Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Introduction / So what's a DSL?

```haskell
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