Chapter 3

Numbers

Numbers in Haskell are complicated because in the Haskell world there are many different kinds of number, including:

`Int`	limited-precision integers in at least the range $[-2^{29}, 2^{29})$. Integer overflow is not detected.
`Integer`	arbitrary-precision integers
`Rational`	arbitrary-precision rational numbers
`Float`	single-precision floating-point numbers
`Double`	double-precision floating-point numbers
`Complex`	complex numbers (defined in `Data.Complex`)

Most programs make use of numbers in one way or another, so we have to get at least a working idea of what Haskell offers us and how to convert between the different kinds. That is what the present chapter is about.

3.1 The type class `Num`

In Haskell all numbers are instances of the type class `Num`:

```
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate      :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

The class `Num` is a subclass of both `Eq` and `Show`. That means every number can be printed and any two numbers can be compared for equality. Any number can be added to, subtracted from or multiplied by another number. Any number can be

https://doi.org/10.1017/CBO9781316092415.004

negated. Haskell allows `-x` to denote `negate x`; this is the only prefix operator in Haskell.

The functions `abs` and `signum` return the absolute value of a number and its sign. If ordering operations were allowed in `Num` (and they aren't because, for example, complex numbers cannot be ordered), we could define

```
abs x    = if x < 0 then -x else x
signum x | x < 0  = -1
         | x == 0 = 0
         | x > 0  = 1
```

The function `fromInteger` is a conversion function. An integer literal such as `42` represents the application of `fromInteger` to the appropriate value of type `Integer`, so such literals have type `Num a => a`. This choice is explained further below after we have considered some other classes of number and the conversion functions between them.

3.2 Other numeric type classes

The `Num` class has two subclasses, the real numbers and the fractional numbers:

```
class (Num a,Ord a) => Real a where
  toRational :: a -> Rational

class (Num a) => Fractional a where
  (/) :: a -> a -> a
  fromRational :: Rational -> a
```

Real numbers can be ordered. The only new method in the class `Real`, apart from the comparison operations which are inherited from the superclass `Ord`, is a conversion function from elements in the class to elements of `Rational`. The type `Rational` is essentially a synonym for pairs of integers. The real number π is not rational, so `toRational` can only convert to an approximate rational number:

```
ghci>  toRational pi
884279719003555 % 281474976710656
```

Not quite as memorable as `22 % 7`, but more accurate. The symbol `%` is used to separate the numerator and denominator of a rational number.

The fractional numbers are those on which division is defined. A complex number cannot be real but it can be fractional. A floating-point literal such as `3.149` represents the application of `fromRational` to an appropriate rational number. Thus

```
3.149 ::  Fractional a => a
```

This type and the earlier type `Num a => a` for `42` explains why we can form a legitimate expression such as `42 + 3.149`, adding an integer to a floating-point number. Both types are members of the `Num` class and all numbers can be added. Consideration of

```
ghci> :type 42 + 3.149
42 + 3.149 :: Fractional a => a
```

shows that the result of the addition is also a fractional number.

One of the subclasses of the real numbers is the integral numbers. A simplified version of this class is:

```
class (Real a, Enum a) => Integral a where
  divMod :: a -> a -> (a,a)
  toInteger :: a -> Integer
```

The class `Integral` is a subclass of `Enum`, those types whose elements can be enumerated in sequence. Every integral number can be converted into an `Integer` through the conversion function `toInteger`. That means we can convert an integral number into any other type of number in two steps:

```
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```

Application of `divMod` returns two values:

```
x `div` y  = fst (x `divMod` y)
x `mod` y  = snd (x `divMod` y)
```

The standard prelude functions `fst` and `snd` return the first and second components of a pair:

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y
```

Mathematically, $x \operatorname{div} y = \lfloor x/y \rfloor$. We will see how to compute $\lfloor x \rfloor$ in the following section. And x mod y is defined by

$$x = (x \operatorname{div} y) * y + x \bmod y$$

For positive x and y we have $0 \leq x \bmod y < x$.

Recall the function `digits2` from the first chapter, where we defined

```
digits2 n = (n `div` 10, n `mod` 10)
```

It is more efficient to say ``digits2 n = n `divMod` 10`` because then only one invocation of `divMod` is required. Even more briefly, we can use a section and write ``digits2 = (`divMod` 10)``.

There are also other numeric classes, including the subclass `Floating` of the class `Fractional` that contains, among others, the logarithmic and trigonometric functions. But enough is enough.

3.3 Computing floors

The value $\lfloor x \rfloor$, the *floor* of x, is defined to be the largest integer m such that $m \leq x$. We define a function `floor :: Float -> Integer` for computing floors. Haskell provides such a function in the standard prelude, but it is instructive to consider our own version.

One student, call him Clever Dick, to whom this task was given came up with the following solution:

```
floor :: Float -> Integer
floor = read . takeWhile (/= '.') . show
```

In words, the number is shown as a string, the string is truncated by taking only the digits up to the decimal point, and the result is read again as an integer. We haven't met `takeWhile` yet, though Clever Dick evidently had. Clever Dick's solution is wrong on a number of counts, and Exercise D asks you to list them.

Instead we will find the floor of a number with the help of an explicit search, and for that we will need a loop:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x = if p x then x else until p f (f x)
```

The function `until` is also provided in the standard prelude. Here is an example:

https://doi.org/10.1017/CBO9781316092415.004

```
ghci>  until (>100) (*7) 1
343
```

Essentially `until f p x` computes the first element `y` in the infinite list

```
[x, f x, f (f x), f (f (f x)), ...]
```

for which `p y = True`. See the following chapter where this interpretation of `until` is made precise.

Thinking now about the design of `floor` it is tempting to start off with a case analysis, distinguishing between the cases $x < 0$ and $x \geq 0$. In the case $x < 0$ we have to find the first number m in the sequence $-1, -2, \ldots$ for which $m \leq x$. That leads to – in the case of a negative argument –

```
floor x = until (`leq` x) (subtract 1) (-1)
          where m `leq` x = fromInteger m <= x
```

There are a number of instructive points about this definition. Firstly, note the use of the prelude function `subtract` whose definition is

```
subtract x y = y-x
```

We have to use `subtract 1` because `(-1)` is *not* a section but the number -1 (look at the third argument of `until`).

Secondly, why have we used `` `leq` `` when the alternative `(<=)` seems perfectly adequate? The answer is that `(<=)` has the type

```
(<=) :: Num a => a -> a -> Bool
```

In particular the two arguments of `(<=)` have to have the same type. But we want

```
leq :: Integer -> Float -> Bool
```

and the two arguments have different numeric types. We therefore need to convert integers to floats using `fromInteger`. Appreciation of the need for conversion functions in some situations is one of the key points to understand about Haskell arithmetic.

Finally, note that ``(`leq` x)`` is not the same as `(leq x)`:

```
(leq x)   y = leq x y
(`leq` x) y = y `leq` x = leq y x
```

It is easy to make this mistake.

If you don't like the subsidiary definition, you can always write

```
floor x = until ((<=x) . fromInteger) (subtract 1) (-1)
```

In this version we have *inlined* the definition of (`leq` x).

We still have to deal with the case $x \geq 0$. In this case we have to look for the first integer n such that $x < n+1$. We can do this by finding the first integer n such that $x < n$ and subtracting 1 from the answer. That leads to

```
floor x = until (x `lt` ) (+1) 1 - 1
          where x `lt` n = x < fromInteger n
```

Putting the two pieces together, we obtain

```
floor x = if x < 0
          then until (`leq` x) (subtract 1) (-1)
          else until (x `lt`) (+1) 1 - 1
```

(Question: why do we not have to write `x < fromInteger 0` in the first line?) The real problem with this definition, apart from the general ugliness of a case distinction and the asymmetry of the two cases, is that it is very slow: it takes about $|x|$ steps ($|x|$ is the mathematician's way of writing `abs x`) to deliver the result.

Binary search

A better method for computing `floor` is to first find integers m and n such that $m \leq x < n$ and then shrink the interval (m,n) to a unit interval (one with $m+1=n$) that contains x. Then the left-hand bound of the interval can be returned as the result. That leads to

```
floor :: Float -> Integer
floor x = fst (until unit (shrink x) (bound x))
          where unit (m,n) = (m+1 == n)
```

The value `bound x` is some pair (m,n) of integers such that $m \leq x < n$. If (m,n) is not a unit interval, then `shrink x (m,n)` returns a new interval of strictly smaller size that still bounds x.

Let us first consider how to shrink a non-unit interval (m,n) containing x, so $m \leq x < n$. Suppose p is any integer that satisfies $m < p < n$. Such a p exists since (m,n) is not a unit interval. Then we can define

```
type Interval = (Integer,Integer)

shrink :: Float -> Interval -> Interval
```

```
shrink x (m,n) = if p `leq` x then (p,n) else (m,p)
                 where p = choose (m,n)
```

How should we define `choose`?

Two possible choices are `choose (m,n) = m+1` or `choose (m,n) = n-1` for both reduce the size of an interval. But a better choice is

```
choose :: Interval -> Integer
choose (m,n) = (m+n) `div` 2
```

With this choice the size of the interval is halved at each step rather than reduced by 1.

However, we need to check that $m < (m+n) \operatorname{div} 2 < n$ in the case $m+1 \neq n$. The reasoning is:

$$
\begin{array}{cl}
 & m < (m+n) \operatorname{div} 2 < n \\
\equiv & \{\text{ordering on integers}\} \\
 & m+1 \leq (m+n) \operatorname{div} 2 < n \\
\equiv & \{\text{since } (m+n) \operatorname{div} 2 = \lfloor (m+n)/2 \rfloor\} \\
 & m+1 \leq (m+n)/2 < n \\
\equiv & \{\text{arithmetic}\} \\
 & m+2 \leq n \wedge m < n \\
\equiv & \{\text{arithmetic}\} \\
 & m+1 < n
\end{array}
$$

Finally, how should we define `bound`? We can start off by defining

```
bound :: Float -> Interval
bound x = (lower x, upper x)
```

The value `lower x` is some integer less than or equal to `x`, and `upper x` some integer greater than `x`. Instead of using linear search to discover these values, it is better to use

```
lower :: Float -> Integer
lower x = until (`leq` x) (*2) (-1)

upper :: Float -> Integer
upper x = until (x `lt`) (*2) 1
```

For a fast version of `bound` it is better to double at each step rather than increase or decrease by 1. For example, with $x = 17.3$ it takes only seven comparisons to compute the surrounding interval $(-1,32)$, which is then reduced to $(17,18)$ in a further five steps. In fact, evaluating both the upper and lower bounds takes time proportional to $\log|x|$ steps, and the whole algorithm takes at most twice this time. An algorithm that takes logarithmic time is much faster than one that takes linear time.

The standard prelude defines `floor` in the following way:

```
floor x = if r < 0 then n-1 else n
          where (n,r) = properFraction x
```

The function `properFraction` is a method in the `RealFrac` type class (a class we haven't discussed and whose methods deal with truncating and rounding numbers). It splits a number x into its integer part n and its fractional part r, so $x = n + r$. Now you know.

3.4 Natural numbers

Haskell does not provide a type for the natural numbers, that is, the nonnegative integers. But we can always define such a type ourselves:

```
data Nat = Zero | Succ Nat
```

This is an example of a *data declaration*. The declaration says that `Zero` is a value of `Nat` and that `Succ n` is also a value of `Nat` whenever `n` is. Both `Zero` and `Succ` are called *data constructors* and begin with a capital letter. The type of `Zero` is `Nat` and the type of `Succ` is `Nat -> Nat`. Thus each of

```
Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero))
```

is an element of `Nat`.

Let us see how to program the basic arithmetical operations by making `Nat` a fully paid-up member of the `Num` class. First, we have to make `Nat` an instance of `Eq` and `Show`:

```
instance Eq Nat where
 Zero == Zero     = True
 Zero == Succ n   = False
 Succ m == Zero   = False
 Succ m == Succ n = (m == n)
```

```
instance Show Nat where
 show Zero            = "Zero"
 show (Succ Zero)     = "Succ Zero"
 show (Succ (Succ n)) = "Succ (" ++ show (Succ n) ++ ")"
```

These definitions make use of *pattern matching*. In particular, the definition of `show` makes use of three patterns, `Zero`, `Succ Zero` and `Succ (Succ n)`. These patterns are different from one another and together cover all the elements of `Nat` apart from $\perp$.

Alternatively, we could have declared

```
data Nat = Zero | Succ Nat  deriving (Eq,Ord,Show)
```

As we said in Exercise E of the previous chapter, Haskell is smart enough to construct automatically instances of some standard classes, including `Eq`, `Ord` and `Show`.

Now we can install `Nat` as a numeric type:

```
instance Num Nat where
  m + Zero     = m
  m + Succ n   = Succ (m+n)

  m * Zero     = Zero
  m * (Succ n) = m * n + m

  abs n           = n
  signum Zero     = Zero
  signum (Succ n) = Succ Zero

  m - Zero        = m
  Zero - Succ n   = Zero
  Succ m - Succ n = m - n

  fromInteger x
   | x <= 0    = Zero
   | otherwise = Succ (fromInteger (x-1))
```

We have defined subtraction as a total operation: $m-n=0$ if $m \leq n$. Of course, the arithmetic operations on `Nat` are horribly slow. And each number takes up a lot of space.

Partial numbers

We have said that there is a value $\perp$ of every type. Thus `undefined :: a` for all types `a`. Since `Succ` is, by definition, a non-strict function, the values

```
undefined, Succ undefined, Succ (Succ undefined), ...
```

are all different and all members of `Nat`. To be honest, these partial numbers are not very useful, but they are there. You can think of `Succ undefined` as being a number about which we know only that it is at least 1:

```
ghci> Zero == Succ undefined
False
ghci> Succ Zero == Succ undefined
*** Exception: Prelude.undefined
```

There is also one further number in `Nat`:

```
infinity :: Nat
infinity = Succ infinity
```

Thus

```
ghci> Zero == infinity
False
ghci> Succ Zero == infinity
False
```

and so on.

In summary, the elements of `Nat` consist of the finite numbers, the partial numbers and the infinite numbers (of which there is only one). We shall see that this is true of other data types: there are the finite elements of the type, the partial elements and the infinite elements.

We could have chosen to make the constructor `Succ` strict. This is achieved by declaring

```
data Nat = Zero | Succ !Nat
```

The annotation `!` is known as *strictness flag*. With such a declaration, we have for example

```
ghci> Zero == Succ undefined
*** Exception: Prelude.undefined
```

This time, evaluating the equality test forces the evaluation of both sides, and the evaluation of `Succ undefined` raises an error message. Making `Succ` strict collapses the natural numbers into just the finite numbers and one undefined number.

3.5 Exercises

Exercise A

Which of the following expressions denote 1?

```
-2 + 3, 3 + -2, 3 + (-2), subtract 2 3, 2 + subtract 3
```

In the standard prelude there is a function `flip` defined by

```
flip f x y = f y x
```

Express `subtract` using `flip`.

Exercise B

Haskell provides no fewer than three ways to define exponentiation:

```
(^)  :: (Num a, Integral b) => a -> b -> a
(^^) :: (Fractional a, Integral b) => a -> b -> a
(**) :: (Floating a) => a -> a -> a
```

The operation `(^)` raises any number to a nonnegative integral power; `(^^)` raises any number to any integral power (including negative integers); and `(**)` takes two fractional arguments. The definition of `(^)` basically follows Dick's method of the previous chapter (see Exercise E). How would you define `(^^)`?

Exercise C

Could you define `div` in the following way?

```
div :: Integral a => a -> a -> a
div x y = floor (x/y)
```

Exercise D

Consider again Clever Dick's solution for computing `floor`:

```
floor :: Float -> Integer
floor = read . (takeWhile (/= '.') . show
```

Why doesn't it work?

Consider the following mini-interaction with GHCi:

```
ghci> 12345678.0 :: Float
1.2345678e7
```

Haskell allows the use of so-called *scientific notation*, also called *exponent notation*, to describe certain floating-point numbers. For example the number above denotes $1.2345678 * 10^7$. When the number of digits of a floating-point number is sufficiently large, the number is printed in this notation. Now give another reason why Clever Dick's solution doesn't work.

Exercise E

The function `isqrt :: Float -> Integer` returns the floor of the square root of a (nonnegative) number. Following the strategy of Section 3.3, construct an implementation of `isqrt x` that takes time proportional to $\log x$ steps.

Exercise F

Haskell provides a function `sqrt :: Floating a => a -> a` that gives a reasonable approximation to the square root of a (nonnegative) number. But, let's define our own version. If y is an approximation to $\sqrt{x}$, then so is x/y. Moreover, either $y \leq \sqrt{x} \leq x/y$ or $x/y \leq \sqrt{x} \leq y$. What is a better approximation to $\sqrt{x}$ than either y or x/y? (Yes, you have just rediscovered Newton's method for finding square roots.)

The only remaining problem is to decide when an approximation y is good enough. One possible test is $|y^2 - x| < \varepsilon$, where $|x|$ returns the absolute value of x and ε is a suitably small number. This test guarantees an *absolute* error of at most ε. Another test is $|y^2 - x| < \varepsilon * x$, which guarantees a *relative* error of at most ε. Assuming that numbers of type `Float` are accurate only to six significant figures, which of these two is the more sensible test, and what is a sensible value for ε?

Hence construct a definition of `sqrt`.

Exercise G

Give an explicit instance of `Nat` as a member of the type class `Ord`. Hence construct a definition of

```
divMod :: Nat -> Nat -> (Nat,Nat)
```

3.6 Answers

Answer to Exercise A

All except `2 + -3` and `2 + subtract 3`, neither of which are well-formed. We have `subtract = flip (-)`.

Answer to Exercise B

```
x ^^ n = if 0 <= n then x^n else 1/(x ^ (negate n))
```

Answer to Exercise C

No. You would have to write

```
div :: Integral a => a -> a -> a
div x y = floor (fromInteger x / fromInteger y)
```

Answer to Exercise D

Clever Dick's function gives `floor (-3.1) = -3` when the answer should be `-4`. And if you tried to repair his solution by subtracting 1 if the solution was negative, you would have `floor (-3.0) = -4` when the answer should be `-3`. Ugh!

Also, Clever Dick's solution has `floor 12345678.0 = 1` because the argument is shown as `1.2345678e7`.

Answer to Exercise E

```
isqrt :: Float -> Integer
isqrt x = fst (until unit (shrink x) (bound x))
          where unit (m,n) = (m+1 == n)

shrink :: Float -> Interval -> Interval
shrink x (m,n) = if (p*p) `leq` x then (p,n) else (m,p)
                 where p = (m+n) `div` 2

bound :: Float -> Interval
bound x = (0,until above (*2) 1)
          where above n = x `lt` (n*n)
```

The functions `` `leq` `` and `` `lt` `` were defined in Section 3.3. Note the parentheses in the expressions ``(p*p) `leq` x`` and ``x `lt` (n*n)``. We didn't state an order of association for `` `leq` `` and `` `lt` ``, so without parentheses these two expressions would have been interpreted as the ill-formed expressions `p * (p `leq` x)` and `(x `lt` n) * n`. (I made just this mistake when first typing in the solution.)

Answer to Exercise F

A better approximation to $\sqrt{x}$ than either y or x/y is $(y+x/y)/2$. The relative-error test is the more sensible one, and the program is

```
sqrt :: Float -> Float
sqrt x = until goodenough improve x
         where goodenough y = abs (y*y-x) < eps*x
               improve y    = (y+x/y)/2
               eps          = 0.000001
```

Answer to Exercise G

It is sufficient to define `(<)`:

```
instance Ord Nat where
  Zero < Zero     = False
  Zero < Succ n   = True
  Succ m < Zero   = False
  Succ m < Succ n = (m < n)
```

Now we can define

```
divMod :: Nat -> Nat -> (Nat,Nat)
divMod x y = if x < y then (Zero,x)
             else (Succ q,r)
             where (q,r) = divMod (x-y) y
```

3.7 Chapter notes

The primary source book for computer arithmetic is *The Art of Computer Programming, Volume 2: Semi-numerical Algorithms* (Addison-Wesley, 1998) by Don Knuth. The arithmetic of floors and other simple numerical functions is studied in depth in *Concrete Mathematics* (Addison-Wesley, 1989) by Don Knuth, Ronald Graham and Oren Patashnik.