Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

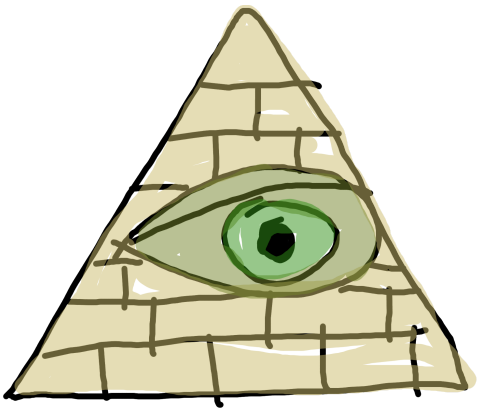
Calculus / Differential calculus

[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*

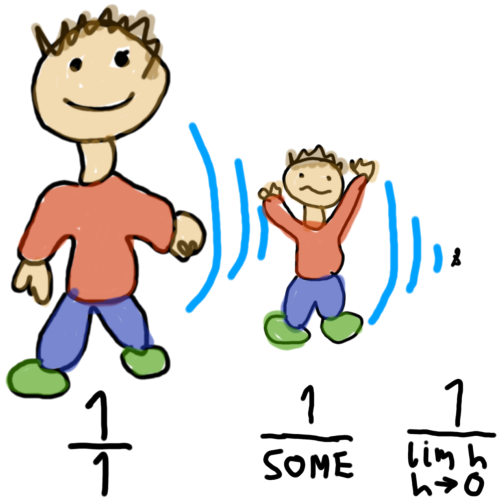
> In calculus, Leibniz's notation, named in honor of the 17th-century German philosopher and mathematician Gottfried Wilhelm Leibniz, uses the symbols dx and dy to represent infinitely small (or infinitesimal) increments of x and y, respectively, just as Δx and Δy represent finite increments of x and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