according to Liouville's theorem. Some examples include e^{-x^2}, $\frac{sin(x)}{x}$, and x^x.

```
integrate :: FunExpr -> FunExpr
```

First, our elementary functions. You can prove them using the methods described above, but the easiest way to find them is to just look them up in some table of integrals (dust off that old calculus cheat sheet) or on WolframAlpha (or Wikipedia, or whatever. Up to you).

```
integrate Exp = Exp :- Const 1
```

Note that $log(x)$ is not defined in $x = 0$, so we don't have to add any corrective constant as $F(0)$ wouldn't make sense anyways.

```
integrate Log  = Id :* (Log :- Const 1)
integrate Sin  = Const 1 :- Cos
integrate Cos  = Sin
integrate Asin =    (Const 1 :- Id:^(Const 2)):^(Const 0.5)
                 :+ Id :* Asin
                 :- Const 0
integrate Acos =    Id :* Acos
                 :- (Const 1 :- Id:^(Const 2)):^(Const 0.5)
                 :+ Const 1
```

These two good boys. Very simple as well.

```
integrate Id        = Id:^Const 2 :/ Const 2
integrate (Const d) = Const d :* Id
```

Addition and subtraction is trivial. Just use the backwards method and compare to how sums and differences are differentiated.

```
integrate (f :+ g) = integrate f :+ integrate g
integrate (f :- g) = integrate f :- integrate g
```

Delta is easy. Just expand it to the difference that it is, and integrate.

```
integrate (Delta h f) = integrate (f :. (Id :+ Const h) :- f)
```

A derivative? That's trivial, the integration and differentiation cancel each other, right? Nope, not so simple! To conform to our specification of `integrate` that $F(0) = 0$, we have to make sure that the constant coefficient is equal to 0, which it might not be if we just cancel the operations. The simplest way to solve this is to evaluate the function at $x = 0$, and check the value. We then subtract a term that corrects the function such that $I(D(f))[0] = 0$. We'll write a simple function `center` for this later.

```
integrate (D f) = center f
```

Integrating an integral? Just integrate the integral!

```
integrate (I f) = integrate (integrate f)
```

Aaaaaand now it starts to get complicated.

There exists a great product rule in the case of differentiation, but not for integration. There just isn't any nice way to integrate a product that always works! The integration rule that's most analogous to the product rule for differentiation, is integration by parts:

$$\int f(x)g(x)dx = f(x)G(x) - \int f'(x)G(x)dx$$

Hmm, this doesn't look quite as helpful as the differentiation product rule, does it? We want this rule to give us an expression of simpler and/or fewer integrals, and it may indeed do so. For example, the integration of the product $x * e^x$ is a great examples of a case where it works well:

$$\int xe^x dx = xe^x - \int 1e^x dx = xe^x - e^x = e^x(x-1)$$

Now THAT is a simplification. However, just by flipping the order of the expressions, we get a case where the integration by parts rule only makes things worse:

$$\begin{aligned}
\int e^x x dx =& e^x \frac{x^2}{2} - \int e^x \frac{x^2}{2} dx \\
=& e^x \frac{x^2}{2} - (e^x \frac{x^3}{3!} - \int e^x \frac{x^3}{3} dx) \\
=& e^x \frac{x^2}{2} - (e^x \frac{x^3}{3!} - (e^x \frac{x^4}{4!} - \int e^x \frac{x^4}{4!} dx)) \\
=& \dots
\end{aligned}$$

Oh no, it's an infinite recursion with successive increase in complexity! Sadly, there's no good way around it. By using heuristics, we could construct a complicated algorithm that guesses the best order of factors in a product when integrating, but that's *way* out of scope for this book.

Further, as a consequence of Liouville's theorem, the integration by parts rule is simply not defined in the case of $g(x)$ not being integrable to $G(x)$. And so, as there exists no definitely good way to do it in ALL cases, we're forced to settle for a conservative approach.

If we rewrite the formula for integration by parts to use g' instead of g

$$\int f(x)g'(x)dx = f(x)g(x) - \int f'(x)g(x)dx$$

we see that there are two cases where the integral is well defined:

$$\int f(x)g'(x)dx = f(x)g(x) - \int f'(x)g(x)dx$$

and

$$\int f'(x)g(x)dx = f(x)g(x) - \int f(x)g'(x)dx$$

I.e., if we already know the integral of one factor, we can integrate the product.

```
integrate (D f :* g) = center (f :* g :- integrate (f :* derive g))
integrate (f :* D g) = center (f :* g :- integrate (derive f :* g))
```

Also, we can add a few cases for integrals we know, like multiplication with a constant

```
integrate (Const c :* f) = Const c :* integrate f
```

The rule for quotients is very similar

```
integrate (D f :/ g) =
  center (f :/ g :+ integrate (f :* (D g :/ (g:^Const 2))))
```

There is no good rule for exponentials in general. Only for certain combinations of functions in the base and exponent is symbolic integration well defined. We'll only treat the special case of x^c, which at least implies that we can use polynomials.

```
integrate (Id :^ Const c) = Id:^(Const (c+1)) :/ Const (c+1)
```

Exercise. Find more rules of integrating exponentials and add to the implementation.

▼ **Solution**

> Wikipedia has a nice list of integrals of exponentials

```
/details>
```

Integration of function composition is, simply said, somewhat complicated. The technique to use is called "integration by substitution", and is something like a reverse of the chain-rule of differentiation. This method is tricky to implement, as the way humans execute this method is highly dependent on intuition and a mind for patterns. A brute-force solution would be possible to implement, but is out of scope for this book, and not really relevant to what we want to learn here. We'll leave symbolic integration of composition undefined.

And if we couldn't integrate the expression as is, first try simplifying it and see if we know how to integrate the new expression. If that fails, just wrap the expression in the `I` constructor, unchanged. This will signify that we don't know how to symbolically integrate the expression. During evaluation, we can use `integrateApprox` to compute the integral numerically.

```
integrate f = let fsim = simplify f
              in if f == fsim
                 then I f
                 else integrate fsim
```

Finally, the helper function `center` to center functions such that $f(0) = 0$.

```
center f = f :- Const ((eval f) 0)
```

The value of evaluation

What comes after construction of function expressions? Well, using them of course!

One way of using a function expression is to evaluate it, and use it just as you would a normal Haskell function. To do this, we need to write an evaluator.

An evaluator simply takes a syntactic representation and returns the semantic value, i.e. `eval :: SYNTAX -> SEMANTICS`.

In the case of our calculus language:

```
eval :: FunExpr -> (RealFunc)
```

To then evaluate a `FunExpr` is not very complicated. The elementary functions and the `Id` function are simply substituted for their Haskell counterparts.

```
eval Exp = exp
eval Log = log
eval Sin = sin
eval Cos = cos
eval Asin = asin
eval Acos = acos
eval Id = id
```

`Const` is evaluated according to the definition $const(c) = x \mapsto c$

```
eval (Const c) = \x -> c
```

How to evaluate arithmetic operations on functions may not be as obvious, but we just implement them as they were defined earlier in the chapter.

```
eval (f :+ g) = \x -> (eval f x + eval g x)
eval (f :- g) = \x -> (eval f x - eval g x)
eval (f :* g) = \x -> (eval f x * eval g x)
eval (f :/ g) = \x -> (eval f x / eval g x)
eval (f :^ g) = \x -> (eval f x ** eval g x)
```

Function composition is similarly evaluated according to the earlier definition

```
eval (f :. g) = \x -> eval f (eval g x)
```

Delta is just expanded to the difference that it really is

```
eval (Delta h f) = eval (f :. (Id :+ Const h) :- f)
```

For derivatives, we just apply the symbolic operation we wrote, and then evaluate the result.

```
eval (D f) = eval (derive f)
```

With integrals, we first symbolically integrate the expression as far as possible, then apply the numerical `integrateApprox` if we can't figure out the integral further.

```
eval (I f) = let _F = simplify (integrate f)
             in if _F == (I f)
                then integrateApprox (eval f) 0.001 0
                else eval _F
```

[src: Calculus/IntegralCalc.lhs] Previous: Differential calculus Table of contents Next: Plotting graphs

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Calculus / Plotting graphs

[src: Calculus/VisVerApp.lhs] Previous: Integral calculus Table of contents Next: Syntax trees

```
module Calculus.VisVerApp where

import Calculus

import Hatlab.Plot
```

Plotting with Hatlab

The brain likes seeing things. Let's give it a good looking reward!

We'll now make combined use of all of our nice functions. `simplify`, `derive`, `integrate`, `eval`, and `show`, all together: the most ambitious crossover event in history!

First, we create some function expressions ready to be `show`n and `eval`uated.

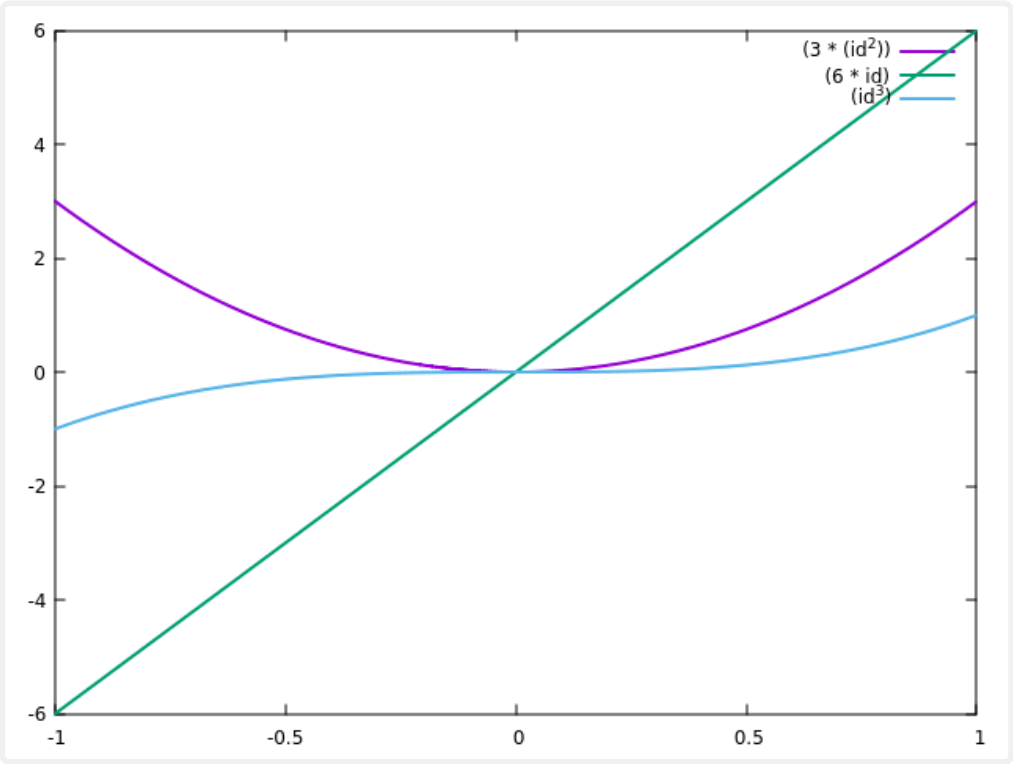
```
f  = Const 3 :* Id:^Const 2
f' = simplify (derive f)
_F = simplify (integrate f)
```

Then, we define a helper function to plot a list of function expressions with Hatlab.

```
plotFunExprs :: [FunExpr] -> IO ()
plotFunExprs = plot . fmap (\f -> Fun (eval f) (show f))
```

Now try it for yourself! Let's see the fruits of our labour!

```
ghci> plotFunExprs [f, f', _F]
```

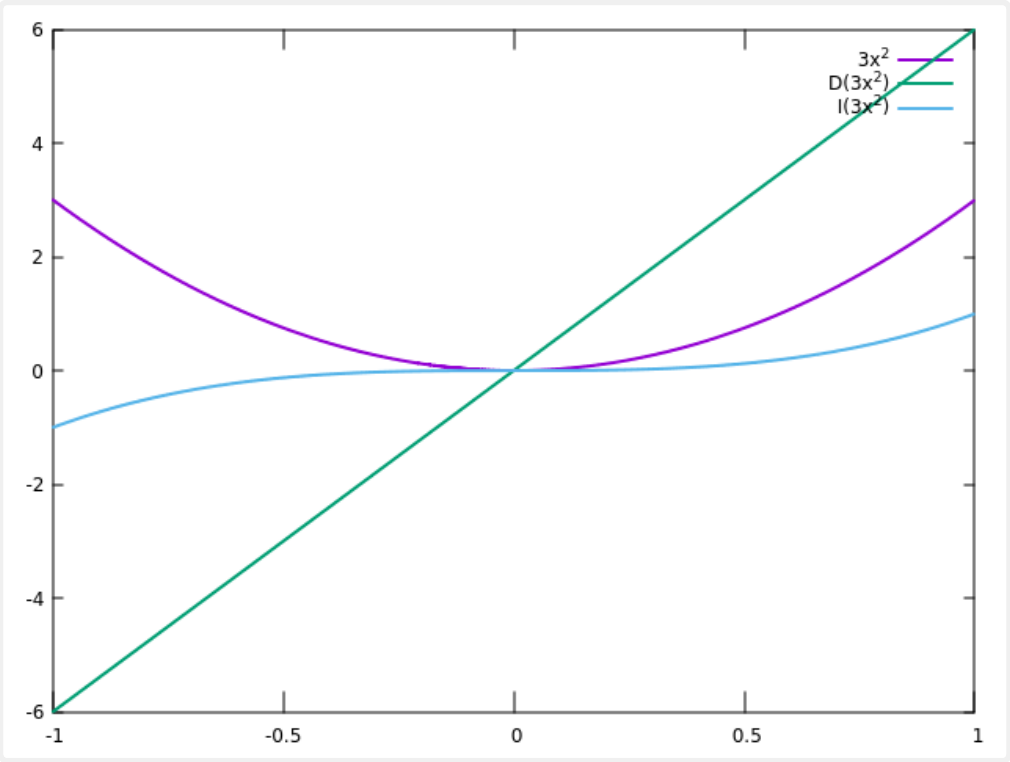


For fun, we can also plot the same functions but using our approximative functions for differentiation and integration

```
g x  = 3 * x^2
g' x = deriveApprox g 0.001 x
_G x = integrateApprox g 0.001 0 x
```

Then plot with

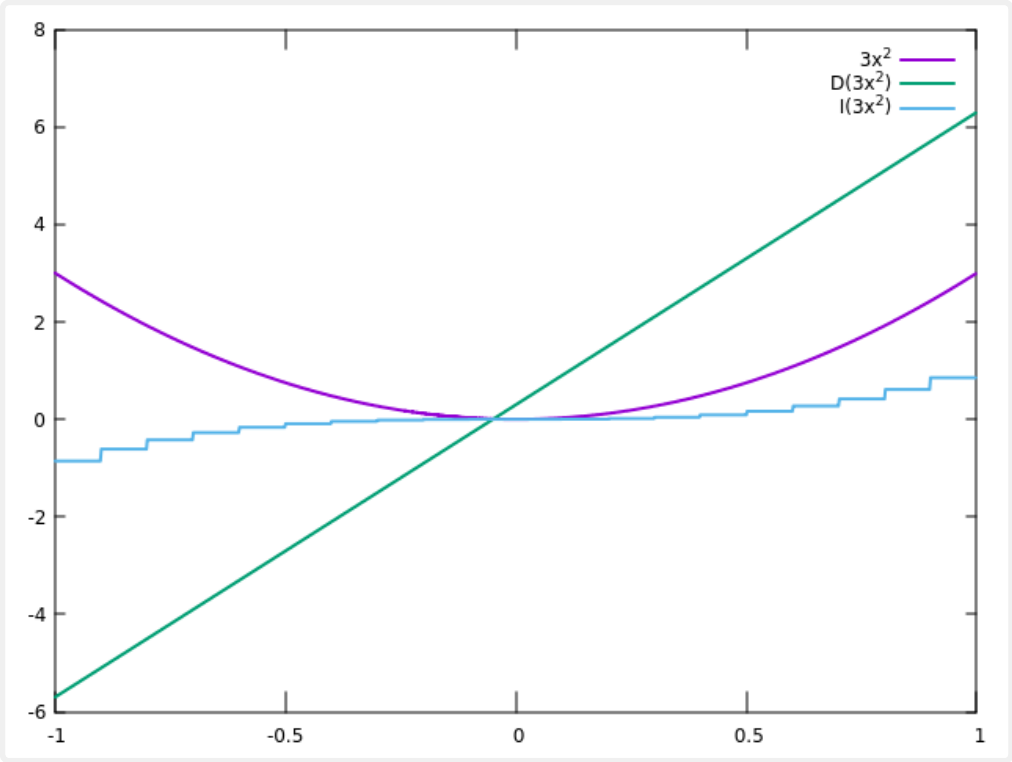
```
ghci> plot [Fun g "3x^2", Fun g' "D(3x^2)", Fun _G "I(3x^2)"]
```



Waddaya know! They look identical! I guess it just goes to show that a good approximation is often good enough.

If we turn down the precision, we start to notice the errors

```
h x  = 3 * x^2
h' x = deriveApprox h 0.1 x
_H x = integrateApprox h 0.1 0 x


ghci> plot [Fun h "3x^2", Fun h' "D(3x^2)", Fun _H "I(3x^2)"]
```



[src: Calculus/VisVerApp.lhs] Previous: Integral calculus Table of contents Next: Syntax trees

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Calculus / Syntax trees

[src: Calculus/SyntaxTree.lhs] Previous: Plotting graphs Table of contents Next: Vectors

```
module Calculus.SyntaxTree where


import           Calculus.FunExpr
import           Calculus.DifferentialCalc
import           Calculus.IntegralCalc
```

A fun and useful way to visualize expressions is to model them as trees. In our case we want to model $FunExpr$ where the nodes and leaves will be our constructors.

In order to do this will import two packages, one for constructing trees and one for pretty printing them.

```
import           Data.Tree         as T
import           Data.Tree.Pretty  as P
```

Now we can construct the function that takes a FunExpr and builds a tree from it. Every node is a string representation of the constructor and a list of its sub trees (branches).

```
makeTree :: FunExpr -> Tree String
makeTree (e1 :+ e2)  = Node "+"    [makeTree e1, makeTree e2]
makeTree (e1 :- e2)  = Node "-"    [makeTree e1, makeTree e2]
makeTree (e1 :* e2)  = Node "*"    [makeTree e1, makeTree e2]
makeTree (e1 :/ e2)  = Node "Div"  [makeTree e1, makeTree e2]
makeTree (e1 :^ e2)  = Node "**"   [makeTree e1, makeTree e2]
makeTree (e1 :. e2)  = Node "o"    [makeTree e1, makeTree e2]
makeTree (D e)       = Node "d/dx" [makeTree e]
makeTree (Delta r e) = Node "Δ"    [makeTree (Const r), makeTree e]
```

```
makeTree (I e)       = Node "I"    [makeTree e]
makeTree Id          = Node "Id"   []
makeTree Exp         = Node "Exp"  []
makeTree Log         = Node "Log"  []
makeTree Sin         = Node "Sin"  []
makeTree Cos         = Node "Cos"  []
makeTree Asin        = Node "Asin" []
makeTree Acos        = Node "Acos" []
makeTree (Const num) = Node (show num) [] --(show (floor num)) [] -- | Note the use of
  floor
```

Now we construct trees from our expressions but we still need to print them out. For this we'll use the function `drawVerticalTree` which does exactly what its name suggests. We can then construct a function to draw expressions.

```
printExpr :: FunExpr -> IO ()
printExpr = putStrLn . drawVerticalTree . makeTree
```

Now let's construct a mildly complicated expression

```
e = Delta 3 (Delta (negate 5) (I Acos) :. (Acos :* Exp))
```

And print it out.

```
ghci > printExpr e
          Δ
          |
  -----------
 /           \
 3           o
             |
         ----------
        /          \
        Δ          *
        |          |
       ----       -----
      /    \     /     \
     -5    I   Acos   Exp
           |
          Acos
```

Pretty prints the steps taken when canonifying an expression

```
prettyCan :: FunExpr -> IO ()
prettyCan e =
  let t  = makeTree e
      e' = canonify e
      t' = makeTree e'
    in if t == t' then putStrLn $ drawVerticalTree t
                  else do
                      putStrLn $ drawVerticalTree t
                      prettyCan e'
```

Pretty prints syntactic checking of equality

```
prettyEqual :: FunExpr -> FunExpr -> IO Bool
prettyEqual e1 e2 = if e1 == e2 then
  do
    putStrLn "It's equal!"
    putStrLn $ drawVerticalForest [makeTree e1, makeTree e2]
    return True
  else do
    putStrLn "Not equal -> Simplifying"
    putStrLn $ drawVerticalForest [makeTree e1, makeTree e2]
    let c1 = canonify e1
        c2 = canonify e2
      in if c1 == e1 && c2 == e2 then putStrLn "Can't simplify no more"
                                          >> return False
                                 else prettyEqual c1 c2
```

Syntactic checking of equality

```
equal :: FunExpr -> FunExpr -> Bool
equal e1 e2 = (e1 == e2) ||
                (let c1 = canonify e1
                     c2 = canonify e2
                   in (not (e1 == c1 && c2 == e2) && equal c1 c2))
```

Parse an expression as a Tree of Strings

Of course this is all bit too verbose, but I'm keeping it that way until every case is covered, Calculus is a bit of a black box for me right now

```
canonify :: FunExpr -> FunExpr
```

Addition

```
canonify (e :+ Const 0)       = canonify e
canonify (Const 0 :+ e)       = canonify e
canonify (Const x :+ Const y) = Const (x + y)
canonify (e1 :+ e2)           = canonify e1 :+ canonify e2
```

Subtraction

```
canonify (e :- Const 0)       = canonify e
canonify (Const a :- Const b) = Const (a - b)
canonify (e1 :- e2)           = canonify e1 :- canonify e2
```

Multiplication

```
canonify (_ :* Const 0)       = Const 0
canonify (Const 0 :* _)       = Const 0
canonify (e :* Const 1)       = canonify e
canonify (Const 1 :* e)       = canonify e
canonify (Const a :* Const b) = Const (a * b)
canonify (e1 :* e2)           = canonify e1 :* canonify e2
```

Division

```
canonify (Const a :/ Const b) = Const (a / b)
canonify (e1 :/ e2)           = canonify e1 :/ canonify e2
```

Delta

```
canonify (Delta r e)            = Delta r $ canonify e
```

Derivatives

```
canonify (D e)                = derive e
```

Composition

```
canonify (e1 :. e2)           = canonify e1 :. canonify e2
```

Catch all

```
canonify e                    = e
```

“Proofs”

```
syntacticProofOfComForMultiplication :: FunExpr -> FunExpr -> IO Bool
syntacticProofOfComForMultiplication e1 e2 = prettyEqual (e1 :* e2) (e2 :* e1)


syntacticProofOfAssocForMultiplication :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfAssocForMultiplication e1 e2 e3 = prettyEqual (e1 :* (e2 :* e3))
                                                              ((e1 :* e2) :* e3)


syntacticProofOfDistForMultiplication :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfDistForMultiplication e1 e2 e3 = prettyEqual (e1 :* (e2 :+ e3))
                                                             ((e1 :* e2) :+ (e1 :*
  e3))


{- syntacticProofOfIdentityForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfIdentityForMultiplication e = -}
{-   putStrLn "[*] Checking right identity" >> -}
{-     prettyEqual e (1 :* e) >> -}
{-       putStrLn "[*] Checking left identity" >> -}
{-         prettyEqual e (e :* 1) -}


{- syntacticProofOfPropertyOf0ForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfPropertyOf0ForMultiplication e = -}
{-   prettyEqual (e :* 0) 0 -}


-- | Fails since default implementation of negate x for Num is 0 - x
{- syntacticProofOfPropertyOfNegationForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfPropertyOfNegationForMultiplication e = -}
{-   prettyEqual (Const (-1) :* e)  (negate e) -}


syntacticProofOfComForAddition :: FunExpr -> FunExpr -> IO Bool
syntacticProofOfComForAddition e1 e2 = prettyEqual (e1 :+ e2) (e2 :+ e1)


syntacticProofOfAssocForAddition :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfAssocForAddition e1 e2 e3 = prettyEqual (e1 :+ (e2 :+ e3))
                                                        ((e1 :+ e2) :+ e3)


test :: FunExpr -> FunExpr -> IO Bool
test b c = prettyEqual b (a :* c)
```

```
  where
    a = b :/ c


syntacticProofOfIdentityForAddition :: FunExpr -> IO Bool
syntacticProofOfIdentityForAddition e = putStrLn "[*] Checking right identity" >>
  prettyEqual e (0 :+ e) >>
    putStrLn "[*] Checking left identity" >>
      prettyEqual e (e :+ 0)
```

Dummy expressions

```
e1 = Const 1
e2 = Const 2
e3 = Const 3
e4 = (Const 1 :+ Const 2) :* (Const 3 :+ Const 4)
e5 = (Const 1 :+ Const 2) :* (Const 4 :+ Const 3)
e6 = Const 2 :+ Const 3 :* Const 8 :* Const 19
```

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Linear algebra / Vectors


```
module Vector.Vector where
import Test.QuickCheck hiding (scale)
```

Vectors in two dimensions.

A physical quantity that has only a magnitude is called a scalar. In Haskell we'll represent this using a Double.

```
type Scalar = Double
```

A vector is a quantity that has both a magnitude and a direction. For instance the *velocity* of a moving body involves its speed (magnitude) and the direction of motion.

We cen represent the direction of a vector in two dimensions using its x and y coordinates, which are both scalars. The direction is then given by the angle between these coordinates and the origin (0,0).

In order to maintain generalizability and useability in the next chapters we'll use a typesynonym in the data declaration but throughout this whole chapter we'll use Scalars as we've defined them here.

```
data Vector2 num = V2 num num
```

We can even introduce a type that makes that specific relationship clearer:

```haskell
type VectorTwo = Vector2 Scalar
```

The magnitude of the vector is it's length. We can calculate this using Pythagorean theorem:

$$x^2 + y^2 = magnitude^2$$

In haskell this would be:

```haskell
magnitude :: VectorTwo -> Scalar
magnitude (V2 x y) = sqrt (x^2 + y^2)
```

And now we can calulate the magnitude of a vector in two dimensions:

```
Vector> let vec = V2 5 3
Vector> magnitude vec
5.830951894845301
```

Addition and subtraction of vectors is accomplished using the components of the vectors. For instance when adding the forces (vectors) acting on a body we would add the components of the forces acting in the x direction and the components in the y direction. So our functions for adding and subtracting vectors in two dimensions are:

```haskell
add :: VectorTwo -> VectorTwo -> VectorTwo
add (V2 x1 y1) (V2 x2 y2) = V2 (x1 + x2) (y1 + y2)

sub :: VectorTwo -> VectorTwo -> VectorTwo
sub (V2 x1 y1) (V2 x2 y2) = V2 (x1 - x2) (y1 - y2)
```

But this only works for two vectors. In reality we might be working with several hundreds of vectors so it would be useful to add, for instance a list of vectors together and get one final vector as a result. We can use *foldr* using the zero vector as a starting value. to acomplish this.

```haskell
zeroVector :: VectorTwo
zeroVector = V2 0 0


addListOfVectors :: [VectorTwo] -> VectorTwo
addListOfVectors = foldr add zeroVector
```

Let’s try it out!

```
*Vector> let vec1 = V2 5 3
*Vector> let vec2 = V2 6 5
*Vector> sub vec1 vec2

<interactive>:23:1: error:
    • No instance for (Show VectorTwo) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

The interpreter is complaining that it doesn’t know how to interpret our datatype for vectors as a string. The easy solution would be to just derive our instance for Show, but to really solidify the fact that we are working with coordinates let’s make our own instance for Show.

```
instance Show num => Show (Vector2 num) where
  show (V2 x y) = "(" ++ show x ++ ", " ++ show y ++ ")"
```

And let’s try our example again:

```
*Vector> let vec1 = V2 5 3
*Vector> let vec2 = V2 6 5
*Vector> sub vec1 vec2
(-1 x, -2 y)
```

And let’s also try adding a list of vectors using our new function:

```
*Vector> let vec3 = V2 8 9
*Vector> let vectors = [vec1, vec2, vec3]
*Vector> addListOfVectors vectors
(19.0 x, 17.0 y)
```

It works!

We can also multiply a vector by a scalar. This is also done componentwise. We’ll call this scaling a vector. So we could double a vector by multiplying it with 2.0 and halving it by multiplying it with 0.5.

```
scale :: Scalar -> VectorTwo -> VectorTwo
scale factor (V2 x y) = V2 (factor * x) (factor * y)
```

Combining this with the unit vectors:

```haskell
unitX :: VectorTwo
unitX = V2 1 0

unitY :: VectorTwo
unitY = V2 0 1
```

We get a new way of making vectors, namely by scaling the unit vectors and adding them together. Let's create the vector (5 x, 3 y) using this approach.

```
*Vector> add (scale 5 unitX) (scale 3 unitY)
(5.0 x, 3.0 y)
```

In order to check that this vector is actually equal to the vector created using the contructor *V2* we need to make our vector an instance of *Eq*.

```haskell
instance Eq num => Eq (Vector2 num) where
  (V2 x1 y1) == (V2 x2 y2) = (x1 == x2) && (y1 == y2)
```

Let's try it out:

```
*Vector> let vec1 = V2 5 3
*Vector> let vec2 =  add (scale 5 unitX) (scale 3 unitY)
*Vector> vec1 == vec2
True
```

We have one final important operation left to define for vectors in two dimensions, the dot product. The formula is quite simple:

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y$$

And our function simply becomes:

```haskell
dotProd :: VectorTwo -> VectorTwo -> Scalar
dotProd (V2 ax ay) (V2 bx by) = ax * bx + ay * by
```

But this doesn't give us any intuition about what it means to take the dot product between vectors. The common interpretation is "geometric projection", but that only makes sense if you already understand the dot product. Let's try to give an easier analogy using the dash panels (boost pads) from *Mario Kart*. The dash panel is designed to give you boost of speed in a specific direction, usually straight forward. So the vector associated with the

dash panel can be represented with a unit vector multiplied with some factor of boost, say 10.

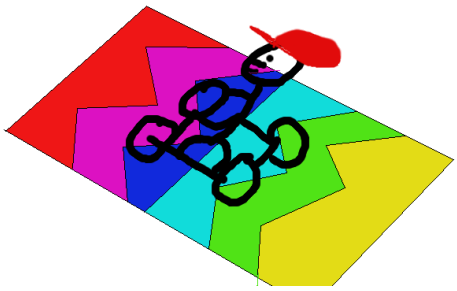
```
dashPanel :: VectorTwo
dashPanel = scale 10 unitY
```

Now let's say that your cart has this arbitrarily chosen velocity vector:

```
cart :: VectorTwo
cart = V2 3 5
```

Depending on which angle you hit the dash panel you'll receive different amounts of boost. Since the x-component of the dashPanel is 0 any component of speed on the x-direction will be reduced to zero. Only the speed in the direction of y will be boosted. But there are worse ways to hit the dash panel. We could for instance create a new velocity vector with the exact same magnitude of speed but which would recieve a worse boost.

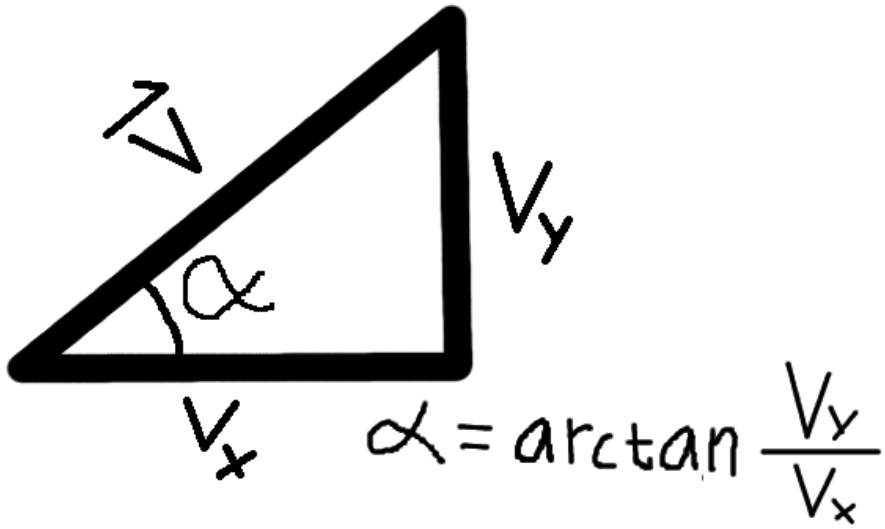
```
worseCart :: VectorTwo
worseCart = V2 5 3
```



Let's see this in action.

```
*Vector> magnitude cart == magnitude worseCart
True
*Vector> dotProd dashPanel cart
50.0
*Vector> dotProd dashPanel worseCart
30.0
```

We talked a lot about angles between vectors but we havn't used it in our code, so lets make a function which calculates the angle of a vector. The formula is as follows:



We'll use Doubles to represent the angle.

```
type Angle  = Double

angle :: VectorTwo -> Scalar
angle (V2 x y) = atan y/x
```

Using angles and magnitudes we can even write a new function for making vectors:

```
mkVector :: Scalar -> Angle -> VectorTwo
mkVector mag angle = V2 x y
  where
    x = mag * cos angle
    y = mag * sin angle
```

Vectors in three dimensions.

The datatype for a vector in three dimensions is basically the same as vector in two dimensions, we'll just add a z-component.

Again we will generalize this over a type synonym

```
data Vector3 num = V3 num num num
```

And a type synonym that we can use throughout the rest of this chapter.

```haskell
type VectorThree = Vector3 Scalar
```

Similarily the functions for adding three dimensional vectors:

```haskell
add :: VectorThree -> VectorThree -> VectorThree
add (V3 x1 y1 z1) (V3 x2 y2 z2) = V3 (x1 + x2) (y1 + y2) (z1 + z2)
```

Multiplying with a scalar:

```haskell
scale3 :: Scalar -> VectorThree -> VectorThree
scale3 fac (V3 x y z) = V3 (fac * x) (fac * y) (fac * z)
```

And for calculating the magnitude:

```haskell
mag3 :: VectorThree -> Scalar
mag3 (V3 x y z) = sqrt (x**2 + y**2 + z**2)
```

Looks earily similar to our functions for vectors in two dimensions. This suggest that there might be a better way to handle this, in order to avoid repeating ourselves.

Addition and subtraction on vectors works by “unpacking” the vectors, taking their components, applying some function to them (+/-) and then packing them up as a new vector. This is very similar to the Haskell function *zipWith* which works over lists instead of vectors.

```haskell
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

When we’re multiplying with a scalar we again unpack the vector and then apply mulitiplication with a factor to each component before packing it up again. This is quite similar to the Haskell function *map*, which again works over lists.

```haskell
map :: (a -> b) -> [a] -> [b]
```

When calculating the magnitude of a vector we first unpack the vector and then apply ^{2} to each component of the vector. This is doable with aformentioned *map*. We then *fold* the components together using $+$ which results in a final scalar value. Those of you familliar with functional languages will know where I’m going with this, those of you who aren’t will hopefully understand where I’m going when reading the examples.

Using this information we can now create a new class for vectors which implement this functionality:

```
class Vector vector where
  vmap      :: (num -> num)          -> vector num -> vector num
  vzipWith  :: (num -> num  -> num) -> vector num -> vector num -> vector num
  vfold     :: (num -> num  -> num) -> vector num -> num
```

Now we have a blueprint for what vector is, so let's implement it for our own vector datatypes.

```
instance Vector Vector2 where
  vmap     f (V2 x y)            = V2 (f x)    (f y)
  vzipWith f (V2 x y) (V2 x' y') = V2 (f x x') (f y y')
  vfold    f (V2 x y)            = f x y

instance Vector Vector3 where
  vmap     f (V3 x y z)                = V3 (f x)    (f y)    (f z)
  vzipWith f (V3 x y z) (V3 x' y' z') = V3 (f x x') (f y y') (f z z')
  vfold    f (V3 x y z)                = f z $ f x y
```

Now we're finally leveraging the power of the Haskell typesystem!

We can now implement more generalized functions for addition and subtraction between vectors.

```
add :: (Num num, Vector vec) => vec num -> vec num -> vec num
add = vzipWith (+)

sub :: (Num num, Vector vec) => vec num -> vec num -> vec num
sub = vzipWith (-)
```

For multiplying with a scalar:

```
scale :: (Num num, Vector vec) => num -> vec num -> vec num
scale factor = vmap (* factor)
```

And for calculating the magnitude of a vector:

```
magnitude :: (Floating num, Vector vec) => vec num -> num
magnitude = sqrt . vfold (+) . vmap (**2)
```

We can even use it to make a generalized function for calculating the dot product.

```
dotProd :: (Num num, Vector vec) => vec num -> vec num -> num
dotProd v1 v2 = vfold (+) $ vzipWith (*) v1 v2
```

Cross Product

We have one final function left to define, the cross product. The formula is as follows:

$$\vec{a} \times \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot sin(\theta)$$

Where θ is the angle between the vectors. And $|\vec{a}|$, $|\vec{b}|$ are the magnitudes of the vectors.

So our function for calculating the cross product becomes:

TODO: Generate normal vector as well. Codify right hand rule

```
crossProd :: Vector3 -> Vector3 -> Vector3
crossProd a b = (magnitude a) * (magnitude b) * sin (angleBetween a b)
  where
    angleBetween :: (Vector vec) => vec -> vec -> Scalar
    angleBetween v1 v2 = acos ((dotProd v1 v2) / ((magnitude v1) * (magnitude v2)))
```

Working cross product using matrix rules.

```
crossProd :: Num num => Vector3 num -> Vector3 num -> Vector3 num
crossProd (V3 x y z) (V3 x' y' z') = V3 (y*z' - z*y') -- X
                                        (z*x' - x*z') -- Y
                                        (x*y' - y*x') -- Z
```

Quickcheck!

There are certain laws or preperties that vectors adher to, for example the jacobi identity:

$$\vec{a} \times (\vec{b} \times \vec{c}) + \vec{b} \times (\vec{c} \times \vec{a}) + \vec{c} \times (\vec{a} \times \vec{b}) = 0$$

Or that the cross product is anticommutative. We can't actually prove these in a meaningful way without a whole bunch of packages and pragmas, but we can quickcheck

them. But to do that we need to be able to generate vectors, so let's make our vectors an instance of *Arbitrary*.

We do this by generating arbitrary scalars and then constructing vectors with them.

```haskell
instance Arbitrary num => Arbitrary (Vector2 num) where
  arbitrary = arbitrary >>= (\(s1, s2) -> return $ V2 s1 s2)

instance Arbitrary num => Arbitrary (Vector3 num) where
  arbitrary = arbitrary >>= (\(s1, s2, s3) -> return $ V3 s1 s2 s3)
```

Let's try it out!

```
ghci> generate arbitrary :: IO (Vector2 Scalar)
(-26.349975377051404 x, 9.71134047527185 y)
```

Seems pretty random to me.

Now we can check some properties, lets' start with commutativity of vector addition:

$$\vec{a} + \vec{b} = \vec{b} + \vec{a}$$

Which translates to:

```haskell
prop_CommutativityAddition :: VectorThree -> VectorThree -> Bool
prop_CommutativityAddition v1 v2 = v1 + v2 == v2 + v1
```

And we test this in the *repl*.

```
ghci> quickCheck prop_CommutativityAddition
+++ OK, passed 100 tests.
```

And associativity of addition:

$$\vec{a} + (\vec{b} + \vec{c}) = (\vec{a} + \vec{b}) + \vec{c}$$

```haskell
prop_AssociativityAddition :: VectorThree -> VectorThree -> VectorThree -> Bool
prop_AssociativityAddition a b c = a + (b + c) == (a + b) + c
```

```
ghci> quickCheck prop_AssociativityAddition
*** Failed! Falsifiable (after 2 tests):
(0.5240133611343812 x, -0.836882545823441 y, -4776.775557184785 z)
```

```
(-0.17261751005585407 x, 0.7893754200476363 y, -0.19757165887775568 z)
(0.3492200657348603 x, 0.10861834028920295 y, 0.45838513657221946 z)
```

This is very strange since the laws should always be correct. But this error stems from the fact that we're using a computer and that using doubles (Scalar) will introduce approximation errors. We can fix this by relaxing our instance for *Eq* and only requiring the components of the vectors to be approximately equal.

% TODO: Equation

```haskell
eps :: Floating num => num
eps = 1 * (10 ** (-5))

instance (Floating num, Eq num, Ord num) => Eq (Vector2 num) where
  (V2 x1 y1) == (V2 x2 y2) = xCheck && yCheck
    where
      xCheck = abs (x1 - x2) <= eps
      yCheck = abs (y1 - y2) <= eps
```

Let's try again.

```
*Vector.Vector> quickCheck prop_AssociativityAddition
+++ OK, passed 100 tests.
```

More laws

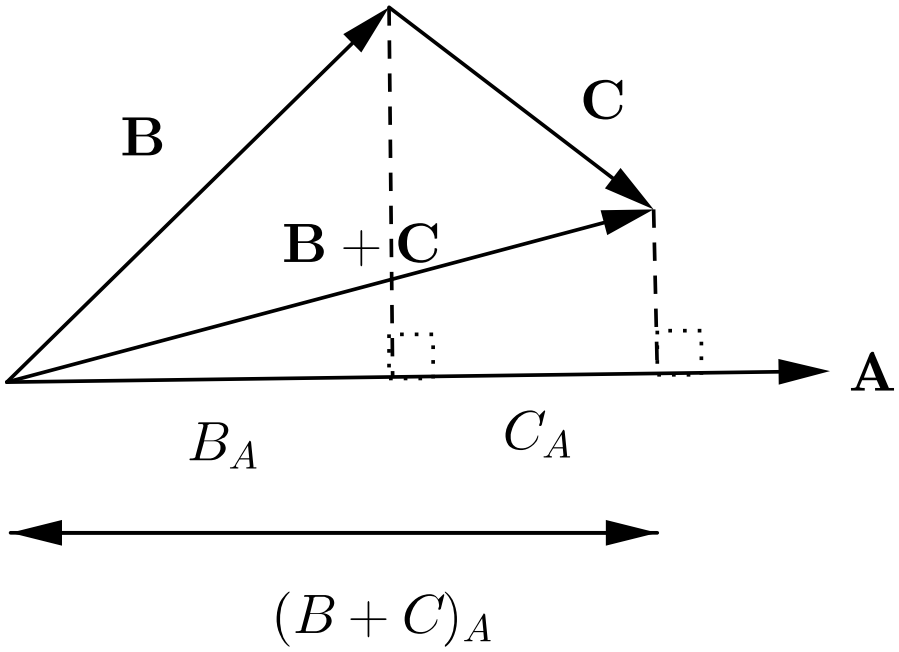
Dot product is commutative:

```haskell
prop_dotProdCommutative :: VectorThree -> VectorThree -> Bool
prop_dotProdCommutative a b = dotProd a b == dotProd b a
```

In order to check some laws which depends on checking the equality of scalars we'll introduce a function which checks that two scalars are approximately equal.

```haskell
-- Approx equal
(~=) :: Scalar -> Scalar -> Bool
rhs ~= lhs = abs (rhs - lhs) <= eps
```

Dot product is distributive over addition:



$$\vec{a} \cdot (\vec{b} + \vec{c}) = (\vec{a} \cdot \vec{b}) + (\vec{a} \cdot \vec{c})$$

```
prop_dotProdDistrubitiveAddition ::  VectorThree -> VectorThree -> VectorThree -> Bool
prop_dotProdDistrubitiveAddition a b c = dotProd a (b + c) ~== (dotProd a b + dotProd a
  c)
```

The dot product is homogeneous under scaling in each variable:

$$(x * \vec{a}) \cdot b = x * (\vec{a} \cdot \vec{b}) = \vec{a} \cdot (x * \vec{b})$$

```
prop_dotProdHomogeneousScaling :: Scalar -> VectorThree -> VectorThree -> Bool
prop_dotProdHomogeneousScaling x a b = e1 == e2 && e2 == e3
  where
    e1 = dotProd (scale 0 a) b
    e2 = 0 * dotProd a b
    e3 = dotProd a (scale 0 b)
```

The cross product of a vector with itself is the zero vector.

```
prop_crossProd_with_self :: VectorThree -> Bool
prop_crossProd_with_self v = crossProd v v == 0
```

The crossproduct is anticommutative:

$$\vec{a} \times \vec{b} = -(\vec{b} \times \vec{a})$$

```
prop_crossProdAntiCommutative :: VectorThree -> VectorThree -> Bool
prop_crossProdAntiCommutative v1 v2 = v1 * v2 == - (v2 * v1)
```

The cross product is distributive over addition:

$$\vec{a} \times (\vec{b} + \vec{c}) = (\vec{a} \times \vec{b}) + (\vec{a} \times \vec{c})$$

```
prop_crossProdDistrubitiveAddition :: VectorThree -> VectorThree -> VectorThree ->
  Bool
prop_crossProdDistrubitiveAddition a b c = a * (b + c) == (a * b) + (a * c)
```

Vector triple product (Lagrange's formula).

$$\vec{a} \times (\vec{b} \times \vec{c}) = \vec{b}(\vec{a} \cdot \vec{c}) - \vec{c}(\vec{a} \cdot \vec{b})$$

```
prop_lagrange :: VectorThree -> VectorThree -> VectorThree -> Bool
prop_lagrange a b c = a * (b * c) == (scale (dotProd a c) b -
                                      scale (dotProd a b) c)
```

The Jacobi identity:

$$\vec{a} \times (\vec{b} \times \vec{c}) + \vec{b} \times (\vec{c} \times \vec{a}) + \vec{v} \times (\vec{a} \times \vec{b})$$

```
prop_JacobiIdentity :: VectorThree -> VectorThree -> VectorThree -> Bool
prop_JacobiIdentity a b c = a * (b * c) +
                            b * (c * a) +
                            c * (a * b) == 0
```

Fun instances

```
instance Num num => Monoid (Vector2 num) where
  mempty  = zeroVector
  mappend = (+)
  mconcat = foldr mappend mempty

instance Num num => Monoid (Vector3 num) where
  mempty  = zeroVector
  mappend = (+)
  mconcat = foldr mappend mempty
```

```haskell
instance Num num => Num (Vector2 num) where
  (+)           = vzipWith (+)
  (*)           = undefined -- Crossproduct not defined for Vector2
  abs           = vmap abs
  negate        = vmap (*(-1))
  -- | Signum can be though of as the direction of a vector
  signum        = vmap signum
  fromInteger i = V2 (fromInteger i) 0

instance Num num => Num (Vector3 num) where
  (+)           = vzipWith (+)
  (*)           = crossProd
  abs           = vmap abs
  negate        = vmap (*(-1))
  -- | Signum can be though of as the direction of a vector
  signum        = vmap signum
  fromInteger i = V3 (fromInteger i) 0 0

-- TODO: Explain why this works

zeroVector :: (Vector vec, Num (vec num)) => vec num
zeroVector = 0

instance Show num => Show (Vector3 num) where
  show (V3 x y z) = "(" ++ show x ++ " x, "
                        ++ show y ++ " y, "
                        ++ show z ++ " z)"
instance (Floating num, Ord num) => Ord (Vector2 num) where
  compare v1 v2 = compare (magnitude v1) (magnitude v2)

instance (Floating num, Ord num) => Ord (Vector3 num) where
  compare v1 v2 = compare (magnitude v1) (magnitude v2)

instance (Ord num, Floating num, Eq num) => Eq (Vector3 num) where
  (V3 x y z) == (V3 x' y' z') = xCheck && yCheck && zCheck
    where
    xCheck = abs (x - x') <= eps
    yCheck = abs (y - y') <= eps
    zCheck = abs (z - z') <= eps


runTests :: IO ()
runTests = do
  putStrLn "Commutativity of vector addition:"
  quickCheck prop_CommutativityAddition
```

```
putStrLn "Associativity of vector addition:"
quickCheck prop_AssociativityAddition
putStrLn "Dot product distributive over addition:"
quickCheck prop_dotProdDistrubitiveAddition
putStrLn "Homogeneous scaling:"
quickCheck prop_dotProdHomogeneousScaling
putStrLn "Commutative dot product:"
quickCheck prop_dotProdCommutative
putStrLn "Crossproduct of a vector with itself:"
quickCheck prop_crossProd_with_self
putStrLn "Cross product is anticommutative"
quickCheck prop_crossProdAntiCommutative
putStrLn "Cross product distributive over addition"
quickCheck prop_crossProdDistrubitiveAddition
putStrLn "Lagrange formula"
quickCheck prop_lagrange
putStrLn "Jacobi identity"
quickCheck prop_JacobiIdentity
```

[src: Vector/Vector.lhs] Previous: Syntax trees Table of contents Next: Introduction

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Introduction

[src: Dimensions/Intro.lhs] Previous: Vectors Table of contents Next: Value-level dimensions

Introduction

```
module Dimensions.Intro where
```

This chapter is about dimensions, quantities and units. What's the difference?

- A **dimension** is "what type of thing something is". For instance, *length* is a thing, *area* is a thing and *velocity* is a thing. Furthermore, they are *different* things.
- A **quantity** is something that can be *quantified*, i.e, something that can have a number associated with it. Examples are the distance between Stockholm and Gothenburg, the area of a soccer field and the speed of light.
- A **unit** is a certain magnitude of something. 1 metre is for instance approximatly the length of your arm.

What's the relation between these? A *quantity* has a *dimension*. The number describing the distance between Stockholm and Gothenburg is of the type length. A *quantity* also has a *unit* that relates the number to a known definite distance. The unit of a quantity must describe the dimension of the quantity. It's not possible to describe a distance with joule. However, describing a distance is possible with both metres and inches. Those are two different units describing a quantity of the same dimension.

The dimension of a quantity is often implicitly understood given its unit. If I have a rope of 1 metre, you know it's a length I'm talking about.

There are 7 *base dimensions*, each with a corresponding SI-unit.

- Length (metre)
- Mass (kilogram)
- Time (seconds)
- Electric current (ampere)
- Temperature (kelvin)
- Amount of substance (mole)
- Luminous intensity (candela)

The outline of this chapter is to first introduce dimensions on *value-level* (to print them nicely). Then we'll do dimensions on *type-level* (to only permit legal operations). And finally we'll combine those results to create a data type for quantities.

In science, SI-units are prefered over all other units. Therefore we'll only care about SI-units. Given this decision, we now have a one-to-one correspondence between dimensions and units, which means that only one concept is really needed!

Let's start with value-level dimensions.

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Value-level dimensions

Value-level dimensions

```
module Dimensions.ValueLevel
    ( Dim(..)
    , mul
    , div
    , length
    , mass
    , time
    , current
    , temperature
    , substance
    , luminosity
    , one
    ) where

import Prelude hiding (length, div)
```

From the introduction, two things became apparanent:

1. Given the unit of a quantity, its dimension is known implicitly.
2. If we only work with SI-units, there is a one-to-one correspondence between dimensions and units.

We'll use these facts when implementing dimensions. More precisely, "length" and "metre" will be interchangable, and so on.

A dimension can be seen as a product of the base dimensions, with an individual exponent on each base dimension. Because the 7 base dimensions are known in advance, we can design our data type using this fact.

```
data Dim = Dim Integer -- Length
               Integer -- Mass
               Integer -- Time
               Integer -- Current
               Integer -- Temperature
               Integer -- Substance
               Integer -- Luminosity
  deriving (Eq)
```

Each field denotes the exponent for the corresponding base dimension. If the exponent is `0`, the base dimension is not part of the dimension. Some examples should clarify.

```
length      = Dim 1 0 0 0 0 0 0
mass        = Dim 0 1 0 0 0 0 0
time        = Dim 0 0 1 0 0 0 0
current     = Dim 0 0 0 1 0 0 0
temperature = Dim 0 0 0 0 1 0 0
substance   = Dim 0 0 0 0 0 1 0
luminosity  = Dim 0 0 0 0 0 0 1


velocity     = Dim 1 0 (-1) 0 0 0 0
```

Velocity is m/s or equivalently $m^1 * s^{-1}$. This explains why the exponents are as above.

Noticed how we used "m" (for metre) for implicitly refering to the dimension "length"? It's quite natural to work this way.

Exercise Create values for acceleration, area and charge.

▼ Solution

```
acceleration = Dim 1 0 (-2) 0 0 0 0
area         = Dim 2 0 0    0 0 0 0
charge       = Dim 0 0 1    1 0 0 0
```

Multiplication and division

Dimensions can be multiplied and divided. Velocity is, as we just saw, a division between length and time. Multiplication and division of dimensions are performed as if they were regular numbers, or variables holding numbers, and hence they follow the power laws. That is, to multiply, the exponents of the two numbers are added, and to divide, the exponents are subtracted.

```
mul :: Dim -> Dim -> Dim
(Dim le1 ma1 ti1 cu1 te1 su1 lu1) `mul` (Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
  Dim (le1+le2) (ma1+ma2) (ti1+ti2) (cu1+cu2) (te1+te2) (su1+su2) (lu1+lu2)
```

Exercise Implement a function for dividing two dimensions.

▼ **Solution**

```
div :: Dim -> Dim -> Dim
(Dim le1 ma1 ti1 cu1 te1 su1 lu1) `div` (Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
  Dim (le1-le2) (ma1-ma2) (ti1-ti2) (cu1-cu2) (te1-te2) (su1-su2) (lu1-lu2)
```

It's now possible to construct dimensions in the following fashion.

```
velocity' = length `div` time
area'     = length `mul` length
force     = mass   `mul` acceleration
momentum  = force  `mul` time
```

A dimension we so far haven't mentioned is the *scalar*, which shows up when working with, for example, coefficients of friction. It's dimensionless because it arises from division of two equal dimensions. The case of coefficients of friction looks like

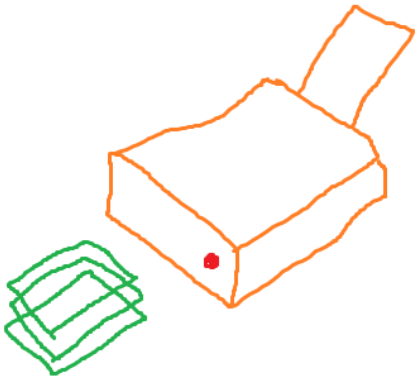
$$F_{friction} = \mu * F_{normal} \iff \mu = \frac{F_{friction}}{F_{normal}}$$

Exercise Create two values, which represent the scalar. They should of course have the same value, but be created in two different ways. One by writing the exponents explicitly. One by dividing two equal dimensions.

▼ Solution

```
one  = Dim 0 0 0 0 0 0 0
one' = force `div` force
```

Pretty-printer



The purpose of value-level dimensions is to be able to print 'em nicely. The pretty printer should be function with the type

```
showDim :: Dim -> String
```

meaning it shows a dimension as a string. But how to actually implement this is not the interesting part of this tutorial. Hence we skip it.

We use `showDim` to make `Dim` an instance of `Show`

```
instance Show Dim where
  show = showDim
```

Now dimensions are printed in a quite pretty way in GHCi

```
ghci> momentum
kg*m/s
```

Note that the SI-unit of the dimensions is used for printing.

The result from this section is the ability to multiply, divide and print dimensions. The next step is to test these operations and see if they actually work.

Learn You a Physics for Great Good!

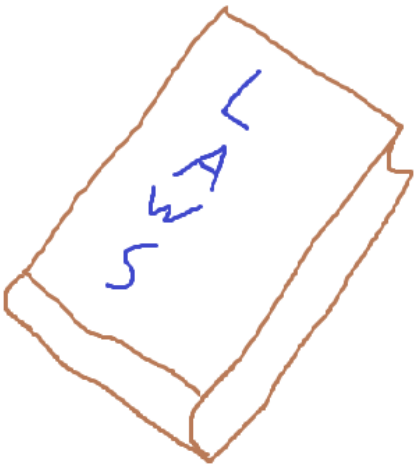
>>> WORK IN PROGRESS <<<

Dimensions / Testing of value-level dimensions

[src: Dimensions/ValueLevel/Test.lhs] Previous: Value-level dimensions | Table of contents | Next: Type-level dimensions

Testing of value-level dimensions

```
module Dimensions.ValueLevel.Test
    ( runTests
    ) where

import Prelude hiding (length, div)
import Test.QuickCheck
import Data.List

import Dimensions.ValueLevel
```



For operations on dimensions, there are a number of laws which should hold. We will here test that the value-level dimensions obey them. One way is to use `QuickCheck`, which produces lots o' random test cases.

Generating arbitrary dimensions

The first thing one needs in order to use `QuickCheck` is an instance of `Arbitrary` for the data type to be used in the tests. In this case it's `Dim`.

An arbitrary example of an `Arbitrary` instance (get it?) could look like

```
data IntPair = IntPair (Int, Int)


genIntPair :: Gen IntPair
genIntPair = do
  first  <- arbitrary
  second <- arbitrary
  return $ IntPair (first, second)


instance Arbitrary IntPair where
  arbitrary = genIntPair
```

Exercise Now try to implement an `Arbitrary` instance of `Dim`.

▼ Solution

> Here's one way to do it.
>
> ```
> genDim :: Gen Dim
> genDim = do
>   le <- arbitrary
>   ma <- arbitrary
>   ti <- arbitrary
>   cu <- arbitrary
>   te <- arbitrary
>   su <- arbitrary
>   lu <- arbitrary
>   return (Dim le ma ti cu te su lu)
>
>
> instance Arbitrary Dim where
>   arbitrary = genDim
> ```

Properties for operations on dimensions

Since dimensions are treated just like regular numbers when it comes to multiplication and division, the laws which ought to hold should be pretty clear. It's the "obvious" laws such as commutativity and so on.

The laws to test are

- Multiplication is commutative
- Multiplication is associative
- one is a unit for multiplication
- Multiplication and division cancel each other out
- Dividing by one does nothing
- Dividing by a division brings up the lowest denominator

$$\frac{x}{\frac{x}{y}} = y$$

- Multiplication by x is the same as dividing by the inverse of x.

The implementation of the first law looks like

```
-- Property: multiplication is commutative
prop_mulCommutative :: Dim -> Dim -> Bool
prop_mulCommutative d1 d2 = d1 `mul` d2 == d2 `mul` d1
```

Excercise. Implement the rest.

▼ **Solution**

Here's what the rest could look like.

```
-- Property: multiplication is associative
prop_mulAssociative :: Dim -> Dim -> Dim -> Bool
prop_mulAssociative d1 d2 d3 = d1 `mul` (d2 `mul` d3) ==
 (d1 `mul` d2) `mul` d3


-- Property: `one` is a unit for multiplication
prop_mulOneUnit :: Dim -> Bool
prop_mulOneUnit d = d == one `mul` d


-- Property: multiplication and division cancel each other out
prop_mulDivCancel :: Dim -> Dim -> Bool
prop_mulDivCancel d1 d2 = (d1 `mul` d2) `div` d1 == d2
```

```haskell
-- Property: dividing by `one` does noting
prop_divOne :: Dim -> Bool
prop_divOne d = d `div` one == d


-- Property: dividing by a division brings up the lowest denominator
prop_divTwice :: Dim -> Dim -> Bool
prop_divTwice d1 d2 = d1 `div` (d1 `div` d2) == d2


-- Property: multiplication same as division by inverse
prop_mulDivInv :: Dim -> Dim -> Bool
prop_mulDivInv d1 d2 = d1 `mul` d2 ==
  d1 `div` (one `div` d2)
```

We should also test the pretty-printer. But just like how that function itself is implemented isn't interesting, neither is the code testing it. We therefore leave it out.

From this module, we export a function `runTests`. That function runs all the tests implemented here and is used with Stack.

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

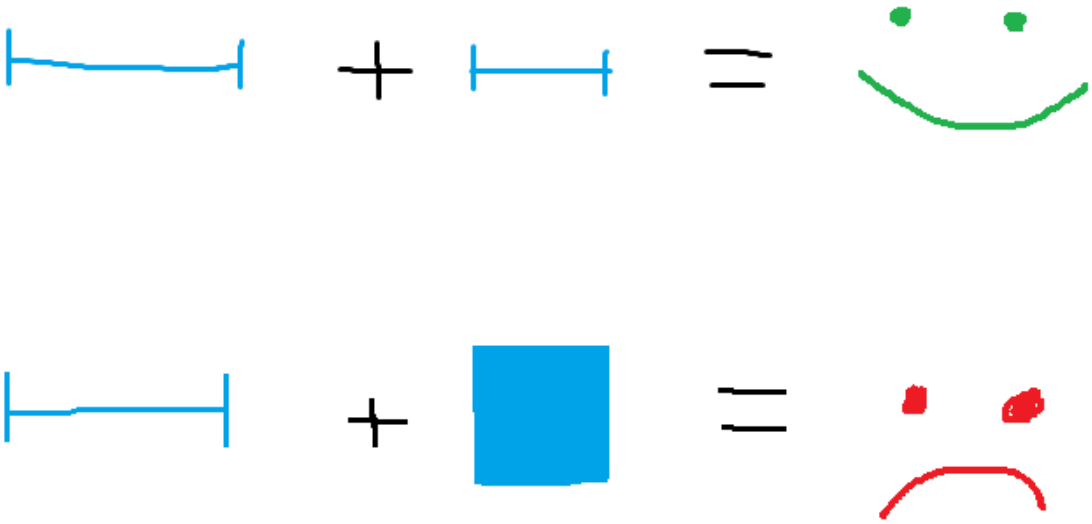
Dimensions / Type-level dimensions

Type-level dimensions

```
module Dimensions.TypeLevel
    ( Dim(..)
    , Mul
    , Div
    , Length
    , Mass
    , Time
    , Current
    , Temperature
    , Substance
    , Luminosity
    , One
    ) where
```

We will now implement *type-level* dimensions. What is type-level? Programs (in Haskell) normally operatate on (e.g. add) values (e.g. `1` and `2`). This is on *value-level*. Now we'll do the same thing but on *type-level*, that is, perform operations on types.

What's the purpose of type-level dimensions? It's so we'll notice as soon as compile-time if we've written something incorrect. E.g. adding a length and an area is not allowed since they have different dimensions.



This implemention is very similar to the value-level one. It would be possible to only have one implementation by using `Data.Proxy`. But it would be trickier. This way is lengthier but easier to understand.

To be able to do type-level programming, we'll need a nice stash of GHC-extensions.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeOperators #-}
```

See the end of the next chapter to read what they do.

We'll need to be able to operate on integers on the type-level. Instead of implementing it ourselves, we will just import the machinery so we can focus on the physics-part.

```
import Numeric.NumType.DK.Integers
```

We make a *kind* for dimensions, just like we in the previous section made *type* for dimensions. On value-level we made a *type* with *values*. Now we make a *kind* with *types*. The meaning is exactly the same, except we have moved "one step up".

```haskell
data Dim = Dim TypeInt -- Length
               TypeInt -- Mass
               TypeInt -- Time
               TypeInt -- Current
               TypeInt -- Temperature
               TypeInt -- Substance
               TypeInt -- Luminosity
```

But `data Dim = ...` looks awfully similar to a regular data type! That's correct. But with the GHC-extension `DataKinds` this will, apart from creating a regular data type, **also** create a *kind*. Perhaps a less confusing syntax would've been `kind Dim = ...`. The above definition can be seen as the two following definitions.

```haskell
-- LHS: Type
-- RHS: Value
data Dim = Dim TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt


-- LHS: Kind
-- RHS: Type
kind Dim = Dim TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
               TypeInt
```

Thanks to the `Dim`-kind we can force certain types in functions to be of this kind.

This may sound confusing, but the point of this will become clear over time. Let's show some example *types* of the `Dim`-kind.

```haskell
type Length      = 'Dim Pos1 Zero Zero Zero Zero Zero Zero
type Mass        = 'Dim Zero Pos1 Zero Zero Zero Zero Zero
type Time        = 'Dim Zero Zero Pos1 Zero Zero Zero Zero
type Current     = 'Dim Zero Zero Zero Pos1 Zero Zero Zero
type Temperature = 'Dim Zero Zero Zero Zero Pos1 Zero Zero
```

```
type Substance   = 'Dim Zero Zero Zero Zero Zero Pos1 Zero
type Luminosity  = 'Dim Zero Zero Zero Zero Zero Zero Pos1
```

`'Dim` is used to distinguish between the *type* `Dim` (left-hand-side of the `data Dim` definition) and the *type constructor* `Dim` (right-hand-side of the `data Dim` definition, with `DataKinds`-perspective). `'Dim` refers to the type constructor. Both are created when using `DataKinds`.

`Pos1`, `Neg1` and so on corresponds to `1` and `-1` in the imported package, which operates on type-level integers.

Exercise Create types for velocity, acceleration and the scalar.

▼ Solution

```
type Velocity     = 'Dim Pos1 Zero Neg1 Zero Zero Zero Zero
type Acceleration = 'Dim Pos1 Zero Neg2 Zero Zero Zero Zero


type One = 'Dim Zero Zero Zero Zero Zero Zero Zero
```

Multiplication and division

Let's implement multiplication and division on the type-level. After such an operation a new dimension is created. And from the previous section we already know what the dimension should look like. To translate to Haskell-language: "after such an operation a new *type* is created". How does one implement that? With `type family`! A `type family` can easiest be thought of as a function on the type-level.

```
type family Mul (d1 :: Dim) (d2 :: Dim) where
  Mul ('Dim le1 ma1 ti1 cu1 te1 su1 lu1)
      ('Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
      'Dim (le1+le2) (ma1+ma2) (ti1+ti2) (cu1+cu2)
        (te1+te2) (su1+su2) (lu1+lu2)
```

- `type family` means it's a function on type-level.
- `Mul` is the name of the function.
- `d1 :: Dim` is read as "the *type* `d1` has *kind* `Dim`".

Exercise As you would suspect, division is very similar, so why don't you try 'n implement it yourself?

▼ Solution

```
type family Div (d1 :: Dim) (d2 :: Dim) where
  Div ('Dim le1 ma1 ti1 cu1 te1 su1 lu1)
      ('Dim le2 ma2 ti2 cu2 te2 su2 lu2) =
      'Dim (le1-le2) (ma1-ma2) (ti1-ti2) (cu1-cu2)
        (te1-te2) (su1-su2) (lu1-lu2)
```

Exercise Implement a type-level function for raising a dimension to the power of some integer.

▼ Solution

```
type family Power (d :: Dim) (n :: TypeInt) where
  Power ('Dim le ma ti cu te su lu) n =
    'Dim (le*n) (ma*n) (ti*n) (cu*n) (te*n) (su*n) (lu*n)
```

Now types for dimensions can be created by combining exisiting types, much like we did for values in the previous chapter.

Exercise Create types for velocity, area, force and impulse.

▼ Solution

```
type Velocity' = Length `Div` Time
type Area      = Length `Mul` Length
type Force     = Mass   `Mul` Length
type Impulse   = Force  `Mul` Time
```

Perhaps not very exiting so far. But just wait 'til we create a data type for quantities. Then the strenghts of type-level dimensions will be clearer.

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Quantities

[src: Dimensions/Quantity.lhs] Previous: Type-level dimensions Table of contents Next: Testing of Quantities

Quantities

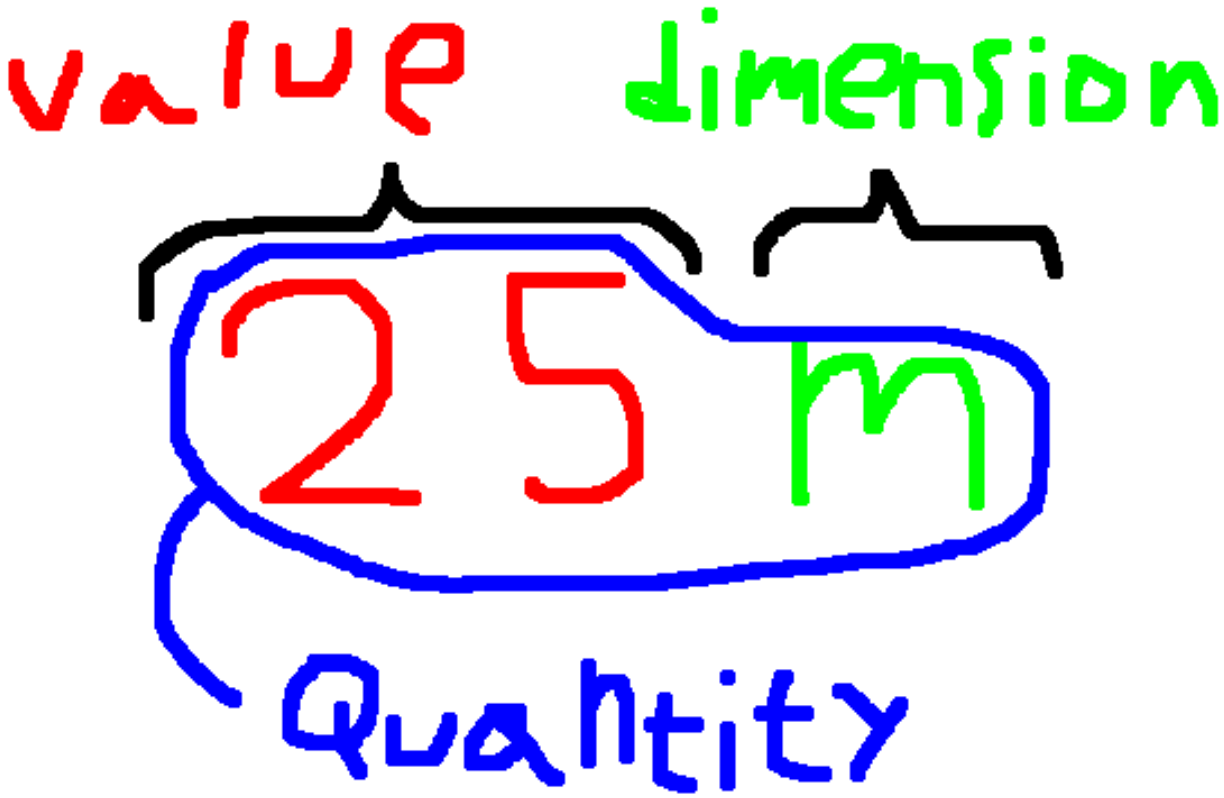
```
module Dimensions.Quantity
    ( Quantity
    , length, mass, time, current, temperature, substance, luminosity, one
    , velocity, acceleration, force, momentum
    , meter, kilogram, second, ampere, kelvin, mole, candela, unitless
    , (~=)
    , isZero
    , (#)
    , (+#), (-#), (*#), (/#)
    , sinq, cosq, asinq, acosq, atanq, expq, logq
    ) where


import qualified Dimensions.ValueLevel as V
import           Dimensions.TypeLevel  as T
import           Prelude               as P hiding (length)
```

We'll now create a data type for quantities and combine dimensions on value-level and type-level. Just as before, a bunch of GHC-extensions are necessary.

```
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
```

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE KindSignatures #-}
```

So what exactly does a physical “quantity” contain? Have a look at this picture



This is what quantities look like in physics calculations. They have a *numerical value* and a *dimension* (which is often given by instead writing its *unit*). The whole thing combined is the *quantity*. This combined thing is what we want to have a data type for in Haskell.

It’s evident the data type should have a numerical value and a dimension. But so far we have created two dimensions! Which should we use? Both! The value-level dimension of the quantity is used to print it nicely. The type-level dimension of the quantity is to get some type-safety. Just like this

$$25m + 7kg = \ldots$$

should upset you, this (in pseudo-Haskell)

```
ghci> let q1 = 25 m -- A value of the quantity type
ghci> let q2 = 7 kg -- Another value of the quantity type
ghci> q1 + q2
...
```

should upset the compiler!

We can't cover all everything at once, but we guarantee you that by the end of this chaper, you'll know exactly how the above ideas are actually implemented.

Let's get on to the actual data type declaration.

```
data Quantity (d :: T.Dim) (v :: *) where
  ValQuantity :: V.Dim -> v -> Quantity d v
```

That was sure a mouthful! Let's break it down. `data Quantity (d :: T.Dim) (v :: *)` creates the *type constructor* `Quantity`. A type constructor takes types to create another type. In this case, the type constructor `Quantity` takes a type `d` of *kind* `T.Dim` and a type `v` of *kind* `*` to create the type `Quantity d v`. Let's see it in action

```
type ExampleType = Quantity T.Length Double
```

`ExampleType` is the type representing quantites of the physical dimension length and where the numerical value is of the type `Double`.

How do we create values of this type? That's where the second row comes in! `ValQuantity` is a *value constructor*, which means it takes values to create another value. In this case, the value constructor `ValQuantity` takes a value of *type* `V.Dim` and `v`. Let's see this one in action as well

```
exampleValue = ValQuantity V.length 25.0
```

`exampleValue` is a value for the quantity representing a length with the numerical value 25.0 .

We can combine the two and write this

```
exampleValue :: ExampleType
```

or

```
exampleValue :: Quantity T.Length Double
exampleValue = ValQuantity V.length 25.0
```

to get the full picture. So, here we are, with a way to create quantity values (`ValQuantity`) of the type `Quantity t1 t2` where `t1` is the type-level dimension of the quantity and `t2` the type of the numerical value. If you still find it confusing, don't worry! Over the course of this chapter, it'll become more concrete as we work more with `Quantity`.

Also note that the type-level dimension and value-level dimension are intented to match when used with `Quantity`! So far nothing enforces this. We'll tackle this problem later.

Exercise create `Quantity` values for 2.5 meters, 6.7 seconds and 9 mol. Recall that the three mentioned SI-units here, and all SI-units in this tutorial in general, have a one-to-one correspondence with their respective dimension.

▼ Solution

```
theDistance :: Quantity T.Length Double
theDistance = ValQuantity V.length 2.5


theTime :: Quantity T.Time Double
theTime = ValQuantity V.time 6.7


theAmountOfSubstance :: Quantity T.Substance Integer
theAmountOfSubstance = ValQuantity V.substance 9
```

Exercise Create a data type which has two type-level dimensions, always use `Rational` as the value-holding type (the role of `v`) but which has no value-level dimensions. It should have three numerical values. Create some values of that type.

▼ Solution

```
data Quantity' (d1 :: T.Dim) (d2 :: T.Dim) where
  ValQuantity' :: Rational -> Rational -> Rational -> Quantity' d1 d2


val1 :: Quantity' T.Luminosity T.Mass
val1 = ValQuantity' 8 7 3
```

```
val2 :: Quantity' T.Substance T.Temperature
val2 = ValQuantity' 2 3 1
```

Pretty-printer

Let's do a pretty-printer for quantities. Most of the work is already done by the value-level dimensions.

```
showQuantity :: (Show v) => Quantity d v -> String
showQuantity (ValQuantity d v) = show v ++ " " ++ show d


instance (Show v) => Show (Quantity d v) where
  show = showQuantity
```

Exercise In a previous exercise, you created some example values of the `Quantity` type. Write them in GHCi and see how they look.

▼ Solution

```
ghci> theDistance
2.5 m
ghci> theTime
6.7 s
ghci> theAmountOfSubstance
9 mol
```

Pretty, huh?

A taste of typos

We'll implement all arithmetic operations on `Quantity`, but for now, to get a taste of types, we show here addition and multiplication and some examples of values of type `Quantity`.

```
quantityAdd :: (Num v) => Quantity d v ->
                          Quantity d v ->
                          Quantity d v
quantityAdd (ValQuantity d v1) (ValQuantity _ v2) = ValQuantity d (v1+v2)
```

The type is interpreted as follows: two values of type `Quantity d v` is the input, where `d` is the type-level dimension. The output is also a value of type `Quantity d v`.

The type of the function forces the inputs to have the same dimensions. For this reason, the dimension on value-level doesn't matter on one of the arguments, because they will be the same. As was already mentioned when the `Quantity` type was created, it's possible to create values where the dimensions on value-level and type-level don't match. Just like then, we ignore this problem for now and fix it later!

Multiplication is implemented as

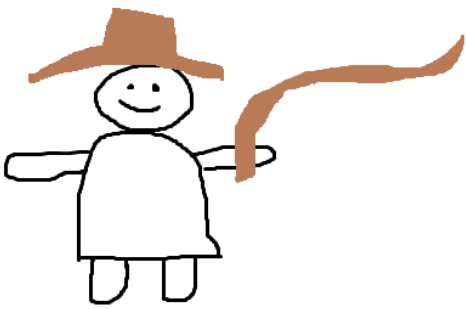
```
quantityMul :: (Num v) => Quantity d1 v ->
                          Quantity d2 v ->
                          Quantity (d1 `Mul` d2) v
quantityMul (ValQuantity d1 v1) (ValQuantity d2 v2) =
  ValQuantity (d1 `V.mul` d2) (v1*v2)
```

The type has the following interpretation: two values of type `Quantity dx v` is input, where `dx` are two types representing (potentially different) dimensions. As output, a value of type `Quantity` is returned. The type in the `Quantity` will be the type that is the product of the two dimensions in.

Exercise Implement subtraction and division.

▼ **Solution**

> Hold on cowboy! Not so fast. We'll come back later to subtraction and division, so check your solution then.



Now on to some example values and types.

```
width :: Quantity T.Length Double
width = ValQuantity V.length 0.5


height :: Quantity T.Length Double
height = ValQuantity V.length 0.3


type Area = Mul T.Length T.Length
```

The following example shows that during a multiplication, the types will change, as they should. The dimensions change not only on value-level but also on type-level.

```
area :: Quantity Area Double
area = quantityMul width height


ghci> width
0.5 m
ghci> :t width
width :: Quantity Length Double
ghci> area
0.15 m^2
ghci> :i area
area :: Quantity Area Double
```

(Try out `:t` instead of `:i` on the last one and see what happens!)

The type-level dimensions are used below to enforce, at compile-time, that only allowed operations are attempted.

```
-- Doesn't even compile
weird = quantityAdd width area
```

If the dimensions had been value-level only, this error would go undetected until run-time.

Exercise What is the volume of a cuboid with sides $1.2\ m$, $0.4\ m$ and $1.9\ m$? Create values for the involved quantities. Use `Float` instead of `Double`.

▼ Solution

```
side1 :: Quantity Length Float
side1 = ValQuantity V.length 1.2
```

```haskell
side2 :: Quantity Length Float
side2 = ValQuantity V.length 0.4


side3 :: Quantity Length Float
side3 = ValQuantity V.length 1.9


type Volume = Length `Mul` (Length `Mul` Length)


volume :: Quantity Volume Float
volume = quantityMul side1 (quantityMul side2 side3)
```

Comparsions

It's useful to be able to compare quantities. Perhaps one wants to know which of two amounts of energy is the largest. But what's the largest of `1 J` and `1 m`? That's no meaningful comparsion since the dimensions don't match. This behaviour is prevented by having type-level dimensions.

```haskell
quantityEq :: (Eq v) => Quantity d v -> Quantity d v -> Bool
quantityEq (ValQuantity _ v1) (ValQuantity _ v2) = v1 == v2


instance (Eq v) => Eq (Quantity d v) where
  (==) = quantityEq
```

Exercise Make `Quantity` an instance of `Ord`.

▼ Solution

```haskell
quantityCompare :: (Ord v) => Quantity d v ->
                              Quantity d v -> Ordering
quantityCompare (ValQuantity _ v1) (ValQuantity _ v2) =
  compare v1 v2


instance (Ord v) => Ord (Quantity d v) where
  compare = quantityCompare
```

We often use `Double` as the value holding type. Doing exact comparsions isn't always possible to due rounding errors. Therefore, we'll create a `~=` function for testing if two quantities are almost equal.

```
infixl 4 ~=
(~=) :: Quantity d Double -> Quantity d Double -> Bool
(ValQuantity _ v1) ~= (ValQuantity _ v2) = abs (v1-v2) < 0.001
```

Testing if a quantity is zero is something which might be a common operation. So we define it here.

```
isZero :: (Fractional v, Ord v) => Quantity d v -> Bool
isZero (ValQuantity _ v) = abs v < 0.001
```

Arithmetic on quantities

Let's implement the arithmetic operations on `Quantity`. Basically it's all about creating functions with the correct type-level dimensions.

```
infixl 6 +#
(+#) :: (Num v) => Quantity d v -> Quantity d v ->
                   Quantity d v
(+#) = quantityAdd


infixl 6 -#
(-#) :: (Num v) => Quantity d v -> Quantity d v ->
                   Quantity d v
(ValQuantity d v1) -# (ValQuantity _ v2) = ValQuantity d (v1-v2)


infixl 7 *#
(*#) :: (Num v) => Quantity d1 v -> Quantity d2 v ->
                   Quantity (d1 `Mul` d2) v
(*#) = quantityMul


infixl 7 /#
(/#) :: (Fractional  v) => Quantity d1 v ->
                           Quantity d2 v ->
```

```
                          Quantity (d1 `Div` d2) v
(ValQuantity d1 v1) /# (ValQuantity d2 v2) =
  ValQuantity (d1 `V.div` d2) (v1 / v2)
```

For all operations on quantities, one does the operation on the value and, in the case of multiplication and division, on the dimensions separetly. For addition and subtraction the in-dimensions must be the same. Nothing then happens with the dimension.

How does one perform operations such as `sin` on a quantity with a *potential* dimension? The answer is that the quantity must be dimensionless, and with the dimension nothing happens.

Why is that, one might wonder. Every function can be written as a power series. For sin the series looks like

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

The dimension of *every* term must be the same. The only way for that to be case is if the input x is dimensionless. Then so will the output.

The other functions can be written as similar power series and we'll see on those too that the input and output must be dimensionless.

```
sinq :: (Floating v) => Quantity One v -> Quantity One v
sinq (ValQuantity dl v) = ValQuantity d1 (sin v)

cosq :: (Floating v) => Quantity One v -> Quantity One v
cosq (ValQuantity dl v) = ValQuantity d1 (cos v)
```

We quickly realize a pattern, so let's generalize a bit.

```
qmap :: (a -> b) -> Quantity One a -> Quantity One b
qmap f (ValQuantity d1 v) = ValQuantity d1 (f v)


qmap' :: (a -> b) -> Quantity dim a -> Quantity dim b
qmap' f (ValQuantity d v) = ValQuantity d (f v)


qfold :: (a -> a -> b) -> Quantity dim a -> Quantity dim a -> Quantity dim b
qfold f (ValQuantity d v1) (ValQuantity _ v2) = ValQuantity d (f v1 v2)
```

```
sinq, cosq, asinq, acosq, atanq, expq, logq :: (Floating v) =>
  Quantity One v -> Quantity One v
sinq  = qmap sin
cosq  = qmap cos
asinq = qmap asin
acosq = qmap acos
atanq = qmap atan
expq  = qmap exp
logq  = qmap log
```

Why not make `Quantity` an instance of `Num`, `Fractional`, `Floating` och `Functor`? The reason is that the functions of those type classes have the following type

```
(*) :: (Num a) => a -> a -> a
```

which isn't compatible with `Quantity` since multiplication with `Quantity` has the following type

```
(*#) :: (Num v) => Quantity d1 v -> Quantity d2 v ->
                   Quantity (d1 `Mul` d2) v
```

The input here may actually be of *different* types, and the output has a type depending on the types of the input. However, the *kind* of the inputs and output are the same, namely `Quantity`. We'll just have to live with not being able to make `Quantity` a `Num`-instance.

However, operations with only scalars (type `One`) has types compatible with `Num`.

Exercise `Quantity One` has compatible types. Make it an instance of `Num`, `Fractional`, `Floating` and `Functor`.

▼ Solution

```
instance (Num v) => Num (Quantity One v) where
  (+) = (+#)
  (-) = (-#)
  (*) = (*#)
  abs = qmap abs
  signum = qmap signum
  fromInteger n = ValQuantity V.one (fromInteger n)
```

```haskell
instance (Fractional v) => Fractional (Quantity One v) where
  (/) = (/#)
  fromRational r = ValQuantity V.one (fromRational r)


instance (Floating v) => Floating (Quantity One v) where
  pi    = ValQuantity V.one pi
  exp   = expq
  log   = logq
  sin   = sinq
  cos   = cosq
  asin  = asinq
  acos  = acosq
  atan  = atanq
  sinh  = qmap sinh
  cosh  = qmap cosh
  asinh = qmap asinh
  acosh = qmap acosh
  atanh = qmap atanh


instance Functor (Quantity One) where
  fmap = qmap
```

Syntactic sugar

In order to create a value representing a certain distance (5 metres, for example) one does the following

```haskell
distance :: Quantity T.Length Double
distance = ValQuantity V.length 5.0
```

Writing that way each time is very clumsy. You can also do “dumb” things such as

```haskell
distance :: Quantity T.Length Double
distance = ValQuantity V.time 5.0
```

with different dimensions on value-level and type-level.

To solve these two problems we’ll introduce some syntactic sugar. First some pre-made values for the 7 base dimensions and the scalar.

```
length :: (Num v) => Quantity Length v
length = ValQuantity V.length 1


mass :: (Num v) => Quantity Mass v
mass = ValQuantity V.mass 1


time :: (Num v) => Quantity Time v
time = ValQuantity V.time 1
```

Exercise Do the rest.

▼ **Solution**

```
current :: (Num v) => Quantity Current v
current = ValQuantity V.current 1


temperature :: (Num v) => Quantity Temperature v
temperature = ValQuantity V.temperature 1


substance :: (Num v) => Quantity Substance v
substance = ValQuantity V.substance 1


luminosity :: (Num v) => Quantity Luminosity v
luminosity = ValQuantity V.luminosity 1


one :: (Num v) => Quantity One v
one = ValQuantity V.one 1
```

And now the sugar.

```
(#) :: (Num v) => v -> Quantity d v -> Quantity d v
v # (ValQuantity d bv) = ValQuantity d (v*bv)
```

The intended usage of the function is the following

```
ghci> let myDistance = 5 # length
ghci> :t myDistance
```

```
t :: Num v => Quantity Length v
ghci> myDistance
5 m
```

To create a `Quantity` with a certain value (here `5`) and a certain dimension (here `length`), you write as above and get the correct dimension on both value-level and type-level.

This way of writing in Haskell is similiar to what you do in traditional physics.

Both mean that “now we get 5 pieces of the SI-unit meters”. This is signified in the implementation by `b*bv` where `bv` stands for “base value”. The base value is the SI-unit, which is 1. Since “length” and “meter” are equivalent in our implementation, `length` *means* “meter”.

But let’s not stop there. It would be prettier if you could actually write `meter` instead of `length`. In fact, not much code is needed for this!

Exercise Make it so that one can write the SI-units instead of the base dimensions when one uses the sugar. Then show how to write 4 seconds.

▼ **Solution**


```
meter    = length
kilogram = mass
second   = time
ampere   = current
kelvin   = temperature
mole     = substance
candela  = luminosity
unitless = one


fourSeconds = 4 # second
```

That's nice and all, but we can actually take this way of thinking even further. The sugar-function `#` multiplies with the *base* value of a unit. Thanks to this, we can actually support more units than just the SI-units! (At least for input.)

Exercise Make it so that one can write units such as inch, foot and pound in the same way one can write the SI-units.

▼ Hint

> `x # unit` multiplies `x`, how many pieces of the unit you want, with the base value of `unit`. So you want to think about how much of the corresponding SI-unit a unit corresponds to.

▼ Solution

> We just need to create pre-made values for the units we want to support.

```
inch = 0.0254 # meter
foot = 0.3048 # meter
pound = 0.45359237 # kilogram
```

All right, so now we have tackled the problem of writing quantities easier than before. What about the second problem?

We want to maintain the invariant that the dimension on value-level and type-level always match. The pre-made values from above maintain it, and so do the arithmetic operations we previously created. Therefore, we only export those from this module! The user will have no choice but to use these constructs and hence the invariant will be maintained.

If the user needs other dimensions than the base dimensions (which it probably will), the follwing example shows how it's done.

```
ghci> let myLength = 5 # length
ghci> let myTime = 2 # time
ghci> let myVelocity = myLength /# myTime
ghci> myVelocity
2.5 m/s
```

New dimensions are created "on demand". Furthermore

```
ghci> let velocity = length /# time
ghci> let otherVelocity = 188 # velocity
```

it's possible to use the sugar from before on user-defined dimensions.

Just for convenience sake we'll define a bunch of common composite dimensions. Erm, *you'll* define.

Exericse Define pre-made values for velocity, acceleration, force, momentum and energy.

▼ Solution

```
velocity     = length   /# time
acceleration = velocity /# time
force        = mass     *# acceleration
momentum     = force    *# time
energy       = force    *# length
```

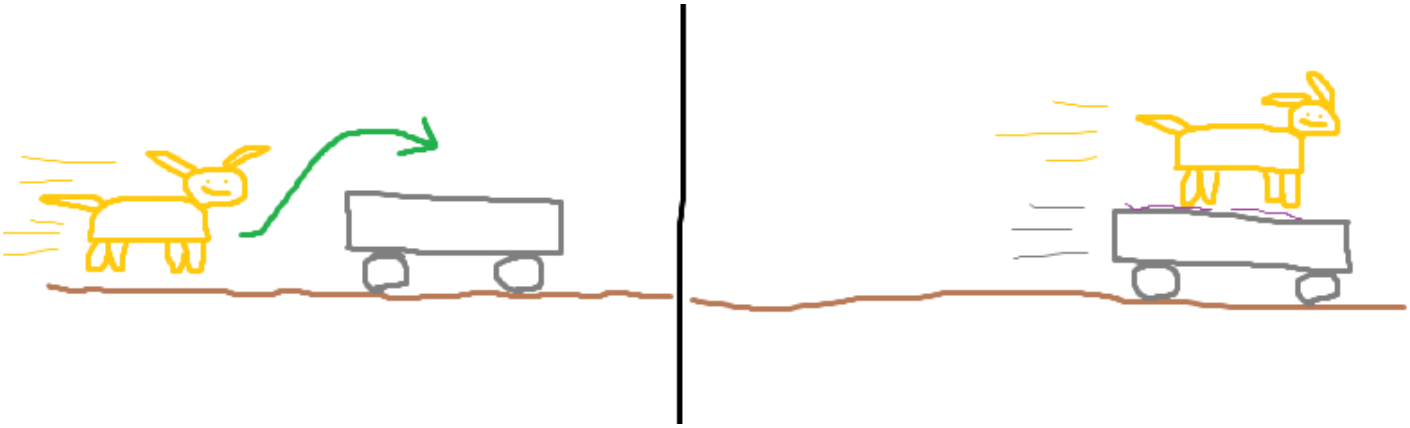
A physics example

To conclude this chapter, we show an example on how to code an exercise and its solution in this domain specific language we've created for quantities.

The code comments show what GHCi prints for that line.

The exercise is "A dog runs, jumps and lands sliding on a carriage. The dog weighs $40\ kg$ and runs in $8\ m/s$. The carriage weighs $190\ kg$. The coefficient of friction between the dog's paws and the carriage is 0.7."

This is illustrated in the painting below.



a. Calculate the (shared) final velocity of the dog and the carriage.

```
mDog      = 40 # mass     -- 40 kg
viDog     = 8 # velocity -- 8 m/s
mCarriage = 190 # mass    -- 190 kg
u         = 0.7 # one     -- 0.7
```

No external forces are acting on the dog and the carriage. Hence the momentum of the system is preserved.

```
piSystem = mDog *# viDog -- 320 kg*m/s
pfSystem = piSystem      -- 320 kg*m/s
```

In the end the whole system has a shared velocity of its shared mass.

```
mSystem  = mDog +# mCarriage   -- 230 kg
vfSystem = pfSystem /# mSystem -- 1.39 m/s
```

b. Calculate the force of friction acting on the dog.

```
  fFriction = u *# fNormal        -- 275 kg*m/s^2
  fNormal   = mDog *# g           -- 393 kg*m/s^2
  g         = 9.82 # acceleration -- 9.82 m/s^2
```

c. For how long time does the dog slide on the carriage?

```
  aDog = fFriction /# mDog -- 6.87 m/s^2


  vDelta = mDog -# vfSystem
```

```
* Couldn't match type 'Numeric.NumType.DK.Integers.Neg1
                  with 'Numeric.NumType.DK.Integers.Zero
...
```

Whoops! That's not a good operation. Luckily the compiler caught it.

```
  vDelta = viDog -# vfSystem -- 6.60 m/s
  tSlide = vDelta /# aDog    -- 0.96 s
```

Conclusion

Okay, so you've read a whole lot of text by now. But in order to really learn anything, you need to practise with some additional exercises. Some suggestions are

- Implement first value-level dimensions, then type-level dimensions and last quantites by yourself, without looking too much at this text.
- Implement a power function.
- Implement a square-root function. The regular square-root function in Haskell uses `exp` and `log`, which only work on `Quantity One`. But taking the the square-root of e.g. an area should be possible.
- Extending the `Quantity` data type here by adding support for prefixes and non-SI-units.
- Ditching the separate implementations of value-level and type-level, and only use one with `Data.Proxy`.

After reading this text and doing some exercies, we hope you've learnt

- The relation between dimensions, quantities and units.

- How dimensions…
  - …restrict operations such as comparsion to quantities of the same dimension.
  - …change after operations such as multiplication.
  - …can be implemented on the type-level in Haskell to enforce the two above at compile-time.
- The basics about kinds and type-level programming in Haskell.

Further reading

The package Dimensional inspired a lot of what we did here. Our `Quantity` is like a "lite" version of Dimensional, which among other things, support different units and use `Data.Proxy`.

The type-level progamming in this text was done with the help of the tutorial Basic Type Level Programming in Haskell. If you want to know more about kinds and type-level programming, it's a very good starting point.

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Testing of Quantities

[src: Dimensions/Quantity/Test.lhs] Previous: Quantities Table of contents Next: Usage

Testing of Quantity

```
module Dimensions.Quantity.Test
( runTests
)
where


{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}


import Prelude hiding (length, div)
import Test.QuickCheck


import Dimensions.TypeLevel
import Dimensions.Quantity
```

`Quantity` consists of a numerical type such as `Double` and a value-level dimension. Let's assume Haskell's implementation of numbers abide the laws. We also know the value-

level dimensions abide the laws since since we tested them in a previous section. So what's there to test now? Well, it's still useful to test the top-level construct. And error may arise when combing different parts.

So let's test *some* things. Due to the type-level dimensions it's impossible to easily and readably design test for all combinations of dimensions. Hence it'll have to suffice with only some combinations.

Generating arbitrary quantities

First we need an `Arbitrary` instance for `Quantity d val`. For `d` we'll mostly use `One` and for `val` we'll exclusively use `Double`.

```
type Q d = Quantity d Double
```

A generator for an arbitrary dimension.

```
genQuantity :: Quantity d Double -> Gen (Q d)
genQuantity quantity = do
  value <- arbitrary
  return (value # quantity)
```

And now we make `Arbitrary` instances of arbitrary selected dimensions in the `Quantity`s.

```
instance Arbitrary (Q One) where
 arbitrary = genQuantity one


instance Arbitrary (Q Length) where
  arbitrary = genQuantity length


instance Arbitrary (Q Mass) where
  arbitrary = genQuantity mass


instance Arbitrary (Q Time) where
  arbitrary = genQuantity time
```

Testing arithmetic properties

On regular numbers, and hence too on quantites with their dimensions, a bunch of properties should hold. The things we test here are

- Addition commutative
- Addition associative
- Zero is identity for addition
- Multiplication commutative
- Multiplication associative
- One is identity for multiplication
- Addition distributes over multiplication
- Subtraction and addition cancel each other out
- Division and multiplication cancel each other out
- Pythagoran trigonometric identity

Let's start!

We could write the type signatures in a general way like

```
prop_addCom :: Q d -> Q d -> Bool
```

But we won't do that since QuickCheck needs concrete types in order to work. So we would have to do a bunch of specialization anyway. And even if we begin with a general signature, we can't cover all cases since there are infinitly many dimensions.

Instead we'll pick some arbitrary dimensions that have an `Arbitrary` instance.

```
-- a + b = b + a
prop_addCom :: Q Length -> Q Length -> Bool
prop_addCom a b = (a +# b) ~= (b +# a)


-- a + (b + c) = (a + b) + c
prop_addAss :: Q Mass -> Q Mass -> Q Mass -> Bool
prop_addAss a b c = a +# (b +# c) ~= (a +# b) +# c


-- 0 + a = a
prop_addId :: Q Time -> Bool
prop_addId a = zero +# a ~= a
  where
    zero = 0 # a
```

```haskell
-- a * b = b * a
prop_mulCom :: Q Length -> Q Mass -> Bool
prop_mulCom a b = a *# b ~= b *# a


-- a * (b * c) = (a * b) * c
prop_mulAss :: Q Time -> Q Length -> Q Mass -> Bool
prop_mulAss a b c = a *# (b *# c) ~=
                    (a *# b) *# c


-- 1 * a = a
prop_mulId :: Q Time -> Bool
prop_mulId a = (1 # one) *# a ~= a


-- a * (b + c) = a * b + a * c
prop_addDistOverMul :: Q Length -> Q Mass -> Q Mass -> Bool
prop_addDistOverMul a b c = a *# (b +# c) ~=
                            a *# b +# a *# c


-- (a + b) - b = a
prop_addSubCancel :: Q Length -> Q Length -> Bool
prop_addSubCancel a b = (a +# b) -# b ~= a


-- (a * b) / b = a
prop_mulDivCancel :: Q Time -> Q Length -> Property
prop_mulDivCancel a b = not (isZero b) ==>
  (a *# b) /# b ~= a


-- sin a * sin a + cos a * cos a = 1
prop_pythagoranIdentity :: Q One -> Bool
prop_pythagoranIdentity a = sinq a *# sinq a +#
                            cosq a *# cosq a ~= (1 # one)
```

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Dimensions / Usage

[src: Dimensions/Usage.lhs] Previous: Testing of Quantities Table of contents Next: Single particle mechanics

Usage

```
module Dimensions.Usage
(
)
where
```

In this module we'll show how to use value-level dimensions, type-level dimensions and `Quantity` in an "actual" progam. Let's first use this fancy GHC-extension

```
{-# LANGUAGE TypeOperators #-}
```

and then import the things we have created.

```
import Dimensions.TypeLevel
import Dimensions.Quantity
import Prelude hiding (length)
```

We don't need to import `Dimensions.ValueLevel` since all its features are available by using the `Quantity` data type. The same thing could be said about `Dimensions.TypeLevel`. So why do we import it? It's so we can write explicit type signatures, the trick of trade in Haskell progamming.

Values and types

Let's create a length, time and mass, in the way hinted in the previous chapter.

```
myLength = 10.0 # length
myTime   = 20.0 # time
myMass   = 30.0 # mass
```

Okay, let's check out their type and value in GHCi.

```
ghci> myLength
10.0 m
ghci> :t myLength
myLength :: Quantity Length Double
ghci> myTime
20.0 s
ghci> :t myTime
myTime :: Quantity Time Double
```

The pretty printing works thanks to dimensions on *value-level* and the Haskell-types thanks to the *type-level* dimensions. Recall that the value-level dimensions just is a pretty simple "average" data type.

Let's write some type signatures on those values (of type `Quantity`).

```
myLength :: Quantity Length Double
myLength = 10.0 # length
myTime :: Quantity Time Double
myTime = 20.0 # time
myMass :: Quantity Mass Double
myMass = 30.0 # mass
```

Neat! `Length`, `Time` and `Mass` are type-level dimensions. Therefore, `Length` *is* a type, just like `String` is a type.

Remember that the sugary style of writing `10.0 # length` is a shorthand for doing

```
myLength :: Quantity Length Double
myLength = Quantity length' 10.0
```

where `length'` refers to `length` in `Dimensions.ValueLevel`.

The reasons for not allowing this “raw” way of creating the value is to maintain the invariant of having the same value-level and type-level dimension. With the `#` function, the are *forced* by it to be the same. The following try to work around it is not possible.

```
myWeird :: Quantity Length Double
myWeird = 10.0 # time
```

```
* Couldn't match type 'Numeric.NumType.DK.Integers.Zero
                  with 'Numeric.NumType.DK.Integers.Pos1
  Expected type: Quantity Length Double
    Actual type: Quantity Time Double
* In the expression: 10.0 # time
  In an equation for `myWeird': myWeird = 10.0 # time
```

Custom values and types

Creating values of a dimensions *not* exported is also possible. Let’s take energy as an example.

```
energy = force *# length
```

`energy` can now be used just as `length` and so on.

```
energyToBoilMyGlassOfWater = 12 # energy
```

In order to write explicit type signatures, we can do like this

```
type Velocity = Length `Div` Time
type Acceleration = Velocity `Div` Time
type Force = Acceleration `Mul` Mass
type Energy = Force `Mul` Length
```

Note we had to introduce some helper types since, unlike the pre-made `Quantity` values, not as many were created for types.

Now `energy` can be defined and used like

```
energy :: Quantity Energy Double
energy = force *# length
```

```
energyToBoilMyGlassOfWater :: Quantity Energy Double
energyToBoilMyGlassOfWater = 12 # energy
```

Some functions

Let's create a function operating on quantities. It should calculate the area of a rectangle given the lengths of its sides. The type should hence look like

```
areaOfRectangle :: Quantity Length Double -> Quantity Length Double ->
                   Quantity (Length `Mul` Length) Double
```

And the actual function like

```
areaOfRectangle width height = width *# height
```

The values (`width` and `height`) can't be pattern matched on. This is so we can maintain the invariant we keep nagging about. The side effect is that only the comparative and artithmetic operations exported by `Dimensions.Quantity` can be used.

To wrap up, let's create a function to calculate the kinectic energy à la

$$E_k = \frac{m * v^2}{2}$$

```
kinecticEnergy :: Quantity Mass Double ->
                  Quantity Velocity Double ->
                  Quantity Energy Double
kinecticEnergy m v = m *# v *# v /# two
  where
    two = 2.0 # one
```

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Newtonian Mechanics / Single particle mechanics

[src: NewtonianMechanics/SingleParticle.lhs] Previous: Usage Table of contents Next: Teeter

```
module NewtonianMechanics.SingleParticle where
```

%< import Calculus.SyntaxTree

```
import           Test.QuickCheck
```

Laws:

- A body remains at rest or in uniform motion unless acted upon by a force.
- If two bodies exert forces on each other, these forces are equal in magnitude and opposite in direction.

We will begin our journey into classical mechanics by studying point particles. A point particle has a mass and it's position is given as vector in three dimensions. Of course it could exist in any number of dimensions but we'll stay in three dimensions since it is more intuitive and easier to understand.

The components of the vector are functions over time that gives the particles position in each dimension, x, y, and z. Since we've already defined vectors and mathmatical functions in previous chapters we won't spend any time on them here and instead just import those two modules.

```
import           Calculus.FunExpr
import           Calculus.DifferentialCalc -- Maybe remove
import           Calculus.IntegralCalc     -- Maybe remove
import           Vector.Vector as V
```

The mass of a particle is just a scalar value so we'll model it using $FunExpr$.

```
type Mass = FunExpr
```

We combine the constructor for vectors in three dimensions with the function expressions defined in the chapter on mathmatical analysis. We'll call this new type `VectorE` to signify that it's a vector of expressions.

```
type VectorE = Vector3 FunExpr
```

Now we are ready to define what the data type for a particle is. As we previously stated a point particle has a mass, and a position given as a vector of function expressions. So our data type is simply:

```
data Particle  = P { pos  :: VectorE -- Position as a function of time, unit m
                   , mass :: Mass    -- Mass, unit kg
                   } deriving Show
```

So now we can create our particles! Let's try it out!

```
particle :: Particle
particle = P (V3 (3 * Id * Id) (2 * Id) 1) 3
```

Let's see what happens when we run this in the interpreter.

```
ghci > particle
P {pos = (((3 * id) * id) x, (2 * id) y, 1 z), mass = 3}
```

We've created our first particle! And as we can see from the print out it's accelerating by $3t^2$ in the x-dimension, has a constant velocity of $2t$ in the y-dimension, is positioned at 1 in the z-dimension, and has a mass of 3.

Velocity & Acceleration

Velocity is defined as the derivative of the position with respect to time. More formally, ($\vec{r}$ denotes the position vector):

$$\vec{v} = \frac{d\vec{r}}{dt}$$

And since the position of our particles are given as vectors we'll do the derivation component-wise. We need not worry about the details of the derivation at all since this is all take care of by the Calculus module, all we need to to is use the constructor `D` for the derivative and apply it to each of the components of the vector. The business of applying something to each component of a vector has also already been taken care of! This was the point of `vmap` to map a function over the components of the vector. So if we combine them we get a rather elegant way of computing the velocity of a particle.

```
velocity :: Particle -> VectorE
velocity = vmap D . pos
```

We can try this out in the interpreter with our newly created particle.

```
ghci > velocity particle
((D ((3 * id) * id)) x, (D (2 * id)) y, (D 1) z)
```

Not very readable but at least we can see that it correctly maps the derivative over the components of the vector.

Acceleration is defined as the derivative of the velocity with respect to time, or the second derivative of the position. More formally:

$$\vec{a} = \frac{d\vec{v}}{dt} = \frac{d^2\vec{r}}{dt^2}$$

Exercise Try to figure out how to define the function for calculating the acceleration of a particle.

▼ **Solution**

We already know how to get the velocity of a particle, so the the only step we need to take is to take the derivative of the velocity.

```
acceleration :: Particle -> VectorE
acceleration = vmap D . velocity
```

Which is the same as:

```haskell
acceleration :: Particle -> VectorE
acceleration = vmap D . vmap D . pos
```

▼ **Trivia**

Those of you familiar with functor laws will probably see that the code for calculating the acceleration could also be written as:

```haskell
acceleration :: Particle -> VectorE
acceleration = vmap (D . D) . pos
```

Forces & Newton's second law

Newton's second law states that

> A body acted upon by a force moves in such a manner that the time rate of change of the momentum equals the force.

This law expresses the relationship between force and momentum and is mathematically defined as follows:

$$\vec{F} = \frac{d\vec{p}}{dt} = \frac{d(m \cdot \vec{v})}{dt}$$

Force is a vector so let's create a type synonym to make this clearer.

```haskell
type Force = VectorE
```

The quantity $m \cdot \vec{v}$ is what we mean when we say momentum. So the law states that the net force on a particle is equal to the rate of change of the momentum with respect to time. And since the definition of acceleration is $\vec{a} = \frac{d\vec{v}}{dt}$ we can write this law in a more familiar form, namely:

$$\vec{F} = m \cdot \vec{a}$$

And thus if the particle is accelerating we can calculate the force that must be acting on it, in code this would be:

```haskell
force :: Particle -> Force
force p = vmap (* m) a
  where
    m = mass p
    a = acceleration p
```

Where the acceleration of particle is found by deriving the velocity of that same particle with respect to t:

Kinetic energy

Kinetic stands for motion, so the kinetic energy of a particle is the energy it gains through movement. Energy is a scalar quantity so we'll model it using a $FunExpr$.

```haskell
type Energy = FunExpr
```

In classical mechanics the kinetic energy of particle is equal to $\frac{1}{2}$ of the product of its mass and the square of its velocity. In mathmatical notation this would be.

$$E_k = mv^2$$

But what does it mean to take the square of speed? We know of an operator to take the square of a number, but velocity is not a number it's a vector. So there seems to be a double meaning to what taking the square of something actually means depending on the type of the expression.

More concrete what we actually mean when we say `the square of vector'' is to take the dot product of the vector with itself. This may seem fairly straightforward and obvious on paper but when we start working with types this overloading of the word`square'' looks a bit strange. For numbers it has the type

```haskell
square :: Num a => a -> a
```

but for vectors it has the type

```haskell
square :: Num a => Vector a -> a
```

same name, vastly different meanings. If we were to encode this double meaning of the function we would have to create a type class which would encode this ability to be squared and then make Num and Vector instances of this class.

But enough yammering about types and double meanings! Let's actually define the function for taking the square of a vector and move on with our lives.

```
square :: VectorE -> FunExpr
square v = dotProd v v
```

Pretty straightforward. So with this out of the way we can now define the function for calculating the kinetic energy of a particle, again this is almost the same as just writing down the mathmatical formula with the added benefit of types.

```
kineticEnergy :: Particle -> Energy
kineticEnergy p = Const 0.5 * m * v2
  where
    m  = mass p
    v  = velocity p
    v2 = square v
```

Potential energy

In classical mechanics potential energy is defined as the energy possed by a particle because of its position relative to other particles, its electrical charge and other factors. This means that to actually calculate the potential energy of a particle we'd have to take into account all other particles that populate the system no matter how far apart they are.

Potential energy near Earth

Thankfully if we're close to the Earths gravitational field it's ok to simplify this problem and only take into account the Earths gravitational pull since all other factors are negliable in comparison.

The potential energy for particles affected by gravity is defined with mathmatical notation as:

$$E_p = m \cdot g \cdot h$$

where m is the mass of the particle, h its height, and g is the acceleration due to gravity (9.82 m/s^2).

```
potentialEnergy :: Particle -> Energy
potentialEnergy p = m * g * h
  where
    m          = mass p
    g          = Const 9.82
    (V3 _ _ h) = pos p
```

Work

If a constant force $\vec{F}$ is applied to a particle that moves from position $\vec{r_1}$ to $\vec{r_2}$ then the *work* done by the force is defined as the dot product of the force and the vector of displacement. In mathmatical notation this is written as

$$W = \vec{F} \cdot \Delta\vec{r}$$

where $\Delta\vec{r} = \vec{r_2} - \vec{r_1}$.

The work-energy theorem states that for a particle of constant mass *m*, the total work *W* done on the particle as it moves from position r_1 to r_2 is equal to the change in kinetic energy E_k of the particle:

$$W = \Delta E_K = E_{k,2} - E_{k,1} = \frac{1}{2}m(\vec{v_2}^2 - \vec{v_1}^2)$$

Let's codify this theorem:

PS: This used to work just fine, but it no longer does since the switch to FunExpr. Problem probably lies somewhere in SyntaxTree

```
prop_WorkEnergyTheorem :: Mass -> VectorE -> VectorE -> IO Bool
prop_WorkEnergyTheorem m v1 v2 = prettyEqual deltaEnergy (kineticEnergy
  displacedParticle)
  where
    particle1 = P v1 m -- | Two particles with the same mass
    particle2 = P v2 m -- | But different position vector
    -- |          E_k,2                      - E_k,1
```

```
    deltaEnergy = kineticEnergy particle2 - kineticEnergy particle1
    displacedParticle = P (v2 - v1) m


-- Test values
v1 = V3 (3 :* Id) (2 :* Id) (1 :* Id)
v2 = V3 0 0 (5 :* Id)
v3 = V3 0 (3 :* Id) 0 :: VectorE
v4 = V3 2 2 2 :: VectorE
m  = 5
p1 = P v1 m
p2 = P v2 m
dE = kineticEnergy p2 - kineticEnergy p1
p3 = P (v2 - v1) m
```

Law of universal gravitation

Newton's law of universal gravitation states that a particle attracts every other particle in the universe with a force that is directly proportional to the product of their masses, and is inversely proportional to the square of the distance between their centers.

This means that every particle with mass attracts every other particle with mass by a force pointing along the line intersecting both points.

There is an equation for calculating the magnitude of this force which states with math what we stated in words above:

$$F = G\frac{m_1 m_2}{r^2}$$

Where *F* is the maginitude of the force, *m1* and *m2* are the masses of the objects interacting, *r* is the distance between the centers of the masses and *G* is the gravitational constant.

The gravitational constant has been finely approximated through experiments and we can state it in our code like this:

```
type Constant = FunExpr

gravConst :: Constant
gravConst = 6.674 * (10 ** (-11))
```

Now we can codify the law of universal gravitation using our definition of particles.

```
lawOfUniversalGravitation :: Particle -> Particle -> FunExpr
lawOfUniversalGravitation p1 p2 = gravConst * ((m_1 * m_2) / r2)
  where
    m_1 = mass p1
    m_2 = mass p2
    r2 = square $ pos p2 - pos p1
```

If a particles position is defined as a vector representing its displacement from some origin O, then its heigh should be x. Or maybe it should be the magnitude of the vector, if the gravitational force originates from O. Hmmm

This seems weird since I don't know what the frame of reference is…

```
potentialEnergy :: Particle -> Energy
potentialEnergy p = undefined
  where
    m          = mass p
    (V3 x _ _) = pos p
```

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Examples / Teeter

[src: Examples/Teeter.lhs] Previous: Single particle mechanics Table of contents Next: Box on an incline

```
module Examples.Teeter where
```

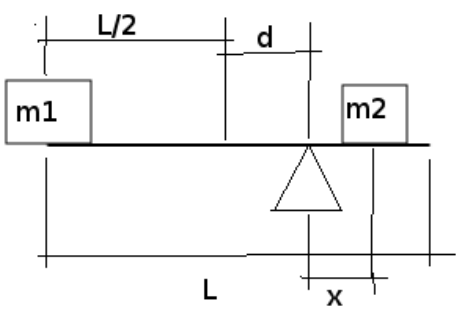
Exam excercise 3, 2017-01-13

```
import Dimensions.TypeLevel
import Dimensions.Quantity
import Prelude hiding (length)
```

Two boxes, m1 and m2, rests on a beam in balance.

Known values:

```
beam_M = 1.0 # mass
m1 = 2.0 # mass
m2 = 5.0 # mass
d = 0.75 # length
beam_L = 5.0 # length
two = 2.0 # one
```



Teeter

Direct implication:

```
beam_left_L = (beam_L /# two) +# d
beam_right_L = beam_L -# beam_left_L
```

We want to be able to represenet the torques.

A torque (sv. vridmoment) is defined as:

$$\tau = distance\ from\ turning\ point \cdot force$$

Since all force values will be composited of a mass and the gravitation, we can ignore the gravitation.

$$\tau = distance\ from\ turning\ point \cdot mass$$

```
m1_torq = m1 *# beam_left_L
```

To get the beams torque on one side, we need to divide by 2 because the beam's torque is spread out linearly (the density of the beam is equal everywhere), which means the left parts mass centrum is of the left parts total length.

$$beamL_\tau = beamL_M \cdot \frac{distance}{2}$$

$$beamL_\tau = \frac{beam\ left\ length}{beam\ length} \cdot beam_M \cdot \frac{beam\ left\ length}{2}$$

```
beamL_torq = ((beam_left_L /# beam_L) *# beam_M) *# (beam_left_L /# two)
```

```
beamR_torq = ((beam_right_L /# beam_L) *# beam_M) *# (beam_right_L /# two)
```

We make an expression for $m2_\tau$, which involves our unknown distance x.

$$m2_\tau = m2 \cdot x$$

For the teeter to be in balance, both sides torques should be equal.

$$Left\ side\ torque = Right\ side\ angular\ torque$$

We try to break out $m2_\tau$ and then x.

$$m1_\tau + beamL_\tau = m2_\tau + beamR_\tau$$

$$m1_\tau + beamL_\tau - beamR_\tau = m2_\tau$$

$$\frac{m1_\tau + beamL_\tau - beamR_\tau}{m2} = x$$

Our solution:

```
x = (m1_torq +# beamL_torq -# beamR_torq) /# m2
```

Security check:

```
m2_torq = m2 *# x


left_side_torque = m1_torq +# beamL_torq
right_side_torque = m2_torq +# beamR_torq
```

Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Examples / Box on an incline

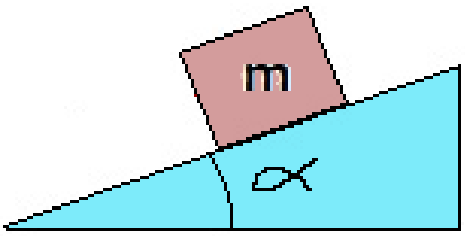
[src: Examples/Box_incline.lhs] Previous: Teeter Table of contents Next: Table of contents

Improvmenet: notation formulas tests

Box on an incline

```
import Vector.Vector
```

All vectors are in newton.



Incline

Notation: fg = gravitational accelleration

m = mass of box

```
fg = V2 0 (-10)
m = 2
```

```
unit_normal :: Double -> Vector2 Double
unit_normal a = V2 (cos a) (sin a)
```

Force against the incline from the box:

```
f_l_ :: Vector2 Double -> Angle -> Vector2 Double
f_l_ fa a = scale ((magnitude fa) * (cos a)) (unit_normal (a-(pi/2)))
```

The normal against the incline:

```
fn :: Vector2 Double -> Angle -> Vector2 Double
fn fa a = negate (f_l_ fa a)
```

Friction free incline:

Resulting force:

```
fr :: Vector2 Double -> Angle -> Vector2 Double
fr fa a = (fn fa a) + fa
```

With friction:

$$F_{friction} = \mu * F_{normal} \iff \mu = \frac{F_{friction}}{F_{normal}}$$

```
us = 0.5
uk = 0.4
```

Add image how friction depends if there is movement.

 Friction Friction

```
type FricConst = Double
```

Friction:

friks = Fn * us, us = friction static

frikk = Fn * uk, uk = friction kinetic

We have the normal force and only needs the constants.

The current speed does not affect the friction.

```
motscalar :: FricConst -> Vector2 Double -> Scalar
motscalar u f = u * (magnitude f)
```

Från en rörelse eller vekt, fixa komplementet

```
enh_vekt :: Vector2 Double -> Vector2 Double
enh_vekt v  | magnitude v == 0 = (V2 0 0)
            | otherwise = scale (1 / (magnitude v)) v


motkrafts :: FricConst -> Scalar -> Vector2 Double -> Vector2 Double
motkrafts u s v = scale (u * s) (negate (enh_vekt v))

motkraftv :: FricConst -> Vector2 Double -> Vector2 Double -> Vector2 Double
motkraftv u n v = scale (u * (magnitude n)) (negate (enh_vekt v))
```

Now we just need to sum the force vectors:

```
fru :: Vector2 Double -> Angle -> FricConst -> Vector2 Double
fru fa a u = (fr fa a) + (motkraftv u (fn fa a) (fr fa a))

fru' :: Vector2 Double -> Angle -> FricConst -> Vector2 Double
fru' fa a u = (motkraftv u (fn fa a) (fr fa a))
```