1 Numbers

1.1 What are numbers and what should they be?

In his famous article "What are numbers and what should they be?" German mathematician Richard Dedekind (1831 – 1916) described numbers as "free creations of the human mind". This idea is anything but self-evident. Dedekind's contemporary and fellow combatant for a new approach in math, namely set theory, Georg Cantor fiercely defended *mathematical platonism*, the idea that mathematical objects are real existing and that mathematical methods are tools that allow us to see these objects like, perhaps, telescopes, enable us to examine stars far away in the universe.

This view clashed with the view of the *constructivist* Leopold Kronecker, a resolute opponent to the novelties introduced by Cantor, Dedekind and others. In the view of the constructivists like Kronecker only those objects that can be constructed in a finite number of steps from a finite number of integers are valid objects of mathematics. Anything else is a chimaera. "The natural numbers are made by god", as he put it, "all else is the work of man."

Both views, constructivism and platonism, even though not mainlines of discussion anymore, are still influential today. The influence of platonism is visible, when it is said that some mathematician has "discovered" a concept, *e.g.* "Gauss discovered a construction of the heptadecagon". Mathematicians in the tradition of constructivism or its modern variant *intuitivism*, insist in a terminology that stresses that mathematical concepts are man-made. They would say Gauss has "invented", "developed" or, simply, "constructed".

It is consensus, however, that, independent of the question whether numbers are real existing things or invented concepts, there must be some kind of representation of numbers in our mind. It is also likely that these representations have some grounding in the nervous system and are thus part of our genetic legacy. A strong evidence therefor is the fact that some animals can distinguish sets of different numbers of objects. Birds, for instance, realise when an egg is missing in their nest, but do not necessarily realise that an egg was replaced by another one. That would suggest that the "creation" of numbers is not as "free" as it appears to be according to Dedekind. The ability to work with numbers would then be part of our biology, just like having language capability is part of our biology.

When we try to reduce numbers to the very heart of what they are, we quickly come to the notion of counting. The assumption that numbers basically represent steps in the process of counting is supported by the fact that many number systems in use during history are modelled on something countable like fingers (leading to the decimal number system), fingers and toes (leading to the vigesimal system), finger bones (leading to systems based on multiples of 12) and so on. According to this view, relations between quantities (length, volume, weight, *etc.*) would represent more abstract concepts engineered in some way on top of the fundamental notion of counting. It is not so clear, however, if this hierarchy corresponds to the real historic development. Among the oldest known math problems are such as determining the quanitity of corn for baking bread in acient Egypt or investigating rhythmic patterns formed by single and multiple beats. Problems of this kind cannot be solved by mere counting, but demand comparison of *continuous* quantities like volume, weight or time intervals leading immediately to fractions. Numbers may hence be related to a human sense of quantity that is more general than counting.

Numbers appear to be very simple and clear concepts. But, in fact, together with the operations defined on them, addition, multiplication, subtraction and division, they quickly give rise to intricate problems. Those problems are not like add-ons to the numbers that were invented later to complicate things, they belong to the very core of the number concept. It appears, in fact, that those problems have already slumbered in the apparently simple concept, but needed some time to unfold themselves – centuries in some cases, millennia in others. Even if the number concept may have been created by the human mind, once fixed, all its consequences and complications are there. In this sense, every mathematician would agree to speak of "discovering" a problem in a certain field of math. Mathematicians may still disagree, however, on whether the solution to such a problem was "discovered" or "developed".

The most notorious group of long-standing problems is number theory, which deals mainly with questions of prime numbers, *i.e.* natural numbers that cannot be constructed from other natural numbers using multiplication. Prime numbers are investigated for some three thousand years now, but, still, many fundamental questions remain unanswered. Infamous are the number-theoretic theorems drafted, but not proven, by Pierre de Fermat in the 17^{th} century. Most of these theorems look quite innocent, but by proving them in the course of the following three centuries, math made huge progress developing methods that had been unknown to Fermat and his contemporaries.

The most insistent problem, known as *Fermat's Last Theorem*, was solved only in the late 20^{th} century by British mathematician Andrew Wiles. What makes this specific problem so mind-boggling is its extremely simple formulation – compared to the proof of some 200 pages. Indeed, Fermat doodled the theorem on the margin of a page in his copy of an acient math book, Diophantos' "Arithmetica", and, to the irritation of generations of mathematicians, he added: "I have found a really wonderful proof, but, unfortunately, this margin is too narrow to contain it." The theorem in question states that there is no solution for equations of the form: $a^z + b^z = c^z$ for a, b, c and z all natural numbers and $z > 2$. For $z = 2$, many solutions exist, the so called *Pythagorean triples*, *e.g.*: $3^2 + 4^2 = 5^2$.

Another set of difficult problems is algebra and the search for solutions of higher-degree equations, which caused a lot of headache throughout the centuries. The general solution for quadratic equations of the form $ax^2+bx+c=0$ was known since antiquity. Solutions for cubic and quartic equations were found by joint efforts and fierce competition in the 16th century by Italian mathematicians, namely Niccolò Fontana Tartaglia, Gerolamo Cardano and Lodovico Ferrari. But it took until the early 19th century, before mathematicians were able to make statements about quintic and even higher-degree equations. In fact, the investigations took an unpredicted direction and led to the development of a new branch of mathematics, *group theory*, which studies sets of objects – not only numbers – and operations defined on these objects, and *abstract algebra*, which generalises group theory making objects of completely different areas of mathematics comparable.

Group theory, which was developed during the 18th and 19th century by Joseph-Louis Lagrange, Paolo Ruffini, young Niels Henrik Abel and the incredible Évariste Galois, is a great helper in organising the number zoo that developed out of simpler notions of numbers in the course of the time. At the beginning – if we believe in numbers being fundamentally related to counting – there may have been only the concept of natural numbers, which was then extended by adding fractions to cope with relations between quantities and negative numbers to deal with negative quantities like debts. The investigations into geometry by Greek mathematicians led to the rise of irrational numbers – such as the number π – which, as a surprise to acient mathematicians, cannot be expressed in terms of ratios of integers. The studies in algebra led to complex numbers, which, in their turn, inspired the construction of hypercomplex numbers.

In this chapter, we will start with the simplest possible number concept, natural numbers, and we will stick to it as long as possible – perhaps longer. We then will make a big leap introducing negative numbers and fractions. Afterwards we enter the confusing world of irrational numbers and will probably understand the annoyance of mathematicians like Kronecker when faced with mathematical concepts dealing with these strange creatures. Later, complex numbers and even more exotic beasts will come into focus. On the way, we will introduce some basic group theory helping us to find our way in the number jungle.

1.2 Peano Numbers

The Italian mathematician Giuseppe Peano (1858 – 1932) defined an axiomatic system to describe the natural numbers and their properties. An axiomatic system consists of axioms, statements that are known or simply assumed to be true, and rules that describe how other statements can be derived from axioms, such that these new statements are true if the axioms are true. In practice the procedure is usually followed the other way round: one would try to find a sequence of applications of the rules that links a given statement with one or more axioms. This process is called a *proof* and is one of the main things mathematicians do to kill their time.

A major part of the discussions about the foundations of math in the first half of the 20th century was about the idea formulated by David Hilbert to construct the whole of mathematics as an axiomatic system and, then, to prove that for every statement that is believed to be true a sequence of rule applications can be found that derives this statement from the axioms. The plan failed in the 1930ies, after releasing an incredible amount of mathematical creativity that resulted, among other things, in a theoretical model of a universal computing device, known today as the *Turing machine* and in the *lambda calculus*, which, as already discussed, is one of the foundations of functional programming. The first task of the Turing machine and of the lambda calculus was indeed to prove that Hilbert's plan is impossible.

Peano's objective was in Hilbert's line: to provide a foundation of natural numbers as a first step towards an axiomatisation of the whole field of arithmetic. In spite of this ambitious goal, Peano's axioms are quite simple. The basic idea is to define natural numbers by two elements: The explicitly defined number *Zero* and a recursive function *Successor* that defines any other number. Peano's axioms boil down to a formulation in Haskell like:

```
data Peano = Zero | S Peano
  deriving Show
```

This captures very well the process of counting. Instead of adding 1 to a given number, we just derive its successor:

```
succ :: Peano → Peano
succ p = S p
```

For instance, the successor of *Zero*, one, is: $S(Zero)$; two is $S(S(Zero))$, three is $S(S(S(Zero)))$ and so on.

We can also define a function to count backwards, *i.e.*:

```
pre :: Peano → Peano
pre Zero  = ⊥
pre (S p) = p
```

Zero, this is one of Peano's axioms, has no predecessor. The predecessor of any other number, is that number with one S removed. The predecessor of $S(S(S(S(S(Zero)))))$, five, for instance, is $S(S(S(S(Zero))))$, four.

We can also devise a simple addition function:

```
add :: Peano → Peano → Peano
add Zero a  = a
add a Zero  = a
add (S a) b = add a (S b)
```

Subtraction is implemented easily as well:

$$\begin{aligned}
&sub :: Peano \to Peano \to Peano\\
&sub\ a\ Zero \qquad\ \ = a\\
&sub\ Zero\ a \qquad\ \ = \bot\\
&sub\ (S\ a)\ (S\ b) = sub\ a\ b
\end{aligned}$$

Note that any number reduced by *Zero* is just that number. *Zero* reduced by any number but *Zero*, however, is undefined. For all other cases, we just reduce a and b by one S until we hit one of the base cases.

We could go on and define multiplication, division and all other arithmetic operations based on this notion of numbers. It is indeed convincing in its simplicity and used as a standard system for research into arithmetic until today. But it is not very convenient, especially when working with huge numbers. Peano numbers are unary numbers, that is, to represent the number 100, one has to write 100 symbols (in fact, 101 symbols: 100 S and 1 *Zero*). As a number system, Peano numbers are even less handy than the roman numerals, which introduce symbols at least for some greater values, such as V, X, L and C. A trick that is often used in literature to mitigate this shortcoming is to add a subscript number to the S symbol to make clear, how many Ses we would have to write to represent this value, for instance, $S_5(Zero)$ would be 5. But, of course, that makes the use of Peano numbers – as a number system – pointless. Already Peano was faced with the clumsiness of his axioms when used as a number system: he tried to use it as a didactic device in his teaching both at the university and at the military academy where he was working. In the case of the military academy, this led to desaster and, eventually, to his dismissal in 1901. His achievements as mathetmatician and logician, however, were respected in the scientific community worldwide.

Let us learn from Peano's didactic failure and look out for a more practical number system, one that allows us to use significantly fewer symbols than the value of the number we want to represent. A system that *scales* in this sense is our well-known decimal number system.

1.3 Decimal Numbers

A numeral system consists of a vocabulary of symbols, which we will call *digits*, rules that define how to compose digits to strings and a model that leads to an arithmetic interpretation of such strings. To make practical use of the numeral system, we must also define a set of basic operations, such as counting forward and backward, addition, subtraction, multiplication, division and whatever we want to do with our numbers.

We define the following vocabulary:

```
data Digit = Zero | One | Two   | Three | Four |
             Five | Six | Seven | Eight | Nine
  deriving (Show, Eq, Ord)
```

Numbers are lists of *Digit*s:

```
type Number = [Digit]
```

Some numbers that are used more often than all others are:

```
zero, unity, two, ten :: Number
zero  = [Zero]
unity = [One]
two   = [Two]
ten   = [One, Zero]
```

Here is the first basic operation, the *successor*, which we already know from Peano's axioms. To avoid confusion with the *succ* function in Haskell's *Prelude*, we will call it *next*:

```
next :: Number → Number
next [] = []
```

This is already the first important decision. We define the *next* of the empty list is the empty list. That implies that *nothing* is something different from *Zero*. We could enter difficult philosophical discussions about this statement. The decision, however, is mainly pragmatic: we need a base case for processing lists and this base case is just the empty list.

The next successors are straight forward:

```
next [Zero]  = [One]
next [One]   = [Two]
next [Two]   = [Three]
next [Three] = [Four]
next [Four]  = [Five]
next [Five]  = [Six]
next [Six]   = [Seven]
next [Seven] = [Eight]
next [Eight] = [Nine]
```

Now it gets interesting:

```
next [Nine] = [One, Zero]
```

Note that we need one more digit to represent the successor of the 10^{th} digit! The first place, read from right to left, returns to *Zero* and the second place goes up from *nothing* to *One*. This latter wording shows that our decision, concerning the empty list, is not so innocent as it may appear at the first sight!

Now we have to define how to proceed with the successor of numbers consisting of more than one digit:

$$\mathit{next}\ (\mathit{Zero} : \mathit{ds}) = \mathit{next}\ \mathit{ds}$$

The first thing we do is to check if the head is $Zero$. In this case, we just reduce to the rest of the list, that is: a leading $Zero$ does not change the value of a number. In all other cases:

$$\begin{aligned}&\mathit{next}\ \mathit{ds} = \textbf{case}\ \mathit{last}\ \mathit{ds}\ \textbf{of}\\ &\qquad\qquad \mathit{Nine} \rightarrow \mathit{next}\ (\mathit{init}\ \mathit{ds}) +\!\!+ [\mathit{Zero}]\\ &\qquad\qquad d \quad\ \ \rightarrow \qquad \mathit{init}\ \mathit{ds}\ +\!\!+ \mathit{next}\ [d]\end{aligned}$$

If the last digit of the number is $Nine$, we concatenate the successor of the number without the last digit (*init*) and [$Zero$]. The point is that the successor of $Nine$, as we have defined it above, is [$One, Zero$]. The last digit of the new number, hence, will be $Zero$ appended to the successor of the initial part. If the last number of the initial part is again $Nine$, we repeat the whole process on the number except the last digit. Example: the successor of the number [$Nine, Nine$] is

$\mathit{next}\ [\mathit{Nine}] +\!\!+ [\mathit{Zero}]$
$[\mathit{One}, \mathit{Zero}] +\!\!+ [\mathit{Zero}]$
$[\mathit{One}, \mathit{Zero}, \mathit{Zero}]$.

For the case that the last digit is not $Nine$, the process is much simpler: we just replace the last digit by its successor. The successor of [$Nine, Eight$], hence, is:

$[\mathit{Nine}] +\!\!+ \mathit{next}\ [\mathit{Eight}]$
$[\mathit{Nine}] +\!\!+ [\mathit{Nine}]$
$[\mathit{Nine}, \mathit{Nine}]$.

Note that this representation of numbers is not optimised for efficient processing. Haskell is not very good at accessing the last element of a list. There are many ideas to speed this up. An idea that suggests itself is to turn numbers around – relative to our usual reading direction – starting with the least siginificant digit, *e.g.* writing [$Zero, One$] instead of [$One, Zero$] to represent the number 10. We could also use a data type – such as the vector type – that allows for fast random access to all its elements. But this kind of optimisations would be better discussed in a Haskell tutorial.

The next basic operation is counting backwards. We start just as we started with *next*:

$$\begin{aligned}&\mathit{prev} :: \mathit{Number} \rightarrow \mathit{Number}\\ &\mathit{prev}\ [\,] = [\,]\end{aligned}$$

But we now have an important difference:

$$\mathit{prev}\ [\mathit{Zero}] = \bot$$

We cannot count below *Zero*! Any attempt to do so will result in an error. We have to take care of this in all operations we will design in the future.

Counting backwards for the digits from *One* to *Nine*, however, is straight backward:

```
prev [One]   = [Zero]
prev [Two]   = [One]
prev [Three] = [Two]
prev [Four]  = [Three]
prev [Five]  = [Four]
prev [Six]   = [Five]
prev [Seven] = [Six]
prev [Eight] = [Seven]
prev [Nine]  = [Eight]
```

But what happens with numbers with more than one digit? First we ignore leading *Zero*s:

```
prev (Zero : ds) = prev ds
```

For all other cases, we use a strategy very similar to the one we used for *next*:

```
prev ds = case last ds of
            Zero → case init ds of
                     [One] → [Nine]
                     ds′   → prev ds′ ++ [Nine]
            d    → init ds ++ prev [d]
```

If the last digit is *Zero*, the last digit of the new number will be *Nine* and the initial part of this number will be its predecessor. If the initial part is just [*One*], its predecessor would be *zero*, which we can ignore for this case. The predecessor of [*One*, *Zero*], hence, is [*Nine*] (not [*Zero*, *Nine*]). If the number is [*One*, *Zero*, *Zero*], the last digit will be *Nine*, which is then appended to the predecessor of [*One*, *Zero*], whose predecessor, as we know already, is [*Nine*]. The result hence is [*Nine*, *Nine*].

For the case that the last digit of the number is not *Zero*, we just append its predecessor to the initial part of the number and we are done. The predecessor of [*Nine*, *Nine*], hence, is just

```
[Nine] ++ prev [Nine]
[Nine] ++ [Eight]
[Nine, Eight].
```

Let us now look at how to add numbers. We start with the same logic we already encountered with Peano Numbers, *i.e.* we add by counting one number up and the other, simultaneously, down until we reach a base case:

```
add :: Number → Number → Number
add a []      = a
add [] b      = b
add a [Zero]  = a
add [Zero] b  = b
add a b       = next a `add` (prev b)
```

That is, any number added to $[Zero]$ (or to the empty list []) is just that number. In all other cases, addition of two numbers a and b is defined recursively as counting a up and b down. When we hit the base case, *i.e.* b reaches $[Zero]$, we have a result.

How many steps would we need to add two numbers this way? Well, that depends directly on the size of b. We will count a up, until b is $[Zero]$. For $b = 100$, we need 100 steps, for $b = 1\,000\,000$, we need 1 000 000 steps. Is there a way to improve on that? Yes, of course! We can just apply the same logic we have used for *next* and *prev*, that is adding single-digit numbers and handling the carry properly. Here is a solution:

```
add2 :: Number → Number → Number
add2 as bs = reverse $ go (reverse as) (reverse bs)
  where go [] ys            = ys
        go xs []            = xs
        go (x : xs) (y : ys) = case add [x] [y] of
                                 [_, r] → r : go xs (go ys [One])
                                 [r]    → r : go xs ys
                                 _      → ⊥
```

We see at once that the logic has changed significantly. First, we suddenly appear to care for efficiency: we reverse the lists before processing them! This, however, is not only for efficiency. We now process the number digit by digit starting with the least significant one, that is we look at the number not as a number, but as a list of digits – we exploit the structure of the data type.

Accordingly, we do not handle the base case $[Zero]$ anymore, we are now concerned with the base case [], since this is the point, when we have consumed all elements of one of the lists. Until we reach the base case, we just add digit by digit. If the result is a number with two digits – note that we can never get to a number with more than two digits by adding two digits – we insert the less significant digit at the head of the list that will be created by continuing the process. We continue with the next step of *go*, but increase one of the numbers, the second one, by [One]. This is the *add carry* that takes care of the the most significant digit in the two-digit result. Again, by adding two digits, we will never get to a number with a digit greater than *One* in the first position. The greatest possible number, in fact, is $[Nine] + [Nine] = [One, Eight]$.

In the other case where the addition of the two digits results in a number with just one digit, we insert the result at the head of the list that is constructed by the regular continuation of *go* – here, we do not have to take care of any carry.

The final line of the function is just to avoid a warning from the Haksell compiler. It does not add any meaning to the definition of *add2*.

Let us look at an example, say, the addition of $765 + 998 = 1763$. We first reverse the lists, that is, we start with $[Seven, Six, Five]$ and $[Nine, Nine, Eight]$, but call *go* with $[Five, Six, Seven]$ and $[Eight, Nine, Nine]$:

```
go [Five, Six, Seven] [Eight, Nine, Nine]
Three : go [Six, Seven] (go [Nine, Nine] [One])
Three : go [Six, Seven] (Zero : go [Nine] (go [] [One])
Three : go [Six, Seven] (Zero : go [Nine] [One])
Three : go [Six, Seven] (Zero : Zero : go [] (go [] [One]))
Three : go [Six, Seven] (Zero : Zero : go [] [One])
Three : go [Six, Seven] [Zero, Zero, One]
Three : Six : go [Seven] [Zero, One]
Three : Six : Seven : go [] [One]
Three : Six : Seven : [One]
```

The computation results in $[Three, Six, Seven, One]$, which reversed is $[One, Seven, Six, Three]$ and, hence, the correct resut.

So, how many steps do we need with this approach? We have one addition per digit in the smaller number plus one addition for the cases where the sum of two digits results in a two-digit number. For the worst case, we, hence, have $d + d = 2d$ steps, where d is the number of digits of the smaller number. When we add two numbers in the order of hundreds (or, more precisely, with the smaller number in the order of hundreds), we would have three additions (one for each digit of a number in the order of hundreds) plus, in the worst case, three add carries. Translated into steps of *next*, this would be for each single-digit addition in the worst case nine steps (any addition with $[Nine]$) and one *next* step per carry (since carry is always an addition of $[One]$). The worst case in terms of *next*, hence, is $9d + d = 10d$, for d the number of digits in the smaller number. For numbers in the order of millions, this amounts to $10 \times 7 = 70$ steps, compared to 1 000 000 steps for the naïve approach.

We could even go on and reduce the worst case of 9 *next* steps per addition to one single step, just by doing to the algorithm what they do to us in school: instead of using *next* for addition we can define addition tables for our 10 digits, *i.e.*

```
add [Zero] a = a
add [One] [One] = [Two]
add [One] [Two] = [Three]
add [One] [Three] = [Four]
...
```

But that approach is quite boring and, therefore, we will not go for it. Instead, we will look at subtraction. First, we implement the naïve approach that we need for subtraction of digits:

```
sub :: Number → Number → Number
sub a [Zero]              = a
sub a b | a ‘cmp‘ b ≡ LT = ⊥
        | otherwise       = prev a ‘sub‘ (prev b)
```

Subtracting *zero* from any number is just that number. For other cases, we first have to compare the numbers. If the first number is less than the second, the result is undefined for natural numbers. Otherwise, we just count the two number down by one and continue until we hit the base case.

We will look at the comparison function *cmp* below. We will first define the more sophisticated version of subtraction for numbers with more than one digit:

```
sub2 :: Number → Number → Number
sub2 as bs | as ‘cmp‘ bs ≡ LT = ⊥
           | otherwise         = clean (reverse (go (reverse as)
                                                    (reverse bs)))
  where go xs [ ]       = xs
        go [ ] _        = ⊥
        go (x : xs) (y : ys)
           | y > x      = let [r] = sub [One, x] [y]
                                      in r : go xs (inc ys)
           | otherwise = let [r] = sub [x] [y] in r : go xs ys
```

As with *add*2, we reverse the lists to compute the result digit by digit starting with the least significant one using the function *go*. Note that we finally *clean* the list. *clean* removes leading *zero*s, a functionality that would certainly be very useful for real-world organisations too. Before we start the hard work entering *go*, we compare the values of the arguments and If the first one is smaller than the second one, the result is undefined.

In the *go* function, we distinguish two cases: if the first digit of the second argument is greater than that of the first one, we subtract y from $10 + x$ and increase ys by one. Otherwise, we just compute $x - y$.

The function *inc* is a variant of *next* for reversed numbers:

```
inc [ ]           = [One]
inc (Nine : xs) = Zero : inc xs
inc (x : xs)      = next [x] ++ xs
```

Applied on the empty list, *inc* yields *unity*, which is a quite different behaviour than that of *next*. Applied on a list that starts with *Nine*, we insert *Zero* as the head of the *inc*'d tail of the list. Otherwise, we just substitute the head by its *next*.

Somewhat strange might be that we need that *cmp* function – we, apparently, do not need it in other cases. The point is that we have declared that we want to derive the

Digit data type from *Ord*. With this declaration, Haskell automatically imposes the order $Zero < One < Two < \cdots < Nine$. But that would not work for lists of digits. Haskell would assume that a list like $[Nine]$ is greater than $[One, Zero]$, which, as we know, is not the case. We have to tell the compiler explicity, how we want lists of digits to be handled. This is what the *cmp* function does:

```
cmp :: Number → Number → Ordering
cmp x y = case lencmp x y of
                GT → GT
                LT → LT
                EQ → go x y
  where go [] [] = EQ
        go (a : as) (b : bs) | a > b      = GT
                             | a < b      = LT
                             | otherwise = go as bs
        go _ _  = ⊥
```

The function goes through all possible cases, explaining that a longer number is always the greater one and that, in the case they are equally long, one must compare all digits until one is found that is greater or smaller than the digit at the same position in the other list.

Note that we use a special function, *lencmp*, to compare the length of two lists. We do this out of purity on one hand and for efficiency on the other. It would not appear *fair* to use the Prelude function *length*, since it is expressed in terms of a number type that is already much more complete than our humble *Number*s. We could, of course, define our own *length* function, for instance:

```
len :: [a] → Number
len = foldl′ (λn _ → next n) zero
```

But, in fact, we are not too much interested in the concrete length of the two lists, we just want to know, which one, if any, is the longer one. It is not necessary to go through both lists separately in order to learn this, we can just run through both lists at the same time:

```
lencmp :: [a] → [a] → Ordering
lencmp [] []             = EQ
lencmp [] _              = LT
lencmp _ []              = GT
lencmp (_ : xs) (_ : ys) = lencmp xs ys
```

The *lencmp* function, bears a fundamental idea of comparing two sets: by assigning each member of one set to a member of the other until one of the sets is exhausted. The one that is not yet exhausted must be the greater one. Counting could be described in terms of this logic as a comparison of a set with the set of natural numbers. We assign the numbers $1, 2, \dots$ until the first set is exhausted. The last number assigned is the size of the first set. We will learn much more about this apparently simple principle in the future.

As we are already talking about little helpers, it is the right time to introduce some fundamental list functions that we will need to elaborate on the number type later. We will need variants of *take* and *drop*:

```
nTake :: Number → [a] → [a]
nTake [Zero] _   = []
nTake _ []       = []
nTake n (x : xs) = x : nTake (prev n) xs

nDrop :: Number → [a] → [a]
nDrop [Zero] xs  = xs
nDrop _ []       = []
nDrop n (_ : xs) = nDrop (prev n) xs
```

Very useful will be a function that turns all elements of a list into *Zero*s:

```
toZero :: [a] → [Digit]
toZero = map (const Zero)
```

We should also introduce the enumeration function that facilitates list definition, *i.e.* that gives us a list for a range of numbers of the form $[1..9]$ or, in terms of the *Number* type, $[[One]..[Nine]]$:

```
enum :: Number → Number → [Number]
enum l u | l `cmp` u ≡ GT = []
         | otherwise      = go l u
  where go a b | a `cmp` b ≡ EQ = [a]
               | otherwise      = a : go (next a) b
```

Finally, we also need the function *clean*, which is defined as:

```
clean :: Number → Number
clean [Zero]      = [Zero]
clean (Zero : ds) = clean ds
clean ds          = ds
```

We, hence, leave the number $[Zero]$ untouched. If the number starts with the digit *Zero*, but has more than just that one number, we ignore this leading *Zero* and continue with the remainder of the list. (Note that, since the case of a list that consists of only the digit *Zero* is already handled in the first case, *ds* in the second case will never be the empty list!) Finally, a list that does not start with *Zero* is just given back as it is.

Now, let us turn to the model for our number type, that is how we interpret a list of digits. There are many ways to interpret numbers. A somewhat natural way is to indicate a function that, for any list of *Digit*s, gives us the numerical value of the number we intend to represent with this list. The, perhaps, most obvious way to do so is to convert the list of *Digit*s into a string and then to read this string in again as integer. We would define a conversion function of the form

```
toString :: Number → String
toString = map toChar
  where toChar Zero = '0'
        toChar One  = '1'
```

and so on. But this approach is not very interesting. It does not give us any insight. What we would like to have instead is a model that explains how numeral systems work in general. The key to understand how such a model can be devised is to see that our system consists of 10 symbols. With one of these symbols, we, hence, can represent 10 different numbers. With two of these symbols, we represent 10×10 numbers, that is the numbers $0 \dots 9$ plus the numbers $10 \dots 19$ plus the numbers $20 \dots 29$ plus ... the numbers $90 \dots 99$. With three of these symbols, we then can represent $10 \times 10 \times 10$ numbers, namely the numbers $0 \dots 999$ and so on. In other words, the *weight* of a digit in a number represented in a numeral system with b symbols corresponds to a power of b, which we therefore call the *base* of that numeral system. A numeral system with 2 symbols would have the base 2, the weight of each digit would therefore be a power of 2. A numeral system with 16 symbols has the base 16 and the weight of each digit would be a power of 16.

The exponent, that is to which number we raise the base, is exactly the position of the digit in a number if we start to count positions with 0. The number 1763 has the value: $1 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 3 \times 10^0 = 1000 + 700 + 60 + 3$:

3	2	1	0
1	7	6	3

We could devise a data type that represents those *weighted* digits nicely as:

```
type WDigit  = (Number, Digit)
type WNumber = [WDigit]
```

The *WDigit* is a tuple of *Number* and *Digit*, where the number is the exponent to which we have to raise the base. We can convert a *Number* easily to a *WNumber* by:

```
weigh :: Number → WNumber
weigh = go [Zero] ∘ reverse
  where go _ []       = []
        go n (d : ds) = (n, d) : go (next n) ds
```

The *weigh* function reverses the input in order to start with the least significant digit and then just passes through this list adding the exponent incrementing it by one in each step. Note that the order of a *WNumber* does not matter anymore, because the decisive information that is encoded in the position of each digit is now made explicit in the exponent.

The inverse of this function is:

```
unweigh :: WNumber → Number
unweigh = reverse ∘ map snd ∘ complete [Zero] ∘ sortW
  where complete _ [] = []
        complete n ((e, d) : xs)
          | n ‘cmp‘ e ≡ LT = (n, Zero) : complete (next n) ((e, d) : xs)
          | otherwise      = (e, d)    : complete (next n) xs
sortW :: WNumber → WNumber
sortW = sortBy (λx y → fst x ‘cmp‘ fst y)
```

We, first, sort the components of the *WNumber* in ascending order according to their exponents. We, then, *complete* the *WNumber*, *i.e.* we fill in *Zero*s for missing exponents such that the resuling *WNumber* has a component for every exponent from 0 to the greatest one present. From this list, we extract the digits and reverse the result.

To build the model, we still need a function that converts digits into one-digit integers. This is straight forward:

```
digit2Int :: Digit → Int
digit2Int Zero  = 0
digit2Int One   = 1
digit2Int Two   = 2
digit2Int Three = 3
digit2Int Four  = 4
digit2Int Five  = 5
digit2Int Six   = 6
digit2Int Seven = 7
digit2Int Eight = 8
digit2Int Nine  = 9
```

To convert a *Number* to an *Integer*, we first convert the *Number* to a *WNumber* and then convert the *WNumber* to an *Integer*:

```
n2Integer :: Number → Integer
n2Integer []  = 0
n2Integer [n] = fromIntegral (digit2Int n)
n2Integer ns  = w2Integer    (weigh ns)
```

As a convention, we convert the empty list into 0. (We could raise an error for this case, but that does not appear to be necessary or even useful.) A one-digit number is simply converted by converting its single digit. Since *digit2Int* converts a digit to an *Int*, but we now want an *Integer*, we still have to call *fromIntegral* on the result, to convert from *Int* to *Integer*.

Numbers with many digits are converted to *WNumber* using *weigh* and then converted to *Integer* using *w2Integer*:

```
w2Integer :: WNumber → Integer
w2Integer          = sum ∘ map conv
  where conv w = let x = n2Integer (fst w)
                     d = fromIntegral (digit2Int (snd w))
                 in  d ∗ 10 ↑ x
```

Weighted numbers are converted to *Integer* by summing up the single values of the digits, which are calculated in terms of powers of 10: $d \times 10^x$, where d is the digit converted to *Int* by *digit2Int* and then converted to *Integer* by *fromIntegral*. x is the exponent converted to *Integer* using *n2Integer*.

This looks like an infinite regress where we convert the exponent of the weighted number, which is a *Number*, to an Integer using *n2Integer*, which then calls *w2Integer*, which again calls *n2Integer* to convert the exponent, which, again, calls *w2Integer* and so on.

It is indeed very well possible that we have extremely large numbers with exponents that are many digits long, but even the greatest number will finally converge to an exponent that is smaller then 10. The incredibly large number $10^{100000000000}$, for example, has an exponent with 12 digits, which, represented as a *Number*, is

[*One*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*].

The greatest exponent in this number, the exponent of the leading *One*, however, has just two digits: [*One*, *One*], which, in the next conversion step, reduces to [*One*] for the most significant digit and, hence, will be converted immediately to 1.

Of course, we do not need the data type *WNumber* for this conversion. We could very well have converted a number by reverting it and then pass through it with an inreasing *Integer* exponent starting from 0. The detour through weighted numbers, however, is a nice illustration of the model for our number system, and, perhaps, there will be use for this or a similar data type later during our journey.

1.4 Multiplication

As addition can be seen as a repeated application of counting up, multiplication can be seen as repeated addtion. A naïve implementation of multiplication, hence, is:

```
mul :: Number → Number → Number
mul _ [Zero] = [Zero]
mul [Zero] _ = [Zero]
mul a [One]  = a
mul [One] b  = b
mul a b      = a ‘add‘ (a ‘mul‘ (prev b))
```

Notable, here, is that we have more base cases than with addition: Any number multiplied by *zero* is just *zero* and any number multiplied by *unity* is just that number. From here on, we add a to a and count b down simultaneously, until b reaches the base case [*One*].

This simple implementation is not optimal in terms of computation complexity: we need b steps to multiply a and b. For a multiplication of two numbers in the range of millions, we need millions of single additions. As with addition, we can improve on this by multiplying digit by digit or, more precisely, by multiplying all digits of the first number by all digits of the second number. This, however, is somewhat more complicated than in the case of addition, because multiplication has effect on the weight of the digits. On two one-digit numbers, weight has no impact, since the weight of each digit is just 0: $(2 \times 10^0) \times (3 \times 10^0) = 6 \times 10^{0+0} = 6 \times 1 = 6$. But, if 2 and 3 above were digits of a number with more than one digit and, themselves not the least significant digits, *e.g.* 20×30, then we would have something of the form: $(2 \times 10^1) \times (3 \times 10^1) = 6 \times 10^2 = 600$.

We, therefore, have to take the weight of the digits into account. But what is the best way to do so? We could of course use the weighted number type we already defined in the previous section. The disadvantage of this approach, however, is that we have to perform additional arithmetic on the weight, potentially searching for equal weights in the resulting number or reordering it to bring equal weights together. We can avoid this overhead by reflecting the weight in numbers, simply by appending $n - 1 + m - 1$ *Zero*s to the result of multiplying the n^{th} digit of one number with the m^{th} digit of the other one and, eventually, adding up all these components.

We first implement a function that multiplies a digit with all digits of a number appending *Zero*s to each result and adding them up:

```
mul1 :: Digit → Number → [Digit] → Number
mul1 _ []            _  = zero
mul1 x (y: _)        [] = [x] ‘mul‘ [y]
mul1 x (Zero : ys) zs = mul1 x ys (tail zs)
mul1 x (y : ys)      zs = add2 ([x] ‘mul‘ [y] ++ zs)
                              (mul1 x ys (tail zs))
```

If the number is exhausted, *i.e.* if we have already multiplied all digits, the result is just *zero*. If the *Zero*s have been exhausted, whatever remains from the number, we multiply x with the head of that rest. We ignore *Zero*s in the number, but make sure to consider their weight by reducing the trail of *Zero*s by one in the continuation. In all other cases, we multiply x and the first digit in the number using the simple *mul* and appending the *Zero*s to the result. This result is then added to the result of the recursion of *mul1* with the tail of the number and the tail of the *Zero*s.

This function is now *mapped* on *Number*:

```
mulN :: Number → Number → [Digit] → Number
mulN []           _ _  = zero
mulN (x: _)       b [] = mul1 x b []
mulN (Zero : xs)  b zs = mulN xs b (tail zs)
mulN (x : xs)     b zs = add2 (mul1 x b zs)
                              (mulN xs b (tail zs))
```

If the first number is exhausted, we just return *zero*. If the *Zero*s are exhausted, we apply *mul1*, *i.e.* we multiply one digit with b, and terminate. Note that the *Zero*s should be exhausted only if there is just one digit left in the numbers. Again, we ignore *Zero*, but respect its weight. In all other cases, we apply *mul1* on the first digit of the first number and add the result with the recursion on the tail of the first number and the tail of *Zero*s.

Finally, we apply *mulN* on two numbers creating the trail of *Zero*s:

```
mul2 :: Number → Number → Number
mul2 [] _ = zero
mul2 _ [] = zero
mul2 a b  = mulN a b ((toZero $ tail a) ++
                      (toZero $ tail b))
```

We handle the cases where one of the numbers is the empty list explicitly to avoid problems with the call of *tail* later on. We then call *mulN* for a and b and the trail of *Zero*s that results from converting all digits but one in a and b to *Zero*.

Note that this is exactly what we do, when we elaborate a multiplication with pen and paper. If we multiplied, say, 13×14, we would write the partial results aligned according to the number of zeros they would have:

13×14
$1 \times 1 = 100$
$1 \times 4 = 040$
$3 \times 1 = 030$
$3 \times 4 = 012$

Now we add up the partial results:

$100 + 040 = 140$
$030 + 012 = 042$
$140 + 042 = 182$

The grouping of additions chosen here corresponds to the additions performed in *mul1* and *mulN*: The first two additions are performed in *mul1*, the last line is done in *mulN*.

Let us look at how *mulN* works for $[One, Three] \times [One, Four]$. We start with

mulN (*One* : [*Three*]) [*One*, *Four*] [*Zero*, *Zero*] =
add2 (*mul1 One* [*One*, *Four*] [*Zero*, *Zero*]) (*mulN* [*Three*] [*One*, *Four*] [*Zero*]).

The first term of $\mathit{add2}$ is

$\mathit{mul1}\ \mathit{One}\ (\mathit{One} : [\mathit{Four}])\ [\mathit{Zero}, \mathit{Zero}] =$
$\mathit{add2}\ (([\mathit{One}]\ `\mathit{mul}`\ [\mathit{One}]) +\!\!+ [\mathit{Zero}, \mathit{Zero}])\ (\mathit{mul1}\ \mathit{One}\ [\mathit{Four}]\ [\mathit{Zero}]).$

The first term, here, reduces to

$[\mathit{One}] +\!\!+ [\mathit{Zero}, \mathit{Zero}] = [\mathit{One}, \mathit{Zero}, \mathit{Zero}],$

which corresponds to **100** in the paper example above. The second term reduces to

$\mathit{mul1}\ \mathit{One}\ (\mathit{Four} : [\,])\ [\mathit{Zero}] =$
$\mathit{add2}\ (([\mathit{One}]\ \mathit{mul}\ [\mathit{Four}]) +\!\!+ [\mathit{Zero}])\ (\mathit{mul1}\ [\mathit{One}]\ [\,]\ [\,]),$

which, in its turn, is just

$\mathit{add2}\ [\mathit{Four}, \mathit{Zero}]\ [\mathit{Zero}] = [\mathit{Four}, \mathit{Zero}]$

and corresponds to **40** in the manual calculation. We, hence, have

$\mathit{add2}\ [\mathit{One}, \mathit{Zero}, \mathit{Zero}]\ [\mathit{Four}, \mathit{Zero}] = [\mathit{One}, \mathit{Four}, \mathit{Zero}]$

at the end of the first round of $\mathit{mul1}$. This is the same result as we obtained in the first addition step in the manual process, *i.e.* **140**. Returning to the first equation, we now have:

$\mathit{mulN}\ (\mathit{One} : [\mathit{Three}])\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}, \mathit{Zero}] =$
$\mathit{add2}\ ([\mathit{One}, \mathit{Four}, \mathit{Zero}])\ (\mathit{mulN}\ [\mathit{Three}]\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}]).$

The second term of $\mathit{add2}$, here, produces:

$\mathit{mulN}\ (\mathit{Three} : [\,])\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}] =$
$\mathit{add2}\ (\mathit{mul1}\ \mathit{Three}\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}])\ (\mathit{mulN}\ [\,]\ [\mathit{One}, \mathit{Four}]\ [\,]).$

The first term, the call to $\mathit{mul1}$, is:

$\mathit{mul1}\ \mathit{Three}\ (\mathit{One} : [\mathit{Four}])\ [\mathit{Zero}] =$
$\mathit{add2}\ (([\mathit{Three}]\ `\mathit{mul}`\ [\mathit{One}]) +\!\!+ [\mathit{Zero}])\ (\mathit{mul1}\ \mathit{Three}\ (\mathit{Four} : [\,])\ [\,]).$

The first term of this addition is $[\mathit{Three}, \mathit{Zero}]$, which corresponds to the same result **30** we had above in the third step of the manual multiplication, and the second term is:

$\mathit{mul1}\ [\mathit{Three}]\ (\mathit{Four} : [\,])\ [\,] = [\mathit{Three}]\ `\mathit{mul}`\ [\mathit{Four}] = [\mathit{One}, \mathit{Two}].$

The result **12** we obtained before. Going back, we now have:

$\mathit{mul1}\ [\mathit{Three}]\ (\mathit{One} : [\mathit{Four}])\ [\mathit{Zero}] =$
$\mathit{add2}\ [\mathit{Three}, \mathit{Zero}]\ [\mathit{One}, \mathit{Two}] = [\mathit{Four}, \mathit{Two}]$

We now have the result of the second addition step in the paper multiplication, *i.e.* **42**. and, returning to the first equation, we get the final result:

$\mathit{mulN}\ (\mathit{One} : [\mathit{Three}])\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}, \mathit{Zero}] =$
$\mathit{add2}\ [\mathit{One}, \mathit{Four}, \mathit{Zero}]\ [\mathit{Four}, \mathit{Two}] = [\mathit{One}, \mathit{Eight}, \mathit{Two}].$

This is the last line of the addition: $140 + 42 = \mathbf{182}$.

How many steps do we need for multiplication with this approach? We, first, multiply all digits of one number with all digits of the other number and, thus, perform $n \times m$ one-digit multiplications, where n and m are the numbers of digits of the first and the second argument respectively. We then add all the $n \times m$ numbers together, resulting in $n \times m - 1$ multi-digit additions. Most of the multi-digit additions, though, add *Zero*s, which is just one comparison and, hence, quite unexpensive. We have, however, many of those simple steps, because we add numbers of the size $n + m - 1, n + m - 2, \dots, 1$. This is the addition of all numbers from 1 to $n + m$, a type of problems, we will study in the next chapter.

Anyhow, the cost for *mul2* grows only in the size of the arguments, whereas the naïve *mul* grows directly in the value of the second number. The number of steps is $nmp + (nm - 1)a$, where p is the cost for a single-digit multiplication and a that of an addition. For very, very large numbers, say, numbers with thousands or millions of digits, the approach, still, is too slow. There are many ways to multiply more efficiently, but that is not our focus here.

Multiplication appears to be such a tiny simple device, but it introduces huge complexity. If we just look at the patterns that *mul2* produces when processing two numbers $[a, b]$ and $[c, d]$: $[ac + ad + bc + bd]$, we see that multiplication is intimately involved with problems of combinatorics, which too will be a major topic of the next chapter. Imagine the multiplication of a number with itself, *i.e.* where c and d equal a and b, respectively:

$$[a, b] \times [a, b] = [aa + ab + ba + bb] = [a^2 + 2ab + b^2]. \tag{1.1}$$

Indeed, multiplying $[One, Two]$ with itself results in $[One, Four, Four]$ and $[One, Three]$ in $[One, Six, Nine]$. This pattern plays a role in many branches of mathematics, like algebra, combinatorics and probability theory, and is truly one of the most important facts you can learn about mathematics. Should equation 1.1 not be familiar to you already, you definitely should memorise it. The tiny device of multiplication, one could contemplate, is a focal point of many complications we will encounter on our journey – and this appears to me as one of the characteristics of mathematics: that small problems, such as multiplication, thought through, develop unforeseen impact on apparently completely different subjects.

A nice illustration of the patterns created by multiplication is the results of squaring numbers that consist only of 1s. Have a look at the following pyramid:

$$1 \times 1 = 1$$
$$11 \times 11 = 121$$
$$111 \times 111 = 12321$$
$$1111 \times 1111 = 1234321$$
$$11111 \times 11111 = 123454321$$

It is as if the digit in the centre of the number on the right-hand side of the equations wanted to tell us the size of the factors used to create it. When we try to fool the numbers, leaving some 1s out in one of the factors, they realise it immediately, and come up with "damaged" results like

$$1 \times 11 = 11$$
$$11 \times 111 = 1221$$
$$11 \times 1111 = 12221$$
$$11111 \times 1111111111 = 12345555554321.$$

Now, the central digits in the result tell us the size of the smaller number and their repetition tells us the difference to the greater factor, which is exactly one less than the number of repetitions.

1.5 Division and the Greatest Common Divisor

It is now time to introduce Euclid. Unusually little is known about this author. Important scholars of the time (about 300 BC) are usually mentioned by name in philosophical texts of other authors and often with some biographical detail. In the case of Euclid, this is different. Euclid is rarely mentioned by name – and when it happens, he is confused with an earlier philosopher of the same name – and nothing is told about his life but the fact that he was active in Alexandria for some time. This is particularly strange, since Euclid's work had a tremendous influence on the antiquity and on through the middle ages up to our days. This has led to the conjecture that Euclid was not a person, but a group of scholars at the university or library of Alexandria. This idea may be inspired by similar conjectures concerning the "person" of Homer or by the existence of groups named after fictional characters in later times like, in the 20^{th} century, the "Association of collaborators of Nicolas Bourbaki", a highly influential group of mathematicians dedicated to the formalisation of mathematics. Nicolas Bourbaki, even though he had an office at the École Normale Supérieure for some time, did not exist. He is a fictional character whose name was used for the publications of the Bourbaki collective.

Euclid – who or whatever he was – is the author of the *Elements*, the mother of all axiomatic systems and, certainly, one of the greatest intellectual achievements of the antiquity. The *Elements* lay out the acient knowledge on geometry, arithmetic and number theory in fifteen books following a rigid plan starting with axioms, called "postulates", followed by theorems and their proofs based only on the axioms. There are some inaccuracies in the choice of the axioms and not all proofs are rock-solid according to modern standards. But, anyway, the rigidity of the Elements was not achieved again before the 19^{th} century, perhaps with the *Disquisitiones Arithmeticae* by 21-year-old Carl Friedrich Gauss.

Here, we are interested mainly in some of the content of book 7, which deals with issues of arithmetic and elementary number theory, in particular division and the greatest common divisor. According to Euclid, division solves equations of the form

$$a \; div \; b = q + r, \tag{1.2}$$

and fulfils the constraint

$$a = qb + r, 0 \leq r < b. \tag{1.3}$$

There is a kind of mismatch between this notion of division, usually called *division with remainder*, and multiplication in that multiplication of any two natural numbers results in a natural number, whereas division with remainder results in two numbers, the *quotient* q and the *remainder* r. The division of two numbers that are *divisible*, *i.e.* the division leaves no remainder, is just a special case of this operation like in $9 \; div \; 3 = 3+0$. In other cases, this does not work: $8 \; div \; 3 = 2+2$, since $2 \times 3 + 2 = 8$. We already have seen such a mismatch with addition and subtraction: the addition of any two natural numbers always produces a natural number; subtraction, however, does only produce a natural number when its second term is less than or, at most, equal to the first term. This will be an important topic in the progress of our investigations.

Euclid's algorithm to solve the equation goes as follows: Division by zero is not defined. Division of zero by another number (not zero) is zero. Otherwise, starting with the quotient $q = 0$ and the remainder $r = a$, if the remainder r is less than the divisor b, then the result is (q, r). Otherwise, we decrement the remainder by b and increment q by one:

```
quotRem :: Number → Number → (Number, Number)
quotRem _ [Zero] = error "division by zero"
quotRem [Zero] _ = (zero, zero)
quotRem a [One]  = (a, zero)
quotRem a b      = go a zero
  where go r q | r `cmp` b ≡ LT = (q, r)
               | otherwise      = go (r `sub` b) (next q)
```

As you should realise at once, this algorithm is not efficient for large numbers a. If a is much larger than b, we will have to subtract lots of bs from it. In fact, the complexity of this algorithm is $\lfloor a/b \rfloor$, since we need $\lfloor a/b \rfloor$ steps to bring a down to an r that is smaller than b. The complexity of the algorithm, hence, equals (a part of) its result!

As usual, we can improve by taking the structure of the numbers into account, namely by operating on digits instead of whole numbers. Have a look at the following, admittedly, scary-looking listing:

```
quotRem2 :: Number → Number → (Number, Number)
quotRem2 _ [Zero] = error "division by zero"
quotRem2 [Zero] _ = (zero, zero)
quotRem2 a [One] = (a, zero)
quotRem2 a b      = go zero [] a
  where go q [] [] = (clean q, zero)
        go q c []  = (clean q, clean c)
        go q c r   = let x = clean (c ++ [head r])
                         y = tail r
                     in if x `cmp` b ≡ LT
                        then go (q ++ zero) x y
                        else let (q', r') = quotRem x b
                                 r2 | r' ≡ zero = []
                                    | otherwise = r'
                             in go (q ++ q') r2 y
```

We start, as usual, with the base cases: division by zero and not defined; zero divided by something else is zero. A number divided by one is just that number.

For all other cases, we call *go* with *zero* as quotient and a as remainder. There is an additional parameter, c, which takes care of carries. If we have exhausted, both the carries and the remainder, then the result is just $(q, zero)$, *i.e.* we have no remainder. If the remainder is exhausted, but not the carries, the carries together are the remainder. Otherwise, we proceed as follows: We take the head of of the remainder and concatenate it to previous carries starting with the empty list. If this number is less than b, we append a *Zero* to q and continue with x as carry and the *tail* of r. Note that, if this happens on the first digit, the *Zero*s appended to q will be cleaned off later. Only *Zero*s between digits are taken into account. This is exactly what we do, when we divide with pencil and paper: when, during the process, the next number in a cannot be divided by b, we append a zero to the partial result obtained so far and append the next number of a to the remainder of the previous calculation.

Otherwise, if x is not less than b, we divide these two numbers using the naïve *quotRem*. The quotient resulting from the application of *quotRem* is appended to the previous result q. The remainder, if not zero, is carried over. Since *quotRem* is applied, as soon as we arrive at a number that is equal to or greater than b appending one digit of a after the other, this number is at most 9 times as big as b. In other words, *quotRem* in this context, will never need more than 9 steps. Nevertheless, *quotRem* is the bottleneck of this implementation. With lookup tables for one-digit divisions, we could reach a significant speed-up. But optimising, again, is not our prime concern here. Therefore, we will stick with this suboptimal solution.

An important aspect of the algorithm is that we chop off leading *Zero*s, whenever we go to use a sequence of digits as a number, in particular before we return the result and before calling *quotRem*. The algorithm handles numbers as sequence of digits that are as such meaningless. But whenever it operates on those sequences it takes care of handling them as proper numbers.

Let us look at a simple example, say, $[One, Two, Three]$ divided by $[Six]$. We start with

go [*Zero*] [] [*One*, *Two*, *Three*]

and compute x as *clean* ([] ++ [*One*]) and y as [*Two*, *Three*]. Since x, which is [*One*], is less than b, [*Six*], we continue with

go ([*Zero*] ++ [*Zero*]) [*One*] [*Two*, *Three*].

This time x is *clean* ([*One*] ++ [*Two*]) and y is [*Three*]. x now is greater than b and therefore we compute

(q', r') = *quotRem* [*One*, *Two*] [*Six*]

where q' is [*Two*] and r' is [*Zero*]. We then continue with

go ([*Zero*, *Zero*] ++ [*Two*]) [] [*Three*]

and compute x as [*Three*] and y as []. Since x, again, is less than b, we continue with

go ([*Zero*, *Zero*, *Two*] ++ [*Zero*]) [*Three*] [],

which is the second base case of *go* leading to

(*clean* [*Zero*, *Zero*, *Two*, *Zero*], *clean* [*Three*]),

which in its turn is just ([*Two*, *Zero*], [*Three*]) expressing the equation $6 \times 20 + 3 = 123$.

There are many interesting things to say about division and especially about the concept of the remainder. First, the remainder is an indicator for *divisibility.* A number b is said to divide a number a or a is divisible by b, $b \mid a$, if a *div* $b = (q, 0)$, *i.e.* if the remainder of the Euclidian division is 0. In Haskell, we can define the remainder as:

```
rem :: Number → Number → Number
rem a b = snd (quotRem2 a b)
```

The quotient, correspondingly, is

```
div :: Number → Number → Number
div a b = fst (quotRem2 a b)
```

Divisibility, then, is:

```
divides :: Number → Number → Bool
divides a b | rem b a ≡ zero = True
            | otherwise      = False
```

There are some rules (valid for natural numbers) that can be defined on divisibility, namely: For all numbers a: $1 \mid a$, that is: 1 divides all numbers, since a *div* $1 = (a, 0)$.

It holds also that $a \mid b \wedge b \mid c \rightarrow a \mid c$. In other words: if a divides b and b divides c, then a also divides c. (The symbol "$\wedge$" means "AND" here.) This is because, if b divides c, then c is a multiple of b and, if a divides b, then b is a multiple of a and, in consequence, c is also a multiple of a. Any number divisible by 4, for instance, is also divisible by 2, since $2 \mid 4$.

Furthermore, if $a \mid b$ and $b \mid a$, then we can say that $a = b$, since, if a were greater than b, then a would not divide b and vice versa.

An interesting – and important – equality is also $a \mid b \wedge a \mid c \rightarrow a \mid (b + c)$. This rule says that the sum of any two numbers b and c, both divisible by another number a is also divisible by a. For the special case $a = 2$, this rule says that the sum of two even numbers is also even: $4 + 6 = 10$, $50 + 28 = 78$, $1024 + 512 = 1536$, ... This is true in general for all numbers a, *e.g.* 5: $10 + 15 = 25$, which is $2 \times 5 + 3 \times 5 = 5 \times 5$, or $35 + 625 = 660$, which is $7 \times 5 + 125 \times 5 = 132 \times 5$. We can go even further and say $a \times b + a \times c = a \times (b + c)$. This is called the distributive law and we have already used it implicitly when defining multiplication. We will come back to it very soon.

The remainder gives rise to an especially interesting concept, the concept of arithmetic *modulo n*. The term modulo refers just to the remainder of the Euclidian division. Most implementations in programming languages, including Haskell, distinguish the operator *mod* and *rem* according to the *signedness* of dividend and divisor. For the moment, that is not relevant for us, since we are working with natural numbers only, so, for the moment, we will treat *mod* and *rem* as being the same concept.

The most common example of modulo arithmetic is time measured with a 12 or 24 hours clock. At midnight, one can say it is 12 o'clock; since $12 \bmod 12 = 0$, we can also say, it is 0 o'clock. With the 24 hours clock, one hour after noon is 13:00 o'clock. $13 \bmod 12 = 1$, 13, thus, is just 1 in the 12 hours clock. This principle works for arbitrary large numbers, *e.g.* 36 is 12, since $36 \bmod 12 = 0$ and, since $36 \bmod 24 = 12$, we can say it is noon. 500 is 8 in the evening, since $500 \bmod 24 = 20$ and $20 \bmod 12 = 8$. With modular arithmetic, arbitrary large numbers modulo n are always numbers from 0 to $n - 1$ and any operation performed on numbers modulo n results in a number between 0 and $n - 1$. This apparently trivial fact is of huge importance. We will come back to it over and over again.

Especially interesting for programmers is arithmetic modulo 2, because any operation has either 0 or 1 as result, *i.e.* the vocabulary of binary number representation. Indeed, addition of the numbers 0 and 1 modulo 2 is just the *exclusive or* (XOR) operation: $0 + 0 = 0 \mod 2$, $1 + 0 = 1 \mod 2$, $1 + 1 = 0 \mod 2$, since $1 + 1 = 2$ and $2 \bmod 2 = 0$.

The XOR operation gives the same results: $0 \oplus 0 = 0$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$. Multiplication modulo 2 is equivalent to AND: $0 \times 0 = 0 \mod 2$, $0 \times 1 = 0 \mod 2$, $1 \times 1 = 1 \mod 2$. The truth values of the formula $p \wedge q$ are shown in the table below:

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

One of the fundamental tools developed in the Elements is *gcd*, the *greatest common divisor*. As the name suggests, the *gcd* of two numbers a and b is the greatest number that divides both, a and b.

The algorithm given in the Elements is called *Euclidian algorithm* and is used with a small, but important variation until today. The original algorithm goes as follows: the gcd of any number a and 0 is a; the gcd of any number a with any number b is $\gcd(b, a - b)$, where $0 < b \leq a$. If $b > a$, we just turn the arguments around: $\gcd(b, a)$.

For large numbers, this is not efficient, especially, if a is much greater than b. The remarks on modulo above, however, hint strongly at a possible optimisation: the use of the remainder operation instead of difference:

```
gcd :: Number → Number → Number
gcd a [Zero] = a
gcd a b      = gcd b (a `rem` b)
```

Let us look at some examples:

gcd [*Nine*] [*Six*] = *gcd* [*Six*] ([*Nine*] ‘*rem*‘ [*Six*]),

which is *gcd* [*Six*] [*Three*], which, in its turn, is

gcd [*Three*] ([*Six*] ‘*rem*‘ [*Three*] = [*Zero*])

and, hence

gcd [*Three*] [*Zero*] = [*Three*].

More complicated is the *gcd* of [One,One] and [Six]:

gcd [*One*, *One*] [*Six*] = *gcd* [*Six*] ([*One*, *One*] ‘*rem*‘ [*Six*]),

which is

gcd [*Six*] [*Five*] = *gcd* [*Five*] ([*Six*] ‘*rem*‘ [*Five*]),

which is

gcd [*Five*] [*One*] = *gcd* [*One*] ([*Five*] ‘*rem*‘ [*One*]), which leads to

$gcd\ [One]\ [Zero] = [One]$.

It is noteworthy that the algorithm always terminates. This is true because, since *rem* always reduces b to a value between *zero* and $a-1$ and, with a getting smaller and smaller, we must at some point reach either *unity* (when b does not divide a) or *zero* (when b does divide a). If we reach *zero*, we have a result; otherwise, we will reach *zero* in the next step, because *unity*, as we have already discussed, divides any number.

Furthermore, if a is the smaller number, *gcd* will just flip the arguments, *e.g.* $gcd\ 10\ 100 = gcd\ 100\ (10\ `rem`\ 100)$ and, since $10\ div\ 100 = (0, 10)$, this corresponds to $gcd\ 100\ 10$.

We will analyse the running time of *gcd* later in chapter 3. For now, it may suffice that each step reduces the problem to $a \bmod b$, which is in the range of $0 \dots b-1$, while, with the original algorithm, the problem is reduced only to $a - b$ per step. With large numbers and, in particular, with a huge difference between a and b, this reduction is quite small. With the reduction by $a \bmod b$, the difference between the numbers and even the size of a do not matter. That is an effect of modular arithmetic.

An important insight related to the *gcd*, is *Euclid's lemma*, which states that if a divides cb, then a must share common factors with c or b. This is easy to see, since, that a divides cb means that there is a number n, such that $na = cb$. This number is $n = cb/a$. If a and cb did not share common factors, then cb/a would not be a natural number. For example 10 and 7 do not share factors with 3; there is thus no natural number n, such that $3n = 7 \times 10$. With 6 instead of 7, however, there is a common factor, namely 3 itself. Therefore, we can solve $3n = 6 \times 10 = 60$, simply by dividing 3 on both sides of the equation: $n = 60/3 = 20$.

Finally, we should mention a cousin of *gcd*, the *least common multiple*, *lcm*, the smallest number that is a multiple of two numbers, a and b. The obvious multiple of two numbers is the product of these numbers $a \times b$. But there may be a smaller number c, such that $a \mid c \wedge b \mid c$. How can we find that number? Well, if a and b have a gcd that is not 1, then any number divisible by a and divisible by b is also divisible by $\gcd(a, b)$. The product of a and b, hence, is divisible by $\gcd(a, b)$ and, since the *gcd* is the common divisor that reduces the product $a \times b$ most, that quotient must be the least common multiple, *i.e.*

$$lcm(a, b) = \frac{a \times b}{\gcd(a, b)}. \tag{1.4}$$

1.6 Powers, Roots and Logarithms

Multiplication can be seen as a kind of higher-order addition: one of the factors tells us how often we want to add the second factor to itself: $a \times b = b + b \dots$ This relation can be expressed nicely with the summation notation $\sum$:

$$a \times b = \sum_{i=1}^{a} b$$

For instance, 2×3 is $\sum_{i=1}^{2} 3 = 3 + 3 = 6$ and 1×3 would just be $\sum_{i=1}^{1} 3 = 3$. For $a = 0$, summation is defined as 0.

In Haskell, for any a of type class *Num*, this is implemented as $sum :: [a] \rightarrow a$, which takes an argument of type $[a]$ and returns the sum of all elements in the input list. For our number type (which we have not yet defined as *Num*), this could be:

```
summation :: [Number] → Number
summation = foldr add2 zero
```

From this definition of multiplication as repeated addition, we can go further. We can introduce an operation that repeats multiplication of a number with itself. This operation is called *power*: $a^b = a \times a \times \ldots$ and can be captured with the product notation:

$$a^b = \prod_{i=1}^{b} a$$

a^2, for instance, is $\prod_{i=1}^{2} a = a \times a$. For $b = 0$, the product is defined as 1.

In Haskell, the product for any a of type class *Num* is implemented as $product :: [a] \rightarrow a$. For our number type, we could define:

```
nProduct :: [Number] → Number
nProduct = foldr mul2 unity
```

We can define *power* as:

```
power :: Number → Number → Number
power _ [Zero] = unity
power a [One] = a
power a b      = a `mul2` power a (prev b)
```

This algorithm, of course, is not efficient, since it needs b steps to calculate the b^{th} power of any number. A common trick to accelerate the algorithm is *exponentiation by squaring* where we reduce b faster than by just decrementing it by one. Indeed, when we exponentiate a number with an even number b, the result is $a^{2^{\frac{b}{2}}}$. What about odd bs? In this case, we reduce b by one, then we have an even number in the exponent, and multiply a once more: $a \times a^{2^{\frac{b}{2}}}$. With this algorithm, we need, instead of b steps, a logarithmic amount of steps (log base 2), which we will discuss in a second, plus one extra multiplication, when b is odd. In Haskell, this variant of power could be implemented as follows:

```
power2 :: Number → Number → Number
power2 _ [Zero] = unity
power2 a [One]  = a
power2 [One] _  = unity
power2 a b      = case b `quotRem2` two of
                    (q, [Zero]) → power2 (a `mul2` a) q
                    (q, _)      → a `mul2`
                                    power2 (a `mul2` a) q
```

From *power*, we can go on further, introducing an operator that operates on powers, and, indeed, there is Knuth's *up-arrow* notation: $a \uparrow\uparrow b = a^{b^{b^{\cdots}}}$. When we have defined this, we can go on by introducing even more arrows: $a \uparrow\uparrow\uparrow b = a(\uparrow\uparrow (b \uparrow\uparrow (b \dots)))$ and we can go on and on *ad infinitum*.

This approach gives us a lot of power to define huge numbers. But what about going backward? How can we invert the effect of power (not to mention Knuth's megapower)? There are in fact two ways to invert the power function. We may ask for the *root* a in $a^b = c$, if we know b and c, and we may ask for the *exponent* b, if a and c are known. The first operation is just called the *root*, whereas the latter is called the *logarithm* of c to base a.

Both these functions are again asymmetric in that any power of two natural numbers a^b results in a natural number. Not all natural numbers c, however, have a natural numbered root a or a natural numbered logarithm b to base a. For natural numbers, we should therefore define these algorithms with a remainder, as we did for division.

A word of caution: The algorithms to follow are not canonical like multiplication or division with remainder. You will not find them in many textbooks on arithmetic. We introduce them here because they are considerably different from the algorithms discussed so far. Most of those algorithms perform a computation and produce their result in one step of that computation (even if the computation itself may be composed of several steps). The algorithm we discuss here is by contrast a searching algorithm. We have to pick numbers and check whether they produce the expected result when they are applied to *power*. In terms of computation complexity, this approach is much more costly than just performing a simple computation.

The most simplistic way for such a search would just count down from c (or up from *unity*) until we find a number that b times multiplied with itself is c:

```
searchRoot :: Number → Number → (Number, Number)
searchRoot _ [Zero] = ⊥
searchRoot [Zero] _ = (zero, zero)
searchRoot c b      = go c
  where go a = let x = power2 a b in case x `cmp` c of
                                     EQ -> (a, zero)
                                     LT -> (a, c `sub2` x)
                                     GT -> go (prev a)
```

We first state that the *zero*th root of any number is undefined. In fact, any number to the zeroth power is one. So, strictly speaking, the zeroth root of any number but one is undefined and the zeroth root of one is all numbers. Since we cannot express *all numbers* in a meaningful way, we just rule this case out.

Any root of zero (but the zeroth root, which we have already considered in the first line) is again zero: zero is the only number that multiplied to itself is zero. For all other cases, we loop from c downwards to find a number a, such that $a^b \leq c$. If we find a number, whose power equals c, the result is just that number, otherwise, if the number is smaller, the result is that number and the difference of c and its power. If the number is greater, we go on searching.

This algorithm is of course extremely unefficient. It could be improved by searching from unity up, since the number in question is certainly much less than c. Note that there is only one number whose root is that number itself, namely one. There is also only one number whose square root is its half, namely four. There is only one number whose square root is its third, namely nine. Continuing this reasoning, we will quickly see that the ratio between the root a of a number c and that number c, $\frac{a}{c}$, becomes smaller and smaller, the greater c becomes.

We can actually narrow this further down, by observing that there is a relation between the number of digits of a number and the number of digits of the root of that number. For instance, $\sqrt{100} = 10$ and $\sqrt{999} \approx 31$. So, the square roots of numbers with three digits appear to have two digits. The same, however, is true for numbers with four digits, since: $\sqrt{1000} \approx 31$ and $\sqrt{9999} \approx 99$. The relation, hence, appears to be that the number of digits of the root is $\lceil \frac{n}{x} \rceil$, where n is the number of the digits in the power and x is the exponent.

There are some exceptions to this rule. First, for the numbers $\{0 \dots 10\}$, which, for several reasons, are the most peculiar ones, the rule is obviously not true, since $\sqrt{4} = 2$ and $\sqrt{9} = 3$. In more general terms, it is only true if the number of digits in the number in question is greater than the exponent. Otherwise, the root will become very small and, ultimately, approximate unity closer and closer the more we increase the exponent.

Knowing the number of digits of the root, reduces the search space significantly. Instead of looping through 10^n numbers, we only have to search through $10^{\sqrt[x]{n}}$ numbers, that is, from exponential to sub-exponential complexity. But this can still be too much. The square root of a 100-digit number, still, has 10 digits and we, hence, have to loop through 10^{10} numbers.

We will therefore adopt an additional technique: instead of looping through all the numbers by testing and incrementing the number by one, we will narrow the search space by halving it. We will start with the median of the search space, then, if this number is too small, we go half the way up towards the greatest; otherwise, if it is too big, we go half the way down towards the smallest and continue until we find a match.

More precisely, we will start with some distance, which is the half of the search space and start in the middle. As long as we maintain the direction, we also maintain the pace, *i.e.* we reduce the current number by the same distance. Only if we change the direction, we half the distance. We, of course, could halve the distance at every step, whether we change the direction or not. But at some point in time, we will have reduced the distance to unity and cannot reduce it any further. We are then creeping one by one up or down even if we are still far away from our target. To avoid this, we reduce the distance more slowly, risking, perhaps, to overshoot the target several times, but certainly fewer times than we had to increment or decrement by one if we were more conservative in advancing in one direction. Here is a possible implementation:

```
root :: Number → Number → (Number, Number)
root _ [Zero] = ⊥
root [Zero] _ = (zero, zero)
root n [One] = (n, zero)
root [One] _ = (unity, zero)
root n x       =
  if cmp n x ∈ [LT, EQ] then (unity, prev n)
    else let s = len n
         in case s `quotRem2` x of
            ([Zero], _) → ply n x unity unity One
            (k, _)      → ply n x (One : zeros (prev k))
                                  (Five : zeros k) One
  where zeros m = nTake m $ repeat Zero
```

We first take care of the base cases, exponentiation with exponent or base *zero* and exponent or base *unity*. We, then for all other cases, compare the base and the exponent. If the exponent is greater or equal, the result is just *unity* with remainder $n-1$. Whatever the size of a number is, if the exponent is greater or equal, the root must be very close to 1. This rule holds, no matter if n is a small or a huge number, *e.g.* $\sqrt{1} = 1+(1-1) = (1,0)$, $\sqrt{2} = 1 + (2-1) = (1,1)$, $\sqrt[100]{100} = 1 + (100-1) = (1,99)$.

Otherwise, we start working with the number of digits of n divided by the exponent. If the quotient is *zero*, *i.e.* the exponent is greater than the number of digits in n, then we start searching from one incrementing by one. In this case, the number must be small and we have a good chance to find it among the smallest numbers. We do this with the function *ply* that takes five arguments: n, x, the number that we start testing with, the distance we will go up or down and a digit, here *One*, that indicates whether we are going up (*One*) or down (*Zero*).

Otherwise, if the quotient k is greater 0, we start searching at 10^k with steps of 5×10^k, which is the half of 10^{k+1}.

Let us have a look at *ply*:

```
ply :: Number → Number → Number → Number → Digit → (Number, Number)
ply n x b d i = case cmp (power2 b x) n of
                  EQ → (b, zero)
                  GT → let d′ = if i ≡ One then nxt d else d
                       in ply n x (b ‘sub2‘ d′) d′ Zero
                  LT  → case cmp (power2 (next b) x) n of
                        EQ → (next b, zero)
                        GT → (b, n ‘sub2‘ (power2 b x))
                        LT → let d′ = if i ≡ One then d else nxt d
                             in ply n x (b ‘add2‘ d′) d′ One
    where nxt [One] = [One]
          nxt d     = d ‘div‘ two
```

The function first computes the x^{th} power of b, the number we feed into *ply*, and if it equals n, we have found the result. If it is greater, we will reduce b by the distance d. If we came up to this step (i equals *One*), we will now change the direction, going down again. In this case, we halve the distance (if it is not one already). Otherwise, we keep it.

If the result is less than n, we first check if the x^{th} power of *next b* is greater or equals n. If it equals n, we have found the result and terminate. If it is greater, the result is b with a remainder. Otherwise, we increase b by the distance d, which is reduced according to whether we change the direction or not.

Computing the square root of [*One*, *Zero*, *Zero*], for instance, we will first determine the number of digits of [*One*, *Zero*, *Zero*], which is [Three], and divide this result by the exponent [Two], which gives [One]. We, hence, start *ply* with *One*, to which no *Zero*s are appended, and define the distance as *Five*, to which we append one *Zero*. Then we pass through the following steps (where we leave out the first to arguments of *ply*, which are always the same):

ply [*One*] [*Five*, *Zero*] *One*
ply [*Five*, *One*] [*Five*, *Zero*] *One*
ply [*Two*, *Six*] [*Two*, *Five*] *Zero*
ply [*One*] [*Two*, *Five*] *Zero*
ply [*One*, *Three*] [*One*, *Two*] *One*
ply [*Seven*] [*Six*] *Zero*
ply [*One*, *Zero*] [*Three*] *One*

For the case of *One*, *Zero*, *Zero*, *Zero*, we would have $k = 2$ and, hence, would start with [*One*, *Zero*] and *Five*, *Zero*, *Zero*:

```
ply [One, Zero] [Five, Zero, Zero] One
ply [Five, One, Zero] [Five, Zero, Zero] One
ply [Two, Six, Zero] [Two, Five, Zero] Zero
ply [One, Zero] [Two, Five, Zero] Zero
ply [One, Three, Five] [One, Two, Five] One
ply [Seven, Three] [Six, Two] Zero
ply [One, One] [Six, Two] Zero
ply [Four, Two] [Three, One] One
ply [Two, Seven] [Five, Zero] Zero
ply [Three, Four] [Seven] One
ply [Three, One] [Three] Zero
```

Since $32^2 = 1024$, the algorithm stops here with the result ([*Three*, *One*], [*Three*, *Nine*]).

A lot of fine-tuning is possible to improve this algorithm. We can, for example, find the limits of the search space with higher precision, so that we would not start at *unity* to find the square root of [*One*, *Zero*, *Zero*], but at [*One*, *Zero*]. Also, the distance could be selected with more care, in fact, the upper limit for the square root of a number with three digits is not necessarily the greatest two-digit number and the distance should therefore not be initialised to 50. But, for the purpose of the demonstration of search algorithms, the code is sufficient. We are even doing fine: for numbers in the range of 10^{10}, the number of steps is in the range of $20 - 30$, which is acceptable. The steps themselves, however, are heavy, since each one consists of computing the power of, potentially, very large numbers. The *root* function is in any case much slower than *mul2* or *quotRem2*.

We will now look at the logarithm. The algorithm to find the logarithm n base b is in fact much lower in complexity than finding the root. The reason for this is that the logarithm is – usually – a much smaller number than the root (otherwise the root is small or n is really huge). In any case, the search space is always the same. In the root searching algorithm, we limit the search space by giving the lower and upper bound as the number of digits in those numbers, which, of course, leads to a search space that is varying with the size of the bounds. There are, for instance, much more numbers between 1000 and 99 999 than there are between 10 and 999. If we use the same approach for the logarithm, we will find upper and lower bounds that are close to the real result. We would divide the length of n by the length of the base. Let us look at it in the following Haskell implementation, where the first argument is the base and the second is the power:

```
nLog :: Number → Number → (Number, Number)
nLog [Zero] _    = ⊥
nLog _ [Zero]    = ⊥
nLog _ [One]     = (zero, zero)
nLog [One] _     = ⊥
nLog b n         = case (len n) `quotRem2` (len b) of
                     ([Zero], _)        → ⊥
                     ([One], [Zero])    → up unity
```

```
                    (k, _)             → up (prev k)
      where up x = case cmp (power2 b (next x)) n of
                    EQ → (next x, zero)
                    GT → (x, n `sub2` power2 b x)
                    LT → up (next x)
```

The first thing to observe is that there are much more undefined cases than in the root algorithm: There is no exponent, for instance, that will turn zero into any number but zero. That is, for the base zero, there is either no exponent that yields the other number or that other number is zero and than all numbers but zero (which always yields one) would qualify as result. We therefore rule this case out.

The power zero, on the other hand, can only be produced from the base zero and, in that case, all numbers would serve. So we rule this case out as well.

The third undefined case is the base one. This case leads to a meaningful result, only if the power is one as well. In fact any number raised to the zeroth power is one. (This case is handled in the third base case.) Otherwise, if the base is one, but the power is not one, there is no solution.

If we finally come to a case that is not trivial and not undefined, we divide the length of the power n by the length of the base b. If this gives the quotient zero, we know that the base is greater than the power and that is not possible with natural numbered exponents and, hence, this case is ruled out too.

If the result is one without remainder, the two numbers, base and power, are equal in size. There are actually very few numbers that raised to some power result into a number that has not more digits than that number itself (besides of course if that exponent is one). Such numbers are, for instance: 1, which raised to any power is 1; 2, which, squared, is 4 and, raised to the third power, is 8 and, finally, 3, which squared is 9. All other numbers, *e.g.* 4, which squared is 16, will, raised to any power, result in a number hat has more digits than itself. More importantly, there are very few exponents that fulfil that rule, namely 1, for any base, 2, for bases 2 and 3 and 3 for base 2 (note that we have ruled out base 1 already).

So, in this case, base and power are equal in size, we just go slowly up from *unity*, certain to find the exponent we are looking for quite quickly. We do so using the *up* function: This function would raise b to the power *next* x. If the result equals n, we have already found the solution. If the result is greater, we use just x and compute the remainder. (This is actually the reason, we use *next* x, instead of x in *up*. With x, we now had to check if x is zero to avoid an exception, when we now yield the result $x - 1$.) Finally, if the result is less than n, we continue with *next* x.

For the interesting case, where the quotient is anything but one, we call *up* with the predecessor of that quotient. In most cases, we will find the exponent quickly. But as you can see, cases like *nLog two* [*One*, *Zero*, *Zero*, *Zero*] already take some steps. The quotient of the length of the two numbers is $[Four]$. We would hence enter *up* with $[Three]$:

up $[Three]$
up $[Four]$
up $[Five]$
up $[Six]$
up $[Seven]$
up $[Eight]$
up $[Nine]$

and now calculate $2^{10} = 1024$, which is of course greater than 1000, and therefore come to the result $(9, 1000 - 2^9) = (9, 1000 - 512 = 488)$. With greater bases, each step of the algorithm will again be costly, since each time we have to calculate the power of that base.

There are three bases whose logarithms are particularly interesting: the logarithm base 10 ($\log_{10}$) is intersting when we are working in the decimal number system. The logarithm base 2 ($\log_2$) is interesting, when working with the binary number system, but also for many other mathematical objects, some of which we will explore later. Then there is the logarithm to the base e ($\log_e$), the so called *natural logarithm.* This number e, which is approximately 2.71828, is one of the most curious mathematical objects. It appears again and again in apparently unrelated problem areas such as number theory, series of fractions, calculus and so on. It, especially, loves to appear, when you least expect it. We have no means to express this number with natural numbers, so we have to come back to it later to define it properly.

The logarithms with these bases are often shortened. Unfortunately, there are different shorthands in different contexts. Computer scientists would write the binary logarithm log, because it is the most common in their field. This shorthand, however, usually means the natural logarithm in most math publications and even many programming language, including Haskell, use the symbol log for $\log_e$. To make it worse, in many engineering disciplines, $\log_{10}$ is considered the most common logarithm and, accordingly, log is considered to mean $\log_{10}$. There is an ISO standard, which, apparently, nobody is following, that gives the following convention: $\log_2 = lb$, $\log_e = ln$ and $\log_{10} = lg$. But even these shorthands are often confused. The best way, therefore, appears to be the explicit use the symbols.

Logarithms adhere to very interesting arithmetic rules. The logarithm (base b) of the product of two numbers equals the sum of the logarithm (base b) of these numbers: $\log_b(n \times m) = \log_b(n) + \log_b(m)$. Example: $\log_2(4 \times 8) = \log_2(32) = 5$ and $\log_2(4) + \log_2(8) = 2 + 3 = 5$.

Accordingly, the logarithm of the quotient of two numbers equals the difference of the numerator and denominator: $\log_b(\frac{n}{m}) = \log_b(n) - \log_b(m)$, for instance $\log_2(\frac{32}{8}) = \log_2(4) =$

2 and $\log_2(32) - \log_2(8) = 5 - 3 = 2$.

The logarithm of a power of a number n equals the exponent multiplied with the logarithm of n: $\log_b(n^x) = x \times \log_b(n)$, *e.g.*: $\log_2(4^3) = \log_2(64) = 6$ and $3 \times \log_2(4) = 3 \times 2 = 6$.

Finally, the logarithm of a root of n equals the logarithm of n divided by the exponent: $\log_b(\sqrt[x]{n}) = \frac{\log_b(n)}{x}$, for example: $\log_2(\sqrt[3]{64}) = \log_2(4) = 2$ and $\frac{\log_2(64)}{3} = \frac{6}{3} = 2$.

We can also convert logarithms with different bases to each other. Let us assume we want to convert the logarithm base b of a number n to the logarithm base a of n; then $\log_a n = \frac{\log_b n}{\log_b a}$, *i.e.* we divide the logarithm $\log_b n$ by the logarithm $\log_b$ of a. We will later show why this rule holds.

1.7 Numbers as Strings

Until now we have looked at numbers as sequences of symbols, *i.e. strings.* In the next section that will end. We will then define our numbers as a fully-fledged Haskell number type. But before we do that, we will pause shortly to make the difference between the two viewpoints on numbers quite clear. Indeed, in many math problems, the representation of numbers as strings is relevant – especially in informatics. So, this viewpoint is not only related to the way how we happened to define our numbers, but is a genuine mathematical approach.

We have already seen some strange effects of multiplication on the characteristics of the resulting sequences of digits. A much simpler example that shows the properties of numbers as being strings is typing errors. There is no obvious numerical analogy between number pairs like 12 and 21, 26 and 62 or 39 and 93. But, obviously, there is a very simple function that produces these numbers, namely *reverse*:

reverse [*One*, *Two*] = [*Two*, *One*]
reverse [*Two*, *Six*] = [*Six*, *Two*]
reverse [*Three*, *Nine*] = [*Nine*, *Three*]

That is not a numeric property, but a property of any kind of sequence of symbols. Numbers as such, however, are not sequences of symbols. We rather make use of sequences of symbols to represent numbers. In some way, however, any formal system used to represent numbers will have the form of sequences of symbols and, as such, numbers exist in both worlds, a *purely* numerical and a symbolic world.

There is a well known sequence of natural numbers living on the very border between the numerical and the string side of numbers, the *look-and-say* sequence, which is often used in recreational math, but is also investigated by serious (even if playful) mathematicians, such as John H. Conway, co-author of the Book of Numbers. Can you guess how to continue the following sequence?

$1, 11, 21, 1211, 111221, \ldots$

The sequence starts just with one. The next number explains to us what its predecessor looks like: it is composed of one "one". This number, now, is composed of two "ones", which, in its turn, is composed of one "two" and one "one". This again is composed of one "one", one "two" and two "ones". Now, you are surely able to guess the next number.

There are some interesting questions about this sequence. What is the greatest digit that will ever occur in any number of this sequence? Well, we can easily prove that this digit is 3. The numbers of the sequence are composed of pairs of digits that describe groups of equal digits. The first digit of each pair says how often the second digit appears in this group. The number 111221, for instance, describes a number composed of three group: 11 12 21. The first group consists of one "one", the second group of one "two" and the last group of two "ones". Now, it may happen that the digit of the current group coincides with the number of digits in the next group. But the digit in that group must differ from the digits in the current group. Otherwise, it would belong to the current group. A good example is 11 12: if the forth number were 1, like 11 11, then we would have said 21 in the first place. Therefore, there will never be more than three equal numbers in a row and the greatest number to appear in any number is thus 3. □

How can we implement this sequence in Haskell? There seem to be two different principles: First, to describe a given number in terms of groups of digits and, second, to bootstrap a sequence where each number describes its predecessor. Let us implement these two principles separately. The first one is very simple:

```
say :: Number → Number
say xs = concat [len x ++ [head x] | x ← group xs]
```

The *group* function is defined in *Data.List* and groups a list according to repeated elements, exactly what we need. On each element of its result set, we apply *len*, the *length* function for natural numbers we defined earlier, and concatenate this result with the head of that element. For instance, *group* [*One*, *Two*, *One*, *One*] would give [[*One*], [*Two*], [*One*, *One*]]. The length of the first list is [*One*] and concatenated with the head of [*One*] gives [*One*, *One*]. The length of the second list, again, is [*One*] and concatenate with the head of [*Two*] gives [*One*, *Two*]. The length of the third list is [*Two*] and concatenated with the head of [*One*, *One*] is [*Two*, *One*]. Calling *concat* on these results gives [*One*, *One*, *One*, *Two*, *Two*, *One*], which converted to an *Integer*, is 111221.

This function is more general than the sequence, however. We can apply it on any number, also on numbers we would never see in the look-and-say sequence. Applied on *unity*, *say* would just give [*One*, *One*]. Then, from *two* to [*Nine*], the results are quite boring: [*One*, *Two*], [*One*, *Three*], ..., [*One*, *Nine*]. But applied on *ten*, it would result in [*One*, *One*, *One*, *Zero*].

We will now use *say* to implement the look-and-say sequence starting from 1:

```
says :: Number → Number
says []      = []
says [Zero]  = []
says [One]   = [One]
says n       = say (says (prev n))
```

First, we handle the cases that are not part of the sequence: the empty list and *zero*. Then, we handle [*One*], which is just [*One*]. Finally, we define the sequence for any number as *say* of *says* of the predecessor of that number. For instance:

say [*Three*] = *say* (*says* (*prev* [*Three*]))
say [*Three*] = *say* (*says* [*Two*])
say [*Three*] = *say* (*say* (*says* (*prev* [*Two*])))
say [*Three*] = *say* (*say* (*says* [*One*]))
say [*Three*] = *say* (*say* [*One*])
say [*Three*] = *say* ([*One*, *One*])
say [*Three*] = [*Two*, *One*].

Sometimes, the two sides of numbers, their numeric properties and their nature as sequences of digits, become entangled. This is the case with *narcissitic numbers*, a popular concept in recreational math – without further known applications in math or science. Narcissistic numbers are defined by the fact that they equal the sum of their digits raised to the power of the number of digits in the whole number. More formally, a narcissistic number n is a number for which holds:

$$n = \sum_{i=0}^{s} n_i^s,$$

where n_i is the digit of n at position i and s is the number of digits in n. In fact, we can define the property of being narcissistic much clearer as a test in Haskell using our number type:

```
narcissistic :: Number → Bool
narcissistic n = foldr (step (len n)) zero n ≡ n
  where step s a b = b `add2` (power2 [a] s)
```

This property holds trivially for all numbers < *ten*. Then, they get rare. The narcissistic numbers between 10 and 1000 are: 153, 370, 371 and 407. 153, for instance, is narcissistic because $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$. Interesting is the pair 370 and 371: $370 = 3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370$. Now, if we add 1, *i.e.* $1^3 = 1$, 371 arises.

The number of narcissistic numbers in a given number system is limited. This is because for sufficient large ks, the smallest possible number of the form 10^{k-1}, *i.e.* the smallest number with k digits, is greater than the greatest number of the form $k \times 9^k$, *i.e.* the greatest number we can build by adding up the k^{th} powers of the digits of a k-digit-number. That means that, for large numbers, the numerical value will always be greater than the sum of the digits raised to the number of digits in that number. In the decimal system, this limit is reached with $k = 61$. 10^{60} is obviously the smallest number with 61 digits. The 61-digit number with which we can build the greatest sum of 61^{st} powers is the number $99 \dots 9$ that consists of 61 9s. If we raise all these 9s to the 61^{st} power and sum the results, we will obtain a number with 60 digits. That number is clearly less than the least number we can represent with 61 digits. Therefore, no narcissistic numbers with more than 60 digits are possible. In practice, there are only 88 narcissistic numbers in the decimal number system and the greatest of those has 39 digits.

Another popular problem from recreational math is that of a 10-digit number, where each position tells how often the digit related to that position counted from left to right and from 0 to 9 is present in the number. If we represent such a number as in the following table

0	1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i	j

then a would tell how often 0 appears in that number, b, how often 1 appears in that number, c, how often 2 appears in that number and so on.

How can we tackle that problem? First, obviously, the numbers we write in the second line of the table must add up to 10, since these numbers tell how often the related digit appears in the whole number. Since the number has 10 digits, there must be in total 10 occurrences.

We can further assume that the most frequent digit that appears in the number is 0. Otherwise, if a greater digit appeared with high frequency, it would imply that also other numbers must appear more often, since every digit that appears in the number implies another digit to appear. For instance, if 5 was the number with most occurrences, then some numbers must appear 5 times, namely those where we actually put the number 5.

So, let us just try. We could say that 0 occurs 9 times. We would have something like

0	1	2	3	4	5	6	7	8	9
9	0	0	0	0	0	0	0	0	1

This means that 0 appears 9 times and 9 appears once. But there are two problems with this solution: First, if 9 appears once, then 1 appears once as well, but, then, there are only 8 places left to put 0s in. Second, if 1 appearas once (to count 9), then we must put 1 below 1 in the table. But then 1 appears twice, so we must put 2 below 1 and, as a consequence, we must put 1 below 2. In fact, whatever the number of 0s is, for all solutions, we need at least two 1s and one 2, hence:

0	1	2	3	4	5	6	7	8	9
x	2	1	x	x	x	x	x	x	x

Let us think: We have to convert two of the xs into numbers, one into a number that we do not know yet and the other to 1 to count that unknown number. In other words, we will have 2, 1, 1 and some other number. Since we know that the numbers must add up to 10, we can just compute that unknown number as $x = 10 - (2 + 1 + 1) = 10 - 4 = 6$. The result then is 6 210 001 000:

0	1	2	3	4	5	6	7	8	9
6	2	1	0	0	0	1	0	0	0

Is this the only possible configuration, or are there others that fulfil the constraints? Let us assume there is another configuration. We already know that 7,8 and 9 do not work. So, instead of 6, we will have a number smaller than 6. This number could be 5. Then, we have to make up the difference between 6 and 5, since the numbers, at the end, must add up to 10. That means, we need one more 1. But, then, we need an additional number that occurs once to justify that additional 1. But, since there is only room for one additional number, that cannot be.

Then, we could try 4. But 4 does not work either, since the difference now is 2 and we cannot just increase the ocurrences of 1 or 2 without justification. If we increased the occurrences of 1, we would have to add another number, to justify that additional 1. We need, in fact, three numbers, but we have only room for two.

Then 3 could be a solution: instead of one 6, we would have two 3s. But now, we must justify the second 3 and there is no room for another number appearing three times. So, since 1 and 2, obviously, will not work, we conclude, that 6 210 001 000 is the only possible configuration.

We are now very close to leave the world where we look at numbers mainly as strings. We will soon look at numbers in a completely different way. But before we do that, we still have to finalise the model of our number type, that is, we should define how to convert an integer into our *Number*. We can of course just convert the integer into a string using *show* and then convert the string digit by digit to *Number*. But, again, that would be boring. We would not learn anything special about numbers, which is the main concern of all our exercises here.

Instead, we will think along the lines of decimal numbers being representations of powers of 10. We will ask: how many powers of 10 does a given number contain? How many powers of 10 are, for example, in the number 9827? To answer this question, we first have to find the floor of $\log_{10} 9827$, *i.e.* a number l such that $10^l \leq 9827$ and $10^{l+1} > 9827$: $\lfloor \log_{10} 9827 \rfloor$. For 9827, that is 3, since $10^3 = 1000$ and $10^4 = 10000$. To learn how many third powers of 10 are in the number, we divide the number by the third power of 10: $\left\lfloor \frac{9827}{10^3} \right\rfloor = 9$. We, hence, have 9 times the third power of 10 in 9827. The first digit is therefore *Nine*. To convert the whole number, we now apply the algorithm on the remainder of the division $\frac{9827}{10^3}$, which is 827.

But hold on: is this not quite expensive with a log operation and a division on each digit of the original integer? Yes, in fact, we can think much simpler in terms of modulo. A number in the decimal system is composed of the digits $0 \dots 9$. Any number modulo 10 is one of these digits. The remainder of 9827 and 10, for instance, is 7, because the Euclidian division of 9827 and 10 is $(982, 7)$; the Euclidian division of 982 and 10 is $(98, 2)$; the result for 98 and 10 is $(9, 8)$ and that for 9 and 10 is just $(0, 9)$. In other words, we can just collect the remainders of the Euclidian division of the integer and 10 and convert each digit into our *Digit* type. Here is the code in Haskell:

```
integer2Num :: Integer → Number
integer2Num 0    = zero
integer2Num 1    = unity
integer2Num 2    = two
integer2Num 3    = [Three]
integer2Num 4    = [Four]
integer2Num 5    = [Five]
integer2Num 6    = [Six]
integer2Num 7    = [Seven]
integer2Num 8    = [Eight]
integer2Num 9    = [Nine]
integer2Num 10   = ten
integer2Num i    = go i
  where go n = case n `quotRem` 10 of
                 (0, r) →          integer2Num r
                 (q, r) → go q ++ integer2Num r
```

We start by handling all one-digit numbers and 10 explicitly This has two advantages: we speed up the processing for one-digit numbers and 10 and we do not need an extra conversion function for digits.

For all values of i not handled in the base cases, we compute quotient and remainder. If the quotient is 0, we are done with *go* and just yield the conversion of r, which must be a digit, since it is a remainder of division by 10. Otherwise, we continue with the quotient to which we append the conversion of the remainder.

For the example 9827, we would create the following sequence:

go 9827 = *go* 827 ++ *integer2Num* 7
go 982 = *go* 82 ++ *integer2Num* 2 ++ *integer2Num* 7
go 98 = *go* 8 ++ *integer2Num* 8 ++ *integer2Num* 2 ++ *integer2Num* 7
go 9 = *integer2Num* 9 ++ *integer2Num* 8 ++ *integer2Num* 2 ++ *integer2Num* 7,

which is

[*Nine*] ++ [*Eight*] ++ [*Two*] ++ [*Seven*]
[*Nine*, *Eight*, *Two*, *Seven*].

1.8 $\mathbb{N}$

We will now convert our *Number* type in a full-fledged Haskell *Num* type. This will allow us to use numeric symbols, *i.e.* the number $0 \dots 9$, instead of the constructors *Zero*... *Nine*, for our type and we will be able to use the standard operators $+, -, *$. The first step is to define a **data** type – until now, we used only a type synonym for [*Digit*]:

```
data Natural = N Number
```

The new data type is called *Natural* and its only constructor is *N* receiving a *Number*, *i.e.* [*Digit*], as parameter. The constructor *N* is named after the symbol for the set of natural numbers in math, which is $\mathbb{N}$.

We, then, make this data type instance of *Eq* and *Show*, where we use the previous defined functions *cmp* for comparisons and *n2Integer* for conversion to *Int* and subsequent *show*:

```
instance Eq Natural where
  (N a) ≡ (N b) = cmp a b ≡ EQ
instance Show Natural where
  show (N ns) = show (n2Integer ns)
```

Now we are ready to make *Natural* instance of *Num*. *Num* has the following methods we have to implement: $+$, $-$, $*$, *negate*, this would be a negative number, which we have not yet defined, so we leave this method undefined, *abs*, the absolute value of a number, *signum*, which is either 0 (for *zero*), 1 (for numbers > 0) or -1 (for numbers < 0), and *fromInteger*, a conversion function that turns instances of type class *Integral*, like *Int* and *Integer*, into our data type. Here is the code:

```
instance Num Natural where
  (N as) + (N bs)                    = N (as `add2` bs)
  (N as) − (N bs) | cmp as bs ≡ LT = error "subtraction below zero"
     | otherwise                     = N (as `sub2` bs)
  (N as) ∗ (N bs)                    = N (as `mul2` bs)
  negate  n                          = ⊥
  abs     n                          = n
  signum (N [Zero])                  = 0
  signum n                           = 1
  fromInteger i                      = N (integer2Num i)
```

Two *Natural*s are added by adding the *Number*s of which they consists using *add2* and calling the constructor *N* on the result. Subtraction and multiplication are implemented accordingly using *sub2* and *mul2* respectively. *abs n* is just *n*, since Natural is always a positive number, we do not need to worry about negative numbers passed in to *abs*.

signum for *zero* is just 0, for any other number, it is 1. Again, because of their absence, we do not need to handle negative numbers. For *fromInteger*, we finally use the conversion function *integer2Num*.

There are some other properties we would like our number type to have. First, numbers, in Haskell, are also *Enums*, *i.e.* objects that can be enumerated. The class *Enum* defines the methods *succ* and *pred* – which we already know from Peano numbers – *toEnum*, the conversion of integrals, especially *Int*s, to our data type, and *fromEnum*, the opposite conversion:

```
instance Enum Natural where
  succ (N n)         = N (next n)
  pred (N [Zero])    = error "zero has no predecessor"
  pred (N n)         = N (prev n)
  toEnum             = N ∘ integer2Num ∘ fromIntegral
  fromEnum (N n)     = fromIntegral (n2Integer n)
```

Numbers, additionally, have order. For every two numbers, we can say which of the two is greater or less than the other. This is captured by the type class *Ord*. The only method we have to implement for making *Natural* instance of *Ord* is compare:

```
instance Ord Natural where
  compare (N as) (N bs) = cmp as bs
```

We also want to be able to convert our numbers into real numbers. To do so, we make *Natural* an instance of *Real*:

```
instance Real Natural where
  toRational (N ns) = fromIntegral $ n2Integer ns
```

Finally, our number type is a kind of integral, *i.e.* not a fraction. To express this in Haskell, we make *Natural* instance of the *Integral* class and implement the methods *quotRem* and *toInteger*. The code is quite obvious, no further explanations are necessary:

```
instance Integral Natural where
  quotRem (N as) (N bs) = let (q, r) = D.quotRem2 as bs in (N q, N r)
  div     a b           = fst $ quotRem a b
  toInteger (N ns)      = n2Integer ns
```

1.9 Abstract Algebra

When we look at the four fundamental arithmetic operations, addition, multiplication, subtraction and division, we see some striking differences between them. We can, more specifically, distinguish two groups of operations, namely addition and multiplication on one side and subtraction and division on the other.

for multiplication and addition, we can state that for any two natural numbers a and b, the result of the operations $a + b$ and $a \times b$ is again a natural number. For subtraction and division that is not true. As you may remember, for subtraction, we had to define an important exception, *viz.* that the second term must not be greater than the first one. Otherwise, the result is not a natural number.

Division according to Euclid, besides having the exception of *zero* in the denominator, differs completely, in that its result is not at all a number, but a pair of numbers (q, r). If we refer to division in terms of the *quot* operation (which returns only the quotient, not the remainder), then division would indeed behave similar to addition and multiplication (besides the division-by-zero exception). But that would leave us with a torso operation that falls behind the other operations in precision and universality.

Another property shared by addition and multiplication is the *associative law*:

$$a + (b + c) = (a + b) + c = a + b + c \tag{1.5}$$

$$a \times (b \times c) = (a \times b) \times c = a \times b \times c \tag{1.6}$$

This, again, is not true for subtraction and division, as can be easily shown by counter examples:

$$4 - (3 - 1) \neq (4 - 3) - 1,$$

since

$$4 - (3 - 1) = 4 - 2 = 2$$

and

$$(4 - 3) - 1 = 1 - 1 = 0.$$

For division, we cannot even state such an equality (or inequality), since the result of the Euclidian division, a pair of numbers, cannot serve as one of its arguments, *i.e.* a pair of numbers cannot be divided. If we, again, accept the *quot* operation as a compromise, we quickly find counter examples which show that the associative law does not hold for division either:

$$3/(2/2) \neq (3/2)/2,$$

since

$$3/(2/2) = 3/1 = 3$$

and

$$(3/2)/2 = 1/2 = 0.$$

Abstract Algebra uses such properties to define different classes of numbers and other "things" – of which we will soon see some examples. The first class of such things we can define is the *magma* or *groupoid*. A magma is a set together with a binary operation such that the set is closed under this operation. We will look at sets in more detail in the next section; for the moment, we can live with an informal intuition of sets being collections of things, here of certain types of numbers. That a set is closed under an operation is just the property we defined first, *i.e.* that c is a natural number if a and b are natural numbers in $a + b = c$. More formally, we can describe a magma as

$$M = (S, \cdot), \tag{1.7}$$

where S is a set and $\cdot$ is a binary operation, such that for all $a, b \in S$ (a and b are *element of* S, *i.e.* they are members of the set S), $a \cdot b \in S$, *i.e.* the result of the operation $a \cdot b$, too, is in S.

When we add the other property, the associative law, to the magma definition, we get a *semigroup*. A semigroup, hence, is a magma, where for the operation $\cdot$ the relation $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ holds.

Natural numbers with either addition or multiplication are clearly semigroups. We can be even more specific: Natural numbers are *abelian semigroups*, since the *commutative law* holds for them as well. The commutative law states that, for an operation $\cdot$ the relation: $a \cdot b = b \cdot a$ holds, which, again, is not true for subtraction and division.

The next property is the identity. This property states that there is an element e in S, for which holds that $a \cdot e = e \cdot a = a$. For addition and subtraction, this element e is *zero*. For multiplication, it is *unity*. For a division operation defined as *quot* this element would be *unity* as well.

A semigroup with identity is called a *monoid*. Natural numbers are hence abelian monoids, since the commutative law holds for them as well. An example for a non-abelian monoid is the set of all strings, STR, with the concatenation operation $+\!\!+$. First note that STR is closed under concatenation, since, for any strings a and b, it holds that (using Haskell syntax) $a +\!\!+ b$ is again a string, *e.g.* `"hello "` $+\!\!+$ `"world"` $\equiv$ `"hello world"`.

Then, the associative law holds, since for any three strings, a, b and c: $a +\!\!+ (b +\!\!+ c) \equiv$

$(a +\!\!+\, b) +\!\!+\, c \equiv a +\!\!+\, b +\!\!+\, c$, for instance: `"hello"` $+\!\!+$ (`" "` $+\!\!+$ `"world"`) $\equiv$ (`"hello"` $+\!\!+$ `" "`) $+\!\!+$ `"world"` $\equiv$ `"hello world"`.

Next, there is an identity, *viz.* the empty string "", such that: $a +\!\!+$ `""` $\equiv$ `""` $+\!\!+\, a \equiv a$, for instance: `"hello world"` $+\!\!+$ `""` $\equiv$ `"hello world"`.

Note, however, that the STR monoid is not commutative: $a +\!\!+\, b \not\equiv b +\!\!+\, a$, for instance: `"hello "` $+\!\!+$ `"world"` $\not\equiv$ `"world"` $+\!\!+$ `"hello "`.

The term *semigroup* suggests that there is also something called a *group*, which, in some way, is more complete than a semigroup – and, indeed, there is. A group is a monoid with the addtional property of *invertibility.* Invertibility means that there is an element to invert the effect of an operation, such that for any a and b for which holds: $a \cdot b = c$, there is an element x, such that: $c \cdot x = a$. Note that this implies for b and its inverse element x: $b \cdot x = e$, where e is the identity.

Unfortunately for our poor natural numbers, there are no such elements with addition and multiplication. Note, however, that, if we had already introduced negative numbers and fractions, there would ineed exist such elements, namely for addition: $a + x = 0$, where obviously $x = -a$ and for multiplication: $a \times x = 1$ with $x = \frac{1}{a}$.

Let us summarise the fundamental properties of binary operations to keep track of all the properties that may hold for different types of objects:

	closure	associativty	identity	invertibility	commutativty
	$a \cdot b \in S$	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	$a \cdot e = e \cdot a = a$	$a \cdot \frac{1}{a} = e,$	$a \cdot b = b \cdot a$
magma	$\times$				
semigroup	$\times$	$\times$			
monoid	$\times$	$\times$	$\times$		
group	$\times$	$\times$	$\times$	$\times$	
abelian x	$\times$	-	-	-	$\times$

It is to be noted that any of the concepts magma, semigroup, monoid and group may have the property of being abelian. There are abelian magmas, semigroups, monoids and groups. Therefore, the *abelian x* is indifferent towards associativity, identitiy and invertibility. This depends entirely on the x, not on the x being abelian or not.

We will now introduce a major step. We have, so far, added additional propterties to magmas, semigroups and so on to create new kinds of objects. Now, we change the underlying definition to create something completely different, namely a *semiring.* A semiring is a set S together with **two** binary operations, denoted $\bullet$ and $\circ$:

$$R = (S, \bullet, \circ). \tag{1.8}$$

The operation $\bullet$ must form an abelian monoid with S and the operation $\circ$ must form a monoid (which may or may not be abelian) with S. These conditions are fulfilled for addition and multiplication on the natural numbers. Since both, addition and multiplication, form abelian monoids, for the moment, both may take either place in the definition. But there is one more property: the operations together must adhere to the *distributive law*, which states that

$$a \circ (b \bullet c) = (a \circ b) \bullet (a \circ c). \tag{1.9}$$

This, again, is true for natural numbers, if $\bullet$ corresponds to addition and $\circ$ to multiplication: $a \times (b + c) = (a \times b) + (a \times c)$. We can simplify this formula by leaving the parentheses out of course: $a \times b + a \times c$ and can even further simplify by adopting the usual convention that $a \times b = ab$: $ab + ac$.

A *ring* is a semiring, for which the additional property of invertibility holds on addition. A ring, hence, consists of an abelian group (addition in case of natural numbers) and a monoid (multiplication). Again, natural numbers do not form a ring, but only a semiring, since there are no negative numbers in natural numbers and there is thus no inverse for addition.

A ring, where multiplication is commutative, hence, a ring with an abelian group (addition) and an abelian monoid (multiplication) is called a *commutative ring*.

The most complete structure, however, is the *field*, where both operations, addition and multiplication are abelian groups.

Here is the complete taxonomy:

	addition	multiplication
semiring	abelian monoid	monoid
ring	abelian group	monoid
commutative ring	abelian group	abelian monid
field	abelian group	abelian group