1 Types, DSLs, and complex numbers

This chapter is partly based on the paper [Ionescu and Jansson, 2016] from the International Workshop on Trends in Functional Programming in Education 2015. We will implement certain concepts in the functional programming language Haskell and the code for this lecture is placed in a module called *DSLsofMath.W01* that starts here:

```
module DSLsofMath.W01 where
import DSLsofMath.CSem (ComplexSem (CS), (.+.), (.*.))
import Numeric.Natural (Natural)
import Data.Ratio (Rational, Ratio, (%))
import Data.List (find)
```

These lines constitute the module header which usually start a Haskell file. We will not go into details of the module header syntax here but the purpose is to "name" the module itself (here *DSLsofMath.W01*) and to **import** (bring into scope) definitions from other modules. As an example, the second to last line imports types for rational numbers and the infix operator (%) used to construct ratios (1 % 7 is Haskell notation for $\frac{1}{7}$, etc.).

1.1 Intro: Pitfalls with traditional mathematical notation

A function or the value at a point? Mathematical texts often talk about "the function $f(x)$" when "the function f" would be more clear. Otherwise there is a risk of confusion between $f(x)$ as a function and $f(x)$ as the value you get from applying the function f to the value bound to the name x.

Examples: let $f(x) = x + 1$ and let $t = 5 * f(2)$. Then it is clear that the value of t is the constant 15. But if we let $s = 5 * f(x)$ it is not clear if s should be seen as a constant or as a function of x.

Paying attention to types and variable scope often helps to sort out these ambiguities.

Scoping The syntax and scoping rules for the integral sign are rarely explicitly mentioned, but looking at it from a software perspective can help. If we start from a simple example, like $\int_1^2 x^2 dx$, it is relatively clear: the integral sign takes two real numbers as limits and then a certain notation for a function, or expression, to be integrated. Comparing the part after the integral sign to the syntax of a function definition $f(x) = x^2$ reveals a rather odd rule: instead of *starting* with declaring the variable x, the integral syntax *ends* with the variable name, and also uses the letter "d". (There are historical explanations for this notation, and it is motivated by computation rules in the differential calculus, but we will not go there now.) It seems like the scope of the variable "bound" by d is from the integral sign to the final dx, but does it also extend to the limits? The answer is no, as we can see from a slightly extended example:

$$f(x) = x^2$$
$$g(x) = \int_x^{2x} f(x)dx \qquad = \int_x^{2x} f(y)dy$$

The variable x bound on the left is independent of the variable x "bound under the integral sign". Mathematics text books usually avoid the risk of confusion by (silently) renaming variables when needed, but we believe this renaming is a sufficiently important operation to be more explicitly mentioned.

Figure 2: Humorously inappropriate use of numbers on a sign in New Cuyama, California. By I, MikeGogulski, CC BY 2.5, Wikipedia.

1.2 Types of data

Dividing up the world (or problem domain) into values of different types is one of the guiding principles of this course. We will see that keeping track of types can guide the development of theories, languages, programs and proofs.

1.2.1 What is a type?

As mentioned in the introduction, we emphasise the dividing line between syntax (what mathematical expressions look like) and semantics (what they mean). As an example we start with *type expressions* — first in mathematics and then in Haskell. To a first approximation you can think of types as sets. The type of truth values, *True* and *False*, is often called *Bool* or just $\mathbb{B}$. Thus the name (syntax) is $\mathbb{B}$ and the semantics (meaning) is the two-element set $\{\mathit{False}, \mathit{True}\}$. Similarly, we have the type $\mathbb{N}$ whose semantics is the infinite set of natural numbers $\{0, 1, 2, ...\}$. Other common types are $\mathbb{Z}$ of integers, $\mathbb{Q}$ of rationals, and $\mathbb{R}$ of real numbers.

So far the syntax is trivial — just names for certain sets — but we can also combine these, and the most important construction is the function type. For any two type expressions A and B we can form the function type $A \to B$. The semantics is the set of "functions from A to B"[2] As an example, the semantics of $\mathbb{B} \to \mathbb{B}$ is a set of four functions: $\{\mathit{const}\ \mathit{False}, \mathit{id}, \neg, \mathit{const}\ \mathit{True}\}$ where $\neg : \mathbb{B} \to \mathbb{B}$ is boolean negation. The function type construction is very powerful, and can be used to model a wide range of concepts in mathematics (and the real world).

Function building blocks. As function types are really important, we will now introduce a few basic building blocks which are as useful for functions as zero and one are for numbers. For each type A there is an *identity function* $\mathit{id}_A : A \to A$. In Haskell all of these functions are defined once and for all as follows:

$$\mathit{id} :: a \to a$$
$$\mathit{id}\ x = x$$

When a type variable (here a) is used in a type signature it is implicitly quantified (bound) as if preceded by "for all types a". This use of type variables is called "parametric polymorphism" and

[2] Formally the semantics is the set of functions from the semantics of A to the semantics of B.

the compiler gives more help when implementing functions with such types. Another "function building block" is *const* which has two type variables and two arguments:

$$const :: a \to b \to a$$
$$const\ x\ \_ = x$$

Two-argument functions like *const* are sometimes used as binary operators.

As a first example of a *higher-order* function we present *flip* which "flips" the two arguments of a binary operator.

$$flip :: (a \to b \to c) \to (b \to a \to c)$$
$$flip\ op\ x\ y = op\ y\ x$$

As an example $flip\ (-)\ 5\ 10 == 10 - 5$ and $flip\ const\ x\ y == const\ y\ x == y$.

Function composition. The infix operator . (period) in Haskell is an implementation of the mathematical operation of function composition. The period is an ASCII approximation of the composition symbol ∘ typically used in mathematics. (The symbol ∘ is encoded as `U+2218` and called RING OPERATOR in Unicode, `&#8728` in HTML, `\circ` in TEX, etc.) Its implementation is:

$$f \circ g = \lambda x \to f\ (g\ x)$$

As an exercise it is good to experiment a bit with these building blocks to see how they fit together and what types their combinations have.

The type is perhaps best illustrated by a diagram with types as nodes and functions (arrows) as directed edges:

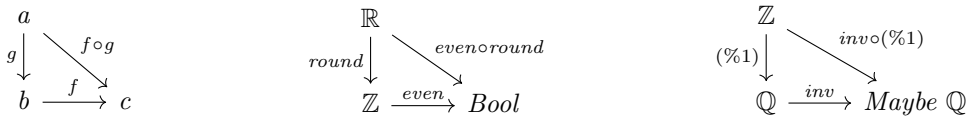


Figure 3: Function composition diagrams: in general, and two examples

In Haskell we get the following type:

$$(\circ) :: (b \to c) \to (a \to b) \to (a \to c)$$

which may take a while to get used to.

Variable names as type hints In mathematical texts there are often conventions about the names used for variables of certain types. Typical examples include f, g for functions, i, j, k for natural numbers or integers, x, y for real numbers and z, w for complex numbers.

The absence of explicit types in mathematical texts can sometimes lead to confusing formulations. For example, a standard text on differential equations by Edwards, Penney, and Calvis [2008] contains at page 266 the following remark:

> The differentiation operator D can be viewed as a transformation which, when applied to the function $f(t)$, yields the new function $D\{f(t)\} = f'(t)$. The Laplace transformation $\mathcal{L}$ involves the operation of integration and yields the new function $\mathcal{L}\{f(t)\} = F(s)$ of a new independent variable s.

This is meant to introduce a distinction between "operators", such as differentiation, which take functions to functions of the same type, and "transforms", such as the Laplace transform, which take functions to functions of a new type. To the logician or the computer scientist, the way of phrasing this difference in the quoted text sounds strange: surely the *name* of the independent variable does not matter: the Laplace transformation could very well return a function of the "old" variable t. We can understand that the name of the variable is used to carry semantic meaning about its type (this is also common in functional programming, for example with the conventional use of a plural "s" suffix, as in the name xs, to denote a list of values.). Moreover, by using this (implicit!) convention, it is easier to deal with cases such as that of the Hartley transform (a close relative of the Fourier transform), which does not change the type of the input function, but rather the *interpretation* of that type. We prefer to always give explicit typings rather than relying on syntactical conventions, and to use type synonyms for the case in which we have different interpretations of the same type. In the example of the Laplace transformation, this leads to

$$\mathcal{L} : (T \to \mathbb{C}) \to (S \to \mathbb{C})$$

where $T = \mathbb{R}$ and $S = \mathbb{C}$ Note that the function type constructor ($\to$) is used three times here: once in $T \to \mathbb{C}$, once in $S \to \mathbb{C}$ and finally at the top level to indicate that the transform maps functions to functions. This means that $\mathcal{L}$ is an example of a higher-order function, and we will see many uses of this idea in this book.

Now we move to introducing some of the ways types are defined in Haskell, the language we use for implementation (and often also specification) of mathematical concepts.

1.2.2 Types in Haskell: type, newtype, and data

There are three keywords in Haskell involved in naming types: **type**, **newtype**, and **data**.

type – abbreviating type expressions The **type** keyword is used to create a type synonym - just another name for a type expression. The semantics is unchanged: the set of values of type *Heltal* is exactly the same as the set of values of type *Integer*, etc.

```
type Heltal   = Integer
type Foo      = (Maybe [String], [[Heltal]])
type BinOp    = Heltal → Heltal → Heltal
type Env v s = [(v, s)]
```

The new name for the type on the right hand side (RHS) does not add type safety, just readability (if used wisely). The *Env* example shows that a type synonym can have type parameters. Note that *Env v s* is a type (for any types v and s), but *Env* itself is not a type but a *type constructor*.

newtype – more protection A simple example of the use of **newtype** in Haskell is to distinguish values which should be kept apart. A fun example of *not* keeping values apart is shown in Figure 2. To avoid this class of problems Haskell provides the **newtype** construct as a stronger version of **type**.

```
newtype Population      = Pop Int   -- Population count
newtype Ftabovesealevel = Hei Int   -- Elevation in feet above sea level
newtype Established     = Est Int   -- Year of establishment

  -- Example values of the new types
pop :: Population;          pop = Pop 562;
hei :: Ftabovesealevel;     hei = Hei 2150;
est :: Established;         est = Est 1951;
```

This example introduces three new types, *Population*, *Ftabovesealevel*, and *Established*, which all are internally represented by an *Int* but which are good to keep apart. The syntax also introduces *constructor functions* $Pop :: Int \to Population$, *Hei* and *Est* which can be used to translate from plain integers to the new types, and for pattern matching. The semantics of *Population* is the set of values of the form *Pop i* for every value $i :: Int$. It is not the same as the semantics of *Int* but it is isomorphic (there is a one-to-one correspondence between the sets).

Later in this chapter we use a newtype for the semantics of complex numbers as a pair of numbers in the cartesian representation but it may also be useful to have another newtype for complex as a pair of numbers in the polar representation.

The keyword data – for syntax trees The simplest form of a recursive datatype is the unary notation for natural numbers:

data $N = Z \mid S\ N$

This declaration introduces

- a new type N for unary natural numbers,
- a constructor $Z :: N$ to represent zero, and
- a constructor $S :: N \to N$ to represent the successor.

The semantics of N is the set infinite $\{Z, S\ Z, S\ (S\ Z), ...\}$ which is isomorphic to $\mathbb{N}$. Examples values: $zero = Z$, $one = S\ Z$, $three = S\ (S\ one)$.

The **data** keyword will be used throughout the course to define datatypes of syntax trees for different kinds of expressions: simple arithmetic expresssions, complex number expresssions, etc. But it can also be used for non-recursive datatypes, like **data** *Bool* = *False* | *True*, or **data** *TownData* = *Town String Population Established*. The *Bool* type is the simplest example of a *sum type*, where each value uses either of the two variants *False* and *True* as the constructor. The *TownData* type is an example of a *product type*, where each value uses the same constructor *Town* and records values for the name, population, and year of establishment of the town modelled. (See exercise 1.11 for the intuition behind the terms "sum" and "product" used here.)

Maybe **and parameterised types.** It is very often possible describe a family of types using a type parameter. One simple example is the type constructor *Maybe*:

data *Maybe a* = *Nothing* | *Just a*

This declaration introduces

- a new type *Maybe a* for every type a,
- a constructor *Nothing* :: *Maybe a* to represent "no value", and
- a constructor $Just :: a \to Maybe\ a$ to represent "just a value".

A maybe type is often used when an operation may, or may not, return a value:

$inv :: \mathbb{Q} \to Maybe\ \mathbb{Q}$
$inv\ 0 = Nothing$
$inv\ r = Just\ (1\ /\ r)$

Two other examples of, often used, parameterised types are (a, b) for the type of pairs (a product type) and *Either a b* for either an a or a b (a sum type).

data *Either p q* = *Left p* | *Right q*

1.2.3 *Env* and variable *lookup*.

The type synonym

$$\textbf{type}\ Env\ v\ s = [(v, s)]$$

is one way of expressing a partial function from v to s. As an example value of this type we can take:

```
env1 :: Env String Int
env1 = [("hej", 17), ("du", 38)]
```

We can see the type *Env* v s as a syntactic representation of a partial function from v to s. We can convert to a total function *Maybe* returning an s using *evalEnv*:

```
evalEnv :: Eq v => Env v s -> (v -> Maybe s)
```

This type signature deserves some more explanation. The first part (*Eq* v $\Rightarrow$) is a constraint which says that the function works, not for *all* types v, but only for those who support a boolean equality check ($(==) :: v \to v \to Bool$). The rest of the type signature ($Env\ v\ s \to (v \to Maybe\ s)$) can be interpreted in two ways: either as the type of a one-argument function taking an *Env* v s and returning a function, or as the type of a two-argument function taking an *Env* v s and a v and maybe returning an s.

```
evalEnv vss var = findFst vss
  where findFst ((v, s) : vss)
              | var == v    = Just s
              | otherwise = findFst vss
        findFst []          = Nothing
```

Or we can use the Haskell prelude function *lookup* = *flip evalEnv*:

```
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
```

We will use *Env* and *lookup* below (in section 1.3) when we introduce abstract syntax trees containing variables.

1.3 A syntax for simple arithmetical expressions

$$\textbf{data}\ AE = V\ String \mid P\ AE\ AE \mid T\ AE\ AE$$

This declaration introduces

- a new type AE for simple arithmetic expressions,
- a constructor $V :: String \to AE$ to represent variables,
- a constructor $P :: AE \to AE \to AE$ to represent plus, and
- a constructor $T :: AE \to AE \to AE$ to represent times.

Example values: $x = V$ `"x"`, $e_1 = P\ x\ x$, $e_2 = T\ e_1\ e_1$

If you want a contructor to be used as an infix operator you need to use symbol characters and start with a colon:

$$\textbf{data}\ AE' = V'\ String \mid AE' :+ AE' \mid AE' :* AE'$$

Example values: $y = V'$ "y", $e_1 = y :+ y$, $e_2 = x :* e_1$

Finally, you can add one or more type parameters to make a whole family of datatypes in one go:

$$\textbf{data}\ AE'\ v = V'\ v \mid AE'\ v :+ AE'\ v \mid AE'\ v :* AE'\ v$$

The purpose of the parameter v here is to enable a free choice of type for the variables (be it *String* or *Int* or something else).

The careful reader will note that the same Haskell module cannot contain both these definitions of AE'. This is because the name of the type and the names of the constructors are clashing. The typical ways around this are: define the types in different modules, or rename one of them (often by adding primes as in AE'). In this book we often take the liberty of presenting more than one version of a datatype without changing the names, to avoid multiple modules or too many primed names.

Together with a datatype for the syntax of arithmetic expressions we also want to define an evaluator of the expressions. The concept of "an evaluator", a function from the syntax to the semantics, is something we will return to many times in this book. We have already seen one example: the function *evalEnv* which translates from a list of key-value-pairs (the abstract syntax of the environment) to a function (the semantics).

In the evaluator for $AE'\ v$ we take this one step further: given an environment *env* and the syntax of an arithmetic expression e we compute the semantics of that expression.

$$\begin{aligned}
&evalAE :: Env\ String\ Integer \to (AE \to Maybe\ Integer)\\
&evalAE\ env\ (V\ x) \quad\ = evalEnv\ env\ x\\
&evalAE\ env\ (P\ e_1\ e_2) = mayP\ (evalAE\ env\ e_1)\ (evalAE\ env\ e_2)\\
&evalAE\ env\ (T\ e_1\ e_2) = mayT\ (evalAE\ env\ e_1)\ (evalAE\ env\ e_2)\\
&mayP :: Maybe\ Integer \to Maybe\ Integer \to Maybe\ Integer\\
&mayP\ (Just\ a)\ (Just\ b) = Just\ (a + b)\\
&mayP\ _\ _ \qquad\qquad\ = Nothing\\
&mayT :: Maybe\ Integer \to Maybe\ Integer \to Maybe\ Integer\\
&mayT\ (Just\ a)\ (Just\ b) = Just\ (a * b)\\
&mayT\ _\ _ \qquad\qquad\ = Nothing
\end{aligned}$$

The corresponding code for AE' is more general and you don't need to understand it at this stage, but it is left here as an example for those with a stronger Haskell background.

$$\begin{aligned}
&evalAE' :: (Eq\ v, Num\ sem) \Rightarrow (Env\ v\ sem) \to (AE'\ v \to Maybe\ sem)\\
&evalAE'\ env\ (V'\ x) \quad\ = evalEnv\ env\ x\\
&evalAE'\ env\ (e_1 :+ e_2) = liftM\ (+)\ (evalAE'\ env\ e_1)\ (evalAE'\ env\ e_2)\\
&evalAE'\ env\ (e_1 :* e_2) = liftM\ (*)\ \ (evalAE'\ env\ e_1)\ (evalAE'\ env\ e_2)\\
&liftM :: (a \to b \to c) \to (Maybe\ a \to Maybe\ b \to Maybe\ c)\\
&liftM\ op \quad (Just\ a)\ (Just\ b) \quad\ = Just\ (op\ a\ b)\\
&liftM\ _op\ _ \qquad\ _ \qquad\qquad\ = Nothing
\end{aligned}$$

1.4 A case study: complex numbers

We now turn to our first case study: an analytic reading of the introduction of complex numbers in Adams and Essex [2010]. We choose a simple domain to allow the reader to concentrate on the essential elements of our approach without the distraction of potentially unfamiliar mathematical

concepts. For this section, we bracket our previous knowledge and approach the text as we would a completely new domain, even if that leads to a somewhat exaggerated attention to detail.

Adams and Essex introduce complex numbers in Appendix A. The section *Definition of Complex Numbers* begins with:

> We begin by defining the symbol i, called **the imaginary unit**, to have the property
>
> $$i^2 = -1$$
>
> Thus, we could also call i the square root of -1 and denote it $\sqrt{-1}$. Of course, i is not a real number; no real number has a negative square.

At this stage, it is not clear what the type of i is meant to be, we only know that i is not a real number. Moreover, we do not know what operations are possible on i, only that i^2 is another name for -1 (but it is not obvious that, say $i * i$ is related in any way with i^2, since the operations of multiplication and squaring have only been introduced so far for numerical types such as $\mathbb{N}$ or $\mathbb{R}$, and not for "symbols").

For the moment, we introduce a type for the symbol i, and, since we know nothing about other symbols, we make i the only member of this type:

```
data ImagUnits = I
i :: ImagUnits
i = I
```

We use a capital I in the **data** declaration because a lowercase constructor name would cause a syntax error in Haskell. For convenience we add a synonym i == I. We can give the translation from the abstract syntax to the concrete syntax as a function *showIU*:

```
showIU :: ImagUnits → String
showIU   I          = "i"
```

Next, we have the following definition:

> **Definition:** A **complex number** is an expression of the form
>
> $$a + bi \qquad \text{or} \qquad a + ib,$$
>
> where a and b are real numbers, and i is the imaginary unit.

This definition clearly points to the introduction of a syntax (notice the keyword "form"). This is underlined by the presentation of *two* forms, which can suggest that the operation of juxtaposing i (multiplication?) is not commutative[3].

A profitable way of dealing with such concrete syntax in functional programming is to introduce an abstract representation of it in the form of a datatype:

```
data ComplexA = CPlus1 ℝ ℝ ImagUnits   -- the form a + bi
              | CPlus2 ℝ ImagUnits ℝ   -- the form a + ib
```

We can give the translation from the abstract syntax to the concrete syntax as a function *showCA*:

[3] See section 1.6 for more about commutativity.

$$
\begin{aligned}
&\mathit{showCA} :: \mathit{ComplexA} \to \quad \mathit{String}\\
&\mathit{showCA}\quad (\mathit{CPlus}_1\ x\ y\ i) = \mathit{show}\ x +\!\!+ \text{" + "} +\!\!+ \mathit{show}\ y +\!\!+ \mathit{showIU}\ i\\
&\mathit{showCA}\quad (\mathit{CPlus}_2\ x\ i\ y) = \mathit{show}\ x +\!\!+ \text{" + "} +\!\!+ \mathit{showIU}\ i +\!\!+ \mathit{show}\ y
\end{aligned}
$$

Notice that the type $\mathbb{R}$ is not implemented yet and it is not really even exactly implementable but we want to focus on complex numbers so we will approximate $\mathbb{R}$ by double precision floating point numbers for now.

$$\textbf{type}\ \mathbb{R} = \mathit{Double}$$

The text continues with examples:

> For example, $3 + 2i$, $\frac{7}{2} - \frac{2}{3}i$, $i\pi = 0 + i\pi$ and $-3 = -3 + 0i$ are all complex numbers. The last of these examples shows that every real number can be regarded as a complex number.

The second example is somewhat problematic: it does not seem to be of the form $a+bi$. Given that the last two examples seem to introduce shorthand for various complex numbers, let us assume that this one does as well, and that $a - bi$ can be understood as an abbreviation of $a + (-b)\ i$. With this provision, in our notation the examples are written as in Table 2.

Mathematics		Haskell
$3 + 2i$		$\mathit{CPlus}_1\ 3\ 2\ I$
$\frac{7}{2} - \frac{2}{3}i$	$= \frac{7}{2} + \frac{-2}{3}i$	$\mathit{CPlus}_1\ (7\ /\ 2)\ (-2\ /\ 3)\ I$
$i\pi$	$= 0 + i\pi$	$\mathit{CPlus}_2\ 0\ I\ \pi$
-3	$= -3 + 0i$	$\mathit{CPlus}_1\ (-3)\ 0\ I$

Table 2: Examples of notation and abstract syntax for some complex numbers.

We interpret the sentence "The last of these examples ..." to mean that there is an embedding of the real numbers in *ComplexA*, which we introduce explicitly:

$$
\begin{aligned}
&\mathit{toComplex} :: \mathbb{R} \to \mathit{ComplexA}\\
&\mathit{toComplex}\ x = \mathit{CPlus}_1\ x\ 0\ I
\end{aligned}
$$

Again, at this stage there are many open questions. For example, we can assume that the mathematical expression $i1$ stands for the complex number $\mathit{CPlus}_2\ 0\ I\ 1$, but what about the expression i by itself? If juxtaposition is meant to denote some sort of multiplication, then perhaps 1 can be considered as a unit, in which case we would have that i abbreviates $i1$ and therefore $\mathit{CPlus}_2\ 0\ I\ 1$. But what about, say, $2i$? Abbreviations with i have only been introduced for the ib form, and not for the bi one!

The text then continues with a parenthetical remark which helps us dispel these doubts:

> (We will normally use $a + bi$ unless b is a complicated expression, in which case we will write $a + ib$ instead. Either form is acceptable.)

This remark suggests strongly that the two syntactic forms are meant to denote the same elements, since otherwise it would be strange to say "either form is acceptable". After all, they are acceptable by definition.

Given that $a + ib$ is only "syntactic sugar" for $a + bi$, we can simplify our representation for the abstract syntax, eliminating one of the constructors:

data *ComplexB* = *CPlusB* $\mathbb{R}$ $\mathbb{R}$ *ImagUnits*

In fact, since it doesn't look as though the type *ImagUnits* will receive more elements, we can dispense with it altogether:

data *ComplexC* = *CPlusC* $\mathbb{R}$ $\mathbb{R}$

(The renaming of the constructor to *CPlusC* serves as a guard against the case that we have suppressed potentially semantically relevant syntax.)

We read further:

> It is often convenient to represent a complex number by a single letter; w and z are frequently used for this purpose. If a, b, x, and y are real numbers, and $w = a + bi$ and $z = x + yi$, then we can refer to the complex numbers w and z. Note that $w = z$ if and only if $a = x$ and $b = y$.

First, let us notice that we are given an important semantic information: to check equality for complex numbers, it is enough to check equality of the components (the arguments to the constructor *CPlusC*). Another way of saying this is that *CPlusC* is injective. The equality on complex numbers is what we would obtain in Haskell by using **deriving** *Eq*.

This shows that the set of complex numbers is, in fact, isomorphic with the set of pairs of real numbers, a point which we can make explicit by re-formulating the definition in terms of a **newtype**:

newtype *ComplexD* = *CD* $(\mathbb{R}, \mathbb{R})$ **deriving** *Eq*

The point of the somewhat confusing discussion of using "letters" to stand for complex numbers is to introduce a substitute for *pattern matching*, as in the following definition:

> **Definition:** If $z = x + yi$ is a complex number (where x and y are real), we call x the **real part** of z and denote it $Re\ (z)$. We call y the **imaginary part** of z and denote it $Im\ (z)$:
>
> $$Re\ (z) = Re\ (x + yi) = x$$
> $$Im\ (z) = Im\ (x + yi) = y$$

This is rather similar to Haskell's *as-patterns*:

$$\begin{aligned} &re :: ComplexD \quad \to \mathbb{R} \\ &re\ z@(CD\ (x, y)) \ = x \\ &im :: ComplexD \quad \to \mathbb{R} \\ &im\ z@(CD\ (x, y)) = y \end{aligned}$$

a potential source of confusion being that the symbol z introduced by the as-pattern is not actually used on the right-hand side of the equations (although it could be).

The use of as-patterns such as "$z = x + yi$" is repeated throughout the text, for example in the definition of the algebraic operations on complex numbers:

> **The sum and difference of complex numbers**
>
> If $w = a + bi$ and $z = x + yi$, where a, b, x, and y are real numbers, then
>
> $$w + z = (a + x) + (b + y)\ i$$
> $$w - z = (a - x) + (b - y)\ i$$

With the introduction of algebraic operations, the language of complex numbers becomes much richer. We can describe these operations in a *shallow embedding* in terms of the concrete datatype *ComplexD*, for example:

$$\begin{aligned}&\mathit{plusD} :: \mathit{ComplexD} \to \mathit{ComplexD} \to \mathit{ComplexD}\\&\mathit{plusD}\ (\mathit{CD}\ (a,b))\ (\mathit{CD}\ (x,y)) = \mathit{CD}\ ((a+x),(b+y))\end{aligned}$$

or we can build a datatype of "syntactic" complex numbers from the algebraic operations to arrive at a *deep embedding* as seen in the next section. Both shallow and deep embeddings will be further explained in section 3.9.

Exercises:

- implement $(*.)$ for *ComplexD*

At this point we can sum up the "evolution" of the datatypes introduced so far. Starting from *ComplexA*, the type has evolved by successive refinements through *ComplexB*, *ComplexC*, ending up in *ComplexD* (see Fig. 4). We can also make a parameterised version of *ComplexD*, by noting that the definitions for complex number operations work fine for a range of underlying numeric types. The operations for *ComplexSem* are defined in module *CSem*, available in appendix F.

$$\begin{array}{lll}\textbf{data} & \mathit{ImagUnits} & = I\\ \textbf{data} & \mathit{ComplexA} & = \mathit{CPlus}_1\ \mathbb{R}\ \mathbb{R}\ \mathit{ImagUnits}\\ & & \mid \mathit{CPlus}_2\ \mathbb{R}\ \mathit{ImagUnits}\ \mathbb{R}\\ \textbf{data} & \mathit{ComplexB} & = \mathit{CPlusB}\ \mathbb{R}\ \mathbb{R}\ \mathit{ImagUnits}\\ \textbf{data} & \mathit{ComplexC} & = \mathit{CPlusC}\ \mathbb{R}\ \mathbb{R}\\ \textbf{newtype} & \mathit{ComplexD} & = \mathit{CD}\ (\mathbb{R},\mathbb{R})\ \textbf{deriving}\ \mathit{Eq}\\ \textbf{newtype} & \mathit{ComplexSem}\ r & = \mathit{CS}\ (r,r)\ \ \textbf{deriving}\ \mathit{Eq}\end{array}$$

Figure 4: Complex number datatype refinement (semantics).

1.5 A syntax for (complex) arithmetical expressions

So far we have tried to find a datatype to represent the intended *semantics* of complex numbers. That approach is called "shallow embedding". Now we turn to the *syntax* instead ("deep embedding").

We want a datatype *ComplexE* for the abstract syntax tree of expressions. The syntactic expressions can later be evaluated to semantic values:

$$\mathit{evalE} :: \mathit{ComplexE} \to \mathit{ComplexD}$$

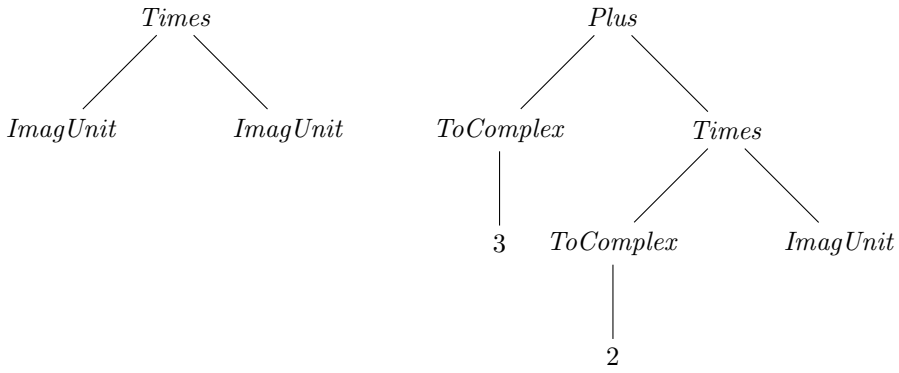
The datatype *ComplexE* should collect ways of building syntactic expression representing complex numbers and we have so far seen the symbol i, an embedding from $\mathbb{R}$, plus and times. We make these four *constructors* in one recursive datatype as follows:

$$\begin{array}{l}\textbf{data}\ \mathit{ComplexE} = \mathit{ImagUnit}\quad \text{-- syntax for } i\text{, not to be confused with the type } \mathit{ImagUnits}\\ \qquad\qquad \mid \mathit{ToComplex}\ \mathbb{R}\\ \qquad\qquad \mid \mathit{Plus}\quad \mathit{ComplexE}\ \mathit{ComplexE}\\ \qquad\qquad \mid \mathit{Times}\ \mathit{ComplexE}\ \mathit{ComplexE}\\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})\end{array}$$

Note that, in *ComplexA* above, we also had a constructor for "plus", but it was another "plus". They are distinguished by type: $CPlus_1$ took (basically) two real numbers, while *Plus* here takes two (expressions representing) complex numbers as arguments.

Here are two examples of type *ComplexE* as Haskell code and as abstract syntax trees:

$$testE1 = Times\ ImagUnit\ ImagUnit$$
$$testE2 = Plus\ (ToComplex\ 3)\ (Times\ (ToComplex\ 2)\ ImagUnit)$$



We can implement the evaluator *evalE* by pattern matching on the syntax tree and recursion. To write a recursive function requires a small leap of faith. It can be difficult to get started implementing a function (like *eval*) that should handle all the cases and all the levels of a recursive datatype (like *ComplexE*). One way to overcome this difficulty is through "wishful thinking": assume that all but one case have been implemented already. All you need to focus on is that one remaining case, and you can freely call the function (that you are implementing) recursively, as long as you do it for subexpressions (subtrees of the abstrac syntax tree datatype).

For example, when implementing the $evalE\ (Plus\ c_1\ c_2)$ case, you can assume that you already know the values $s_1, s_2 :: ComplexD$ corresponding to the subtrees c_1 and c_2 of type *ComplexE*. The only thing left is to add them up componentwise and we can assume there is a function $plusD :: ComplexD \to ComplexD \to ComplexD$ taking care of this step (in fact, we implemented it earlier in section 1.4). Continuing in this direction (by "wishful thinking") we arrive at the following implementation.

$$\begin{aligned}
evalE\ ImagUnit &= imagUnitD \\
evalE\ (ToComplex\ r) &= toComplexD\ r \\
evalE\ (Plus\ c_1\ c_2) &= plusD\ \ (evalE\ c_1)\ (evalE\ c_2) \\
evalE\ (Times\ c_1\ c_2) &= timesD\ (evalE\ c_1)\ (evalE\ c_2)
\end{aligned}$$

Note the pattern here: for each constructor of the syntax datatype we assume there exists a corresponding semantic function. The next step is to implement these functions, but let us first list their types and compare with the types of the syntactic constructors:

$$\begin{aligned}
&imagUnitD :: ComplexD && \text{-- } ComplexE \\
&toComplexD :: \mathbb{R} \to ComplexD && \text{-- } \mathbb{R} \to ComplexE \\
&timesD :: ComplexD \to ComplexD \to ComplexD && \text{-- } ComplexE \to ComplexE \to ComplexE
\end{aligned}$$

As we can see, each use of *ComplexE* has been replaced be a use of *ComplexD*. Finally, we can start filling in the implementations:

$$\begin{aligned}
imagUnitD &= CD\ (0,1) \\
toComplexD\ r &= CD\ (r,0)
\end{aligned}$$

The function *plusD* was defined earlier and *timesD* is left as an exercise for the reader. To sum up we have now implemented a recursive datatype for mathematical expressions describing complex

numbers, and an evaluator that computes the underlying number. Note that many different syntactic expressions will evaluate to the same number (*evalE* is not injective).

Generalising from the example of *testE2* we also define a function to embed a semantic complex number in the syntax:

$$\begin{aligned}&\mathit{fromCD} :: \mathit{ComplexD} \to \mathit{ComplexE}\\&\mathit{fromCD}\ (\mathit{CD}\ (x, y)) = \mathit{Plus}\ (\mathit{ToComplex}\ x)\ (\mathit{Times}\ (\mathit{ToComplex}\ y)\ \mathit{ImagUnit})\end{aligned}$$

This function is injective.

1.6 Laws, properties and testing

There are certain laws we would like to hold for operations on complex numbers. To specify these laws, in a way which can be easily testable in Haskell, we use functions to *Bool* (also called *predicates* or *properties*). The intended meaning of such a boolean function (representing a law) is "forall inputs, this should return *True*". This idea is at the core of *property based testing* (pioneered by Claessen and Hughes [2000]) and conveniently available in the library QuickCheck.

The simplest law is perhaps $i^2 = -1$ from the start of section 1.4,

$$\begin{aligned}&\mathit{propImagUnit} :: \mathit{Bool}\\&\mathit{propImagUnit} = \mathit{Times}\ \mathit{ImagUnit}\ \mathit{ImagUnit} \mathrel{===} \mathit{ToComplex}\ (-1)\end{aligned}$$

Note the we use a new operator here, $(===)$, because the left hand side (LHS) is clearly not syntactically equal to the right hand side (RHS). The new operator is used to test for equality *after evaluation*:

$$\begin{aligned}&(===) :: \mathit{ComplexE} \to \mathit{ComplexE} \to \mathit{Bool}\\&z === w = \mathit{evalE}\ z == \mathit{evalE}\ w\end{aligned}$$

Another law is that *fromCD* is an embedding: if we start from a semantic value, translate it to syntax, and evaluate that syntax we get back to the value we started from.

$$\begin{aligned}&\mathit{propFromCD} :: \mathit{ComplexD} \to \mathit{Bool}\\&\mathit{propFromCD}\ s = \mathit{evalE}\ (\mathit{fromCD}\ s) == s\end{aligned}$$

Other desirable laws are that *Plus* and *Times* should be associative and commutative and *Times* should distribute over *Plus*:

$$\begin{aligned}&\mathit{propAssocPlus}\ x\ y\ z &&= \mathit{Plus}\ (\mathit{Plus}\ x\ y)\ z &&=== \mathit{Plus}\ x\ (\mathit{Plus}\ y\ z)\\&\mathit{propAssocTimes}\ x\ y\ z &&= \mathit{Times}\ (\mathit{Times}\ x\ y)\ z &&=== \mathit{Times}\ x\ (\mathit{Times}\ y\ z)\\&\mathit{propDistTimesPlus}\ x\ y\ z &&= \mathit{Times}\ x\ (\mathit{Plus}\ y\ z) &&=== \mathit{Plus}\ (\mathit{Times}\ x\ y)\ (\mathit{Times}\ x\ z)\end{aligned}$$

These three laws actually fail, but not because of the implementation of *evalE*. We will get back to that later but let us first generalise the properties a bit by making the operator a parameter:

$$\begin{aligned}&\mathit{propAssocA} :: \mathit{Eq}\ a \Rightarrow (a \to a \to a) \to a \to a \to a \to \mathit{Bool}\\&\mathit{propAssocA}\ (+?)\ x\ y\ z = (x +?\ y) +?\ z == x +?\ (y +?\ z)\end{aligned}$$

Note that *propAssocA* is a higher order function: it takes a function (a binary operator name $(+?)$) as its first parameter. It is also polymorphic: it works for many different types a (all types which have an $==$ operator).

Thus we can specialise it to *Plus*, *Times* and other binary operators. In Haskell there is a type class *Num* for different types of "numbers" (with operations $(+)$, $(*)$, etc.). We can try out *propAssocA* for a few of them.

$$\begin{aligned} &propAssocAInt &&= propAssocA\ (+) :: Int &&\to Int &&\to Int &&\to Bool \\ &propAssocADouble &&= propAssocA\ (+) :: Double &&\to Double &&\to Double &&\to Bool \end{aligned}$$

The first is fine, but the second fails due to rounding errors. QuickCheck can be used to find small examples — I like this one best:

$$\begin{aligned} &notAssocEvidence :: (Double, Double, Double, Bool) \\ &notAssocEvidence = (lhs, rhs, lhs - rhs, lhs == rhs) \\ &\quad \textbf{where}\ lhs = (1 + 1) + 1 \,/\, 3 \\ &\qquad\quad rhs = 1 + (1 + 1 \,/\, 3) \end{aligned}$$

For completeness: these are the values:

```
(2.3333333333333335      -- Notice the five at the end
,2.333333333333333,      -- which is not present here.
,4.440892098500626e-16   -- The difference
,False)
```

This is actually the underlying reason why some of the laws failed for complex numbers: the approximative nature of *Double*. But to be sure there is no other bug hiding we need to make one more version of the complex number type: parameterise on the underlying type for $\mathbb{R}$. At the same time we combine *ImagUnit* and *ToComplex* to *ToComplexCart*:

$$\begin{aligned} &\textbf{data}\ ComplexSyn\ r = ToComplexCart\ r\ r \\ &\qquad\qquad\qquad\qquad \mid ComplexSyn\ r :+: ComplexSyn\ r \\ &\qquad\qquad\qquad\qquad \mid ComplexSyn\ r :*: ComplexSyn\ r \\ &toComplexSyn :: Num\ a \Rightarrow a \to ComplexSyn\ a \\ &toComplexSyn\ x = ToComplexCart\ x\ 0 \end{aligned}$$

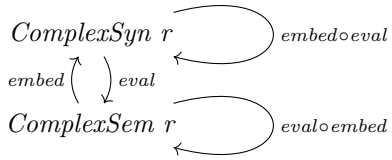
From appendix F we import **newtype** $ComplexSem\ r = CS\ (r, r)$ **deriving** Eq and the semantic operations $(.+.)$ and $(.*.)$ corresponding to *plusD* and *timesD*.

$$\begin{aligned} &evalCSyn :: Num\ r \Rightarrow ComplexSyn\ r \to ComplexSem\ r \\ &evalCSyn\ (ToComplexCart\ x\ y) = CS\ (x, y) \\ &evalCSyn\ (l :+: r) = evalCSyn\ l\ .+.\ evalCSyn\ r \\ &evalCSyn\ (l :*: r)\ = evalCSyn\ l\ .*.\ evalCSyn\ r \end{aligned}$$

From syntax to semantics and back We have seen evaluation functions from abstract syntax to semantics ($eval :: Syn \to Sem$). Often an inverse is also available: $embed :: Sem \to Syn$. For our complex numbers we have

$$\begin{aligned} &embed :: ComplexSem\ r \to ComplexSyn\ r \\ &embed\ (CS\ (x, y)) = ToComplexCart\ x\ y \end{aligned}$$

The embedding should satisfy a round-trip property: $eval\ (embed\ s) == s$ for all semantic complex numbers s. Here is a diagram showing how the types and the functions fit together



Exercise 1.14: What about the opposite direction? When is *embed* (*eval* e) == e?

More about laws Some laws appear over and over again in different mathematical contexts. Binary operators are often associative or commutative, and sometimes one operator distributes over another. We will work more formally with logic in chapter 2 but we introduce a few definitions already here:

Associative $(+) = \forall\ a, b, c.\ \ (a + b) + c = a + (b + c)$

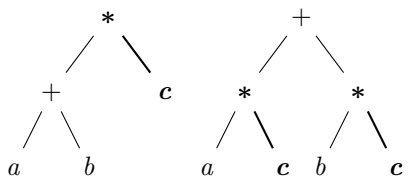
Commutative $(+) = \forall\ a, b.\ \ a + b = b + a$

Non-examples: division is not commutative, average is commutative but not associative.

Distributive $(*)\ (+) = \forall\ a, b, c.\ \ (a + b) * c = (a * c) + (b * c)$

We saw implementations of some of these laws as *propAssocA* and *propDistTimesPlus* earlier, and learnt that the underlying set matters: $(+)$ for $\mathbb{R}$ has some properties, but $(+)$ for *Double* has other. When implementing, approximation is often necessary, but makes many laws false. Thus, we should attempt to do it late, and if possible, leave a parameter to make the degree of approximation tunable (*Int*, *Integer*, *Float*, *Double*, $\mathbb{Q}$, syntax trees, etc.).

To get a feeling for the distribution law, it can be helpful to study the syntax trees of the left and right hand sides. Note that $(*c)$ is pushed down (distributed) to both a and b:



(In the language of section 4.2.1, distributivity means that $(*c)$ is a $(+)$-homomorphism.)

Exercise: Find other pairs of operators satisfying a distributive law.

1.7 Notation and abstract syntax for (infinite) sequences

As a bit of preparation for the language of sequences and limits in later lectures we here spend a few lines on the notation and abstract syntax of sequences.

Common math book notation: $\{a_i\}_{i=0}^{\infty}$ or just $\{a_i\}$ and (not always) an indication of the type X of the a_i. Note that the a at the center of this notation actually carries all of the information: an infinite family of values a_i each of type X. If we interpret "subscript" as function application we can see that $a : \mathbb{N} \to X$ is a useful typing of a sequence. Some examples:

```
type ℕ      = Natural   -- imported from Numeric.Natural
type ℚ⁺     = Ratio ℕ   -- imported from Data.Ratio
type Seq a = ℕ → a
idSeq :: Seq ℕ
idSeq i = i             -- {0, 1, 2, 3, ...}
```

$$
\begin{aligned}
&\mathit{invSeq} :: \mathit{Seq}\ \mathbb{Q}^+ \\
&\mathit{invSeq}\ i = 1\ \%\ (1+i) && \text{-- } \{\tfrac{1}{1}, \tfrac{1}{2}, \tfrac{1}{3}, \tfrac{1}{4}, ...\} \\
&\mathit{pow2} :: \mathit{Num}\ r \Rightarrow \mathit{Seq}\ r \\
&\mathit{pow2} = (2\hat{}) && \text{-- } \{1, 2, 4, 8, ...\} \\
&\mathit{conSeq} :: a \to \mathit{Seq}\ a \\
&\mathit{conSeq}\ c\ i = c && \text{-- } \{c, c, c, c, ...\}
\end{aligned}
$$

What operations can be performed on sequences? We have seen the first one: given a value c we can generate a constant sequence with *conSeq c*. We can also add sequences componentwise (also called "pointwise"):

$$
\begin{aligned}
&\mathit{addSeq} :: \mathit{Num}\ a \Rightarrow \mathit{Seq}\ a \to \mathit{Seq}\ a \to \mathit{Seq}\ a \\
&\mathit{addSeq}\ f\ g\ i = f\ i + g\ i
\end{aligned}
$$

and in general lift any binary operation $op :: a \to b \to c$ to the corresponding, pointwise, operation of sequences:

$$
\begin{aligned}
&\mathit{liftSeq}_2 :: (a \to b \to c) \to \mathit{Seq}\ a \to \mathit{Seq}\ b \to \mathit{Seq}\ c \\
&\mathit{liftSeq}_2\ \mathit{op}\ f\ g\ i = \mathit{op}\ (f\ i)\ (g\ i) && \text{-- } \{\mathit{op}\ (f\ 0)\ (g\ 0), \mathit{op}\ (f\ 1)\ (g\ 1), ...\}
\end{aligned}
$$

Similarly we can lift unary operations, and "nullary" operations:

$$
\begin{aligned}
&\mathit{liftSeq}_1 :: (a \to b) \to \mathit{Seq}\ a \to \mathit{Seq}\ b \\
&\mathit{liftSeq}_1\ h\ f\ i = h\ (f\ i) && \text{-- } \{h\ (f\ 0), h\ (f\ 1), h\ (f\ 2), ...\} \\
&\mathit{liftSeq}_0 :: a \to \mathit{Seq}\ a \\
&\mathit{liftSeq}_0\ c\ i = c
\end{aligned}
$$

Exercice 1.13: what does function composition do to a sequence? For a sequence a what is $a \circ (1+)$? What is $(1+) \circ a$?

Another common mathematical operator on sequences is the limit. We will get back to limits in later sections (2.11, 2.13), but here we just analyse the notation and typing. This definition is slightly adapted from Wikipedia (2017-11-08):

> We call L the limit of the sequence $\{x_n\}$ if the following condition holds: For each real number $\epsilon > 0$, there exists a natural number N such that, for every natural number $n \geqslant N$, we have $|x_n - L| < \epsilon$.
>
> If so, we say that the sequence converges to L and write
>
> $$L = \lim_{i \to \infty} x_i$$

There are (at least) two things to note here. First, with this syntax, the $\lim_{i\to\infty} x_i$ expression form binds i in the expression x_i. We could just as well say that *lim* takes a function $x :: \mathbb{N} \to X$ as its only argument. Second, an arbitrary x, may or may not have a limit. Thus the customary use of $L =$ is a bit of abuse of notation, because the right hand side may not be well defined. One way to capture that is to give *lim* the type $(\mathbb{N} \to X) \to \mathit{Maybe}\ X$. Then $L = \lim_{i\to\infty} x_i$ would mean $\mathit{Just}\ L = \mathit{lim}\ x$ We will return to limits and their proofs in Sec. 2.12 after we have reviewed some logic.

Here we just define one more common operation: the sum of a sequence (like $\sigma = \sum_{i=0}^{\infty} 1/i!$[4]). Just as not all sequences have a limit, not all have a sum either. But for every sequence we can define a new sequence of partial sums:

[4] Here $n! = 1 * 2 * ... * n$ is the factorial (sv: fakultet).

$$
\begin{aligned}
&\mathit{sums} :: \mathit{Num}\ a \Rightarrow \mathit{Seq}\ a \to \mathit{Seq}\ a \\
&\mathit{sums} = \mathit{scan}\ (+)\ 0
\end{aligned}
$$

The function *sums* is perhaps best illustrated by examples:

$$
\begin{aligned}
&\mathit{sums}\ (\mathit{conSeq}\ c) &&== \{0, c, 2 * c, 3 * c, ...\} \\
&\mathit{sums}\ (\mathit{idSeq}) &&== \{0, 0, 1, 3, 6, 10, ...\}
\end{aligned}
$$

The general pattern is to start at zero and accumulate the sum of initial prefixes of the input sequence. The definition of *sums* uses *scan* which is a generalisation which "sums" with a user-supplied operator $(+)$ starting from an arbitrary z (instead of zero).

$$
\begin{aligned}
&\mathit{scan} :: (b \to a \to b) \to b \to \mathit{Seq}\ a \to \mathit{Seq}\ b \\
&\mathit{scan}\ (+)\ z\ a = s \\
&\quad \textbf{where}\ s\ 0 = z \\
&\qquad\quad s\ i = s\ (i-1) + a\ i
\end{aligned}
$$

And by combining this with limits we can state formally that the sum of a sequence a exists and is S iff the limit of *sums* a exists and is S. As a formula we get $\mathit{Just}\ S = \mathit{lim}\ (\mathit{sums}\ a)$, and for our example it turns out that it converges and that $\sigma = \sum_{i=0}^{\infty} 1/i! = e$ but we will not get to that until Sec. 8.1.

We will also return to limits in Sec. 3.3 about derivatives where we explore variants of the classical definition

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

To sum up this subsection, we have defined a small Domain Specific Language (DSL) for infinite sequences by defining a type (*Seq* a), some operations (*conSeq*, *addSeq*, $\mathit{liftSeq}_1$, *sums*, *scan*, ...) and some "run functions" or predicates (like *lim* and *sum*).

1.8 Exercises: Haskell, DSLs and complex numbers

Exercise 1.1. Consider the following data type for simple arithmetic expressions:

```
data Exp = Con Integer
         | Exp `Plus` Exp
         | Exp `Minus` Exp
         | Exp `Times` Exp
  deriving (Eq, Show)
```

Note the use of "backticks" around *Plus* etc. which makes it possible to use a name as an infix operator.

1. Write the following expressions in Haskell, using the *Exp* data type:
   (a) $a_1 = 2 + 2$
   (b) $a_2 = a_1 + 7 * 9$
   (c) $a_3 = 8 * (2 + 11) - (3 + 7) * (a_1 + a_2)$
2. Create a function $eval :: Exp \to Integer$ that takes a value of the *Exp* data type and returns the corresponding number (for instance, $eval\ ((Con\ 3)\ `Plus`\ (Con\ 3)) == 6$). Try it on the expressions from the first part, and verify that it works as expected.
3. Consider the following expression:

   $$c_1 = (x - 15) * (y + 12) * z$$
   $$\textbf{where } x = 5; y = 8; z = 13$$

   In order to represent this with our *Exp* data type, we are going to have to make some modifications:
   (a) Update the *Exp* data type with a new constructor *Var String* that allows variables with strings as names to be represented. Use the updated *Exp* to write an expression for c_1 in Haskell.
   (b) Create a function $varVal :: String \to Integer$ that takes a variable name, and returns the value of that variable. For now, the function just needs to be defined for the variables in the expression above, i.e. *varVal* `"x"` should return 5, *varVal* `"y"` should return 8, and *varVal* `"z"` should return 13.
   (c) Update the *eval* function so that it supports the new *Var* constructor, and use it get a numeric value of the expression $c1$.

Exercise 1.2. We will now look at a slightly more generalized version of the *Exp* type from the previous exercise:

```
data E2 a = Con a
          | Var String
          | E2 a `Plus` E2 a
          | E2 a `Minus` E2 a
          | E2 a `Times` E2 a
  deriving (Eq, Show)
```

The type has now been parametrized, so that it is no longer limited to representing expressions with integers, but can instead represent expressions with any type. For instance, we could have an *E2 Double* to represent expressions with doubles, or an *E2 ComplexD* to represent expressions with complex numbers.

1. Write the following expressions in Haskell, using the new *E2* data type.

   (a) $a_1 = 2.0 + a$

   (b) $a_2 = 5.3 + b * c$

   (c) $a_3 = a * (b + c) - (d + e) * (f + a)$

2. In order to evaluate these expressions, we will need a way of translating a variable name into the value. The following table shows the value of each variable in the expressions above:

Name	a	b	c	d	e	f
Value	1.5	4.8	2.4	7.4	5.8	1.7

   In Haskell, we can represent this table using a value of type $Table\ a = Env\ String\ a = [(String, a)]$, which is a list of pairs of variable names and values, where each entry in the list corresponds to a column in the table.

   (a) Express the table above in Haskell by creating *vars* :: *Table Double*.

   (b) Create a function *varVal* :: $Table\ a \to String \to a$ that returns the value of a variable, given a table and a variable name. For instance, *varVal vars* `"d"` should return 7.4

   (c) Create a function *eval*::$Num\ a \Rightarrow Table\ a \to E2\ a \to a$ that takes a value of the new *E2* data type and returns the corresponding number. For instance, *eval vars* ((*Con* 2)‘*Plus*‘ (*Var* `"a"`)) == 3.5. Try it on the expressions from the first part, and verify that it works as expected.

Exercise 1.3. *From exam 2017-08-22*

A semiring is a set R equipped with two binary operations $+$ and $\cdot$, called addition and multiplication, such that:

- $(R, +, 0)$ is a commutative monoid with identity element 0:

  $$(a + b) + c = a + (b + c)$$
  $$0 + a = a + 0 = a$$
  $$a + b = b + a$$

- $(R, \cdot, 1)$ is a monoid with identity element 1:

  $$(a \cdot b) \cdot c = a \cdot (b \cdot c)$$
  $$1 \cdot a = a \cdot 1 = a$$

- Multiplication left and right distributes over $(R, +, 0)$:

  $$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$
  $$(a + b) \cdot c = (a \cdot c) + (b \cdot c)$$
  $$a \cdot 0 = 0 \cdot a = 0$$

1. Define a datatype *SR v* for the language of semiring expressions (with variables of type *v*). (These are expressions formed from applying the semiring operations to the appropriate number of arguments, e.g., all the left hand sides and right hand sides of the above equations.)

2. Give a type signature for, and define, a general evaluator for *SR v* expressions on the basis of an assignment function.

Exercise 1.4. *From exam 2016-03-15*

A *lattice* is a set L together with two operations $\vee$ and $\wedge$ (usually pronounced "sup" and "inf") such that

- $\vee$ and $\wedge$ are associative:

$$\forall\, x, y, z \in L.\ \ (x \vee y) \vee z = x \vee (y \vee z)$$
$$\forall\, x, y, z \in L.\ \ (x \wedge y) \wedge z = x \wedge (y \wedge z)$$

- $\vee$ and $\wedge$ are commutative:

$$\forall\, x, y \in L.\ \ x \vee y = y \vee x$$
$$\forall\, x, y \in L.\ \ x \wedge y = y \wedge x$$

- $\vee$ and $\wedge$ satisfy the *absorption laws*:

$$\forall\, x, y \in L.\ \ x \vee (x \wedge y) = x$$
$$\forall\, x, y \in L.\ \ x \wedge (x \vee y) = x$$

1. Define a datatype for the language of lattice expressions.
2. Define a general evaluator for *Lattice* expressions on the basis of an assignment function.

Exercise 1.5. *From exam 2016-08-23*

An *abelian monoid* is a set M together with a constant (nullary operation) $0 \in M$ and a binary operation $\oplus : M \to M \to M$ such that:

- 0 is a unit of $\oplus$

$$\forall\, x \in M.\ \ 0 \oplus x = x \oplus 0 = x$$

- $\oplus$ is associative

$$\forall\, x, y, z \in M.\ \ x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

- $\oplus$ is commutative

$$\forall\, x, y \in M.\ \ x \oplus y = y \oplus x$$

1. Define a datatype *AbMonoidExp* for the language of abelian monoid expressions. (These are expressions formed from applying the monoid operations to the appropriate number of arguments, e.g., all the left hand sides and right hand sides of the above equations.)
2. Define a general evaluator for *AbMonoidExp* expressions on the basis of an assignment function.

Exercise 1.6. Read the full chapter and complete the definition of the instance for *Num* for the datatype *ComplexSyn*. Also add a constructor for variables to enable writing expressions like (*Var* `"z"`) :∗: *toComplex* 1.

Exercise 1.7. Read the next few pages of Appendix I (in [Adams and Essex, 2010]) defining the polar view of Complex Numbers and try to implement complex numbers again, this time based on magnitude and phase for the semantics.

Exercise 1.8. Implement a simplifier $simp :: ComplexSyn\ r \to ComplexSyn\ r$ that handles a few cases like $0 * x = 0$, $1 * x = x$, $(a + b) * c = a * c + b * c$, ... What class context do you need to add to the type of *simp*?

Exercise 1.9. Functions and pairs (the "tupling transform"). From one function $f :: a \to (b, c)$ returning a pair, you can always make a pair of two functions $pf :: (a \to b, a \to c)$. Implement this transform:

$$f2p :: (a \to (b, c)) \to (a \to b, a \to c)$$

Also implement the opposite transform:

$$p2f :: (a \to b, a \to c) \to (a \to (b, c))$$

This kind of transformation if often useful, and it works also for n-tuples.

Exercise 1.10. There is also a "dual" to the tupling transform: to show this, implement these functions:

$$s2p :: (Either\ b\ c \to a) \to (b \to a, c \to a)$$
$$p2s :: (b \to a, c \to a) \to (Either\ b\ c \to a)$$

Exercise 1.11. Counting values. Now assume we have *f2p*, *s2f*, etc used with three finite types with cardinalites A, B, and C. (For example, the cardinality of *Bool* is 2, the cardinality of *Weekday* is 7, etc.) Then what is the cardinality of $Either\ a\ b$? (a, b)? $a \to b$? etc. These rules for computing the cardinality suggests that *Either* is similar to sum, $(,)$ is similar to product and $(\to)$ to (flipped) power. These rules show that we can use many intuitions from high-school algebra when working with types.

Exercise 1.12. Functions as tuples. For any type t the type $Bool \to t$ is basically "the same" as the type (t, t). Implement the two functions *isoR* and *isoL* forming an isomorphism:

$$isoR :: (Bool \to t) \to (t, t)$$
$$isoL :: (t, t) \to (Bool \to t)$$

and show that $isoL \circ isoR = id$ and $isoR \circ isoL = id$.

Exercise 1.13. From section 1.7:

- What does function composition do to a sequence? (composition on the left?, on the right?)
- How is $liftSeq_1$ related to *fmap*? $liftSeq_0$ to *conSeq*?

Exercise 1.14. When is $embed\ (eval\ e) == e$?

Step 0: type the quantification: what is the type of e?

Step 1: what equality is suitable for this type?

Step 2: if you use "equality up to eval" — how is the resulting property related to the first round-trip property?