Dyadic arithmetic in the book of elements

Now discrete mathematics is thought about in the technological context of modern computers, and we struggle to understand the connection with weaving. However, if we look back, we realise that the birth of discrete mathematics took place when weaving was the predominant technology. The hypothesis of Klück (TODO: cross ref Ellen's article), which lies at the core of the present special issue, is that discrete mathematics began with thought processes of weavers, and that this is implicit in the system of counting defined and applied in Euclid's Elements. To engage with this hypothesis, we attempted to implement this system of counting in a contemporary programming language.

We will make this attempt using the Haskell programming language, known for its very strong focus on defining the types of things in clear way. We will approach the definitions given in book VII in turn, in the following, to see how far we can get, and what problems arise.

Definition 1: A unit is that by virtue of which each of the things

that exist is called one.

```
data Unit = Unit
```

The first definition appears straightforward, but allows us to introduce our first piece of Haskell code, which simply defines a *data type*. On the left is given the name of the data type, in this case `Unit`, and on the right all the possible instances of that type, which here is again called `Unit`.

However, the above already has an error, in that in ancient Greece, a unit would not be thought of as abstract in this way; you could think about a sheep unit, or a cow unit, but ancient Greeks would find it nonsensical to think of a unit as being independent of such a category. Haskell allows us to model this as this by adding a parameter `a` for the type:

```
data Unit a = forall a. (Unit a)
```

As the `forall` suggests, the type `a` can represent any other type that we might define, such as `Sheep` or `Cow`. In practice this type parameter does nothing, apart from indicate that a `Unit` is thought about with reference to a concrete type of thing. So to model an unit of Sheep, we could do the following:

```
data Sheep = Sheep {colour :: String}

sheep = Unit (Sheep "white")
```

However we are not interested in `Unit`s having a particular identity here, so we will use a definition which specifies a type parameter, but does not require a value when a Unit instance is being created:

```haskell
data Unit a = forall a. Unit

sheep :: Unit Sheep
sheep = Unit
```

Our sheep here still has the type of `Unit Sheep`, but does not define anything about a particular sheep.

Definition 2: A number is a multitude composed of units.

The second definition again appears straightforward; in code terms we can think of a multitude as a list, which is denoted by putting the `Unit` datatype in brackets:

```haskell
type Multitude a = [Unit a]
```

Note that the same type parameter is used in `Unit a` and `Multitude a`, which means that you for example can't mix `Sheep` and `Cow`s in the same multitude.

However again, the detail makes this a little bit complex. Firstly, such a list is allowed to be empty, whereas the number zero is not represented by this definition. Indeed, neither is the number one – a multitude begins with two, less than that is not considered to be a number.

A way around this is to define a multitude relative to the number two, as a pair of units:

```haskell
data Multitude a = Pair (Unit a) (Unit a)
    | Next (Unit a) (Multitude a)
```

The number four would then be constructed with the following:

```haskell
four = Next Unit (Next Unit (Pair Unit Unit))
```

It would be nice if we could 'see' this multitude more clearly. We can visualise it by telling Haskell to show a `Unit` with an `x` and by stringing together the units across instances of `Pair` and `Next`:

~~{.haskell .colourtex} instance Show (Unit a) where show x = "x"

instance Show (Multitude a) where show (Pair u u') = show u ++ show u' show (Next u n) = show u ++ show n ~~{.haskell .colourtex}

Then the number `four` is shown like this (here the `>` prefixes the expression `four`, and the result is shown below):

```
> four
xxxx
```

Here is a handy function for turning integers into multitudes, which first defines the conversion from 2, and then the general case, based upon that.

```
fromInt 2 = Pair Unit Unit
fromInt n | n < 2 = error "There are no multitudes < 2"
          | otherwise = Next Unit $ fromInt (n-1)
```

Definition 3. A number is a part of a number, the less of the greater, when it measures the greater

Definition 3 is a little more complex, and because a single unit is not a number, awkward to express in contemporary programming languages. Lets begin by defining the cases where `lesser` is true, or otherwise false:

```
lesser (Pair _ _) (Next _ _) = True
lesser (Next _ a) (Next _ b) = lesser a b
lesser _ _ = False

greater a b = lesser b a
```

We can define the equality operator `==` with the same approach:

```
instance Eq (Multitude a) where
  (==) (Pair _ _) (Pair _ _) = True
  (==) (Next _ a) (Next _ b) = a == b
  (==) _ _ = False
```