1 The number systems in Haskell 98

1.0.1 Foreword

Of all more or less popular programming languages, Haskell has the most complicated number system by far. And it is complicated from every perspective, there is no simple angle to start from. It is not very elegant in itself, but powerful and flexible. In this respect, purity and beauty has been sacrificed for the sake of usefulness. It is difficult to understand and difficult to explain, because it is the complex result of many different design paradigms.
But however complicated, it is at least compact and we can summarize everything on one or two pages: figure 1 is the complete listing of all number–related Haskell 98 declarations. As far as the core mathematical aspect of the number system is concerned, figure 2 is a comprehesive summary and should suffice as a reference, once the picture is explained and understood. And for all string conversions of numeral representations, there is a separate part, summarized in figure 3.
So understanding Haskells number system is no more than understanding the pictures 2 and 3. And we introduce into this world by stepwise building up these hierarchies.

1.0.2 Remark

There are two established ways to look at numbers:

- In the *mathematical tradition*, there is a hierarchy, an evolution

  $$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$$

  from NATURAL NUMBERS, INTEGERS, RATIONAL NUMBERS, REAL NUMBERS, to COMPLEX NUMBERS. In this sequence, each number system emerges by overcoming certain operational limitations of the predecessor system. The whole is a beautiful and elegant achievement, shaped in the 19th century and a standard part of scientific culture ever since.
- More recent is the *computer science tradition*. In the first place, this is making computers do what the mathematical tradition has taught us. But that dit not come without certain sacrifices in accuracy and number size. In the computer language C for example, we have types like `int` for integers, and `float` and `double` for real numbers. But different to the mathematical number systems, these types are defined by *machine words*: `int` numbers are stored in 2 or 4 bytes, depending on the actual implementation, each `float` is made of 4 byte and `double` comprises 8 byte words (hence the title: "double" is "double size float"). And when the actual numbers become too big or too small for these limitations, things are rounded. Strictly speaking, that violates destroys the whole mathematical design. Of course, these inaccuracies can be precisely determined, there are established standardizations by now, the result is just another kind of mathematical theory. But the point is, that this is a different kind of thinking, nevertheless.

For a real understanding of Haskells number concept, we need to be aware of these two traditions, because they are both explicitly present. There is $\mathbb{Z}$ and $\mathbb{Q}$ in their full potentials (called `Integer` and `Rational` in Haskell), but `int`, `float` and `double` from C are reborn in Haskell as well (only with capital initials: `Int`, `Float` and `Double`).

1.0.3 Introduction

Anyway, let us start all over again. Our goal is the stepwise (re)construction of figure 2. And we take off in the middle.

\newpage

1.1 The four sorts of numbers

1.1.1 Definition — the four sorts of numbers

There are four "sorts" of numbers in Haskell:

`Integral` `RealFloat` `Ratio` `Complex`

(1) `Integral` numbers are the Haskell version of the *integers* $\mathbb{Z}$ in mathematics. As usual, the default representation is by *decimal numerals* with an optional negation symbol, as in

`123456789` `0` `-77`

But it is also possible to use *octal numerals* (with a `0o` prefix; e.g. `0o123` denotes the integer `83`) and *hexadecimal numerals* (with a `0x` prefix; e.g. `0x123` stands for `291`).

(2) `RealFloat` numbers are the Haskell name for what is commonly called FLOATING–POINT NUMBERS. As usual, there is the dot notation with optional `e` or `E` exponent, e.g.

`12.34` `-12.34e56` `1234e-56` `0.0` `10E1234`

Floating point numbers approximate the *real numbers* $\mathbb{R}$, but can only cope with a certain accuracy.

(3) `Ratio` numbers are the Haskell version of the *rational numbers* $\mathbb{Q}$. Recall, that the standard mathematical notation of a $\mathbb{Q}$ element is

$$\frac{n}{d} \quad \text{with} \quad n, d \in \mathbb{Z} \text{ and } d \neq 0$$

Due to the layout restrictions of a programming language, this has become

`n % d` with `n` and `d` being `Integral` numbers

in Haskell.

(4) `Complex` numbers are the Haskell version of the standard *complex number* system $\mathbb{C}$ in mathematics. The default representation of such a number is a pair

$$\langle x, y \rangle \quad \text{or} \quad x + y\imath \quad \text{with} \quad x, y \in \mathbb{R}$$

($\imath$ being the *imaginary unit* with $\imath^2$ = -1) In Haskell notation, this is written

`x :+ y` with x and y being `RealFloat` numbers

***** picture 4 shows the syntax of Integer and Float literals, as in the Haskell Report; but that is probably too much information *****

1.1.2 Remark

(1) The four names `Integral`, `RealFloat`, `Ratio` and `Complex` are Hakell keywords, but they are no proper types as such. For example, we cannot say "`123456::Integral`" or "`123.456::RealFloat`", that is no legal Haskell code. Of course, Haskell has *types* and *type classes*, but no *sorts*. Nevertheless, let us continue with our four "sorts" for now.

(2) The contructors `%` for `Ratio` and `:+` for `Complex` numbers may have optional spaces around them. [1]

(3) In general, the *number* notion may refer to both, a kind of platonic value or a syntactic sequence of symbols. But if one specifically refers to the latter, i.e. the syntactical representation, one often uses the term *numeral* instead.[2] For example, the decimal numeral `100`, the octal numeral `0o144` and the hexadicimal numeral `0x64` all denote the same number.

(4) Are these four number sorts distinct? Well, "yes" and "no", the full answer is complicated and has to wait after the introduction of the numeric type classes.[3] But the short answer is a "yes", we can use an `Integral` numeral like `1234` for any of the other three sorts as well.

[1] There doesn't seem to be a real standard in this respect. For example, "`12%34`" (without spaces) and "`12 :+ 34`" (with spaces) is the default layout in GHC. But Hugs outputs "`12 % 34`" instead.

[2] In the Haskell Report, a numeral is called a *numeric literal*.

[3] ML has a similar type system, and there, `int` and `real` numbers are really distinct. The numeral `0` is of type `int` and one has to write something like `0.0` to refer to zero in `real`. To migrate from one type to the other, one has to use explicit type converter functions.

\newpage

1.1.3 Example — input of numerals

Let us input some numbers in a GHCi session.[4] We call `ghci` from the shell and its prompt invites for input. By default, this prompt is `Prelude>` .

(1) `Integral` numbers
in default decimal notation are basic values in the sense that they are not evaluated any further

```
Prelude> 1234
1234
```

Note, that it is possible in Haskell to deal with `Integral` numbers of arbitrary size

```
Prelude> 1234567890123456789012345678901234456789012345678901234567890
1234567890123456789012345678901234456789012345678901234567890
```

Hexadicimal numerals (with a `0x`, "0" is zero) and octal numerals are converted into the default decimal representation

```
Prelude> 0x1234
4660
Prelude> 0o1234
668
```

Recall, that these conversions are computed by

$$\begin{array}{lclcl} \texttt{0x1234} & = & 4\cdot 16^0 + 3\cdot 16^1 + 2\cdot 16^2 + 1\cdot 16^3 & = & 4660 \\ \texttt{0o1234} & = & 4\cdot 8^0 + 3\cdot 8^1 + 2\cdot 8^2 + 1\cdot 8^3 & = & 668 \end{array}$$

(2) `RealFloat` numbers.
One common representation is the decimal dot notation. But note, that the accuracy is limited and all too long numbers are shortened.

```
Prelude> 12.34
12.34
Prelude> 0.0
0.0
Prelude> 12.345678901234567890123456789O
12.345678901234567
Prelude> 7.77777777777777777777777777777777777777777777777777777777
7.7777777777777778
Prelude> 3.33333333333333333333333333333333333333333333333333333333
3.3333333333333335
```

Also, there is the notation with the "e" or "E". Recall, that "nEm" or "nem" stands for $n\cdot 10^m$ (with $10^{-m} = \frac{1}{10^m}$)

```
Prelude> 12.34e0
12.34
Prelude> 12.34e1
123.4
Prelude> 12.34e-1
1.234
```

And as usual, the default representation is with one digit preceding the dot:

```
Prelude> -12.34e56
-1.234e57
Prelude> 1234e-56
1.234e-53
```

But again, the accuracy is limited:

```
Prelude> 10E-1234
0.0
```

and so is the size of `RealFloat` numbers:

```
Prelude> 10E1234
Infinity
```

(3) `Ratio` numbers
are implemented in the standard `Ratio` module. So we need to make its entities available first. Depending on the interpreter, there are several ways to load `Ratio`. In the GHC interpreter we use the `:module` or `:m` command.

```
Prelude> :module Ratio
Prelude Ratio>
```

The changed prompt indicates a successful loading. Example input is always changed to the unique reduced form (with positive donomiator and no common devisor in nominator and denomiator).

```
Prelude Ratio> -7 % 5
(-7)%5
Prelude Ratio> 7 % (-5)                    {- mind the parentheses! -}
-7%5
Prelude Ratio> -35%25
(-7)%5
```

Made of two `Integral` numbers, `Ratio` numbers don't suffer from any limits in size

```
Prelude Ratio> 7 % 1234567890123456789012345678901234567890
7%1234567890123456789012345678901234567890
```

Zero denominators are refused, as in other programming languages

```
Prelude Ratio> 123%0
*** Exception:  Ratio.%:  zero denominator
```

(4) `Complex` numbers
are implemented in the standard `Complex` module. Again, we have its entities available after calling the `:module` or `:m` command.

```
Prelude> :module Complex
Prelude Complex>
```

Any pair `x`, `y` of `RealFloat` numbers makes a complex number `x :+ y`, where `x` is the real and `y` is the imaginary part.[5]

```
Prelude Complex> 0.123 :+ 123.0
0.123 :+ 123.0
Prelude Complex> (-1234.56e-3) :+ (-222.22)      {- parentheses! -}
(-1.23456) :+ (-222.22)
```

In this context, every `Integral` `x` or `y` is accepted as a `RealFloat`

```
Prelude Complex> 1 :+ 1
1.0 :+ 1.0
```

Being pairs of `RealFloat` numbers, `Complex` numbers suffer from the same limitations in size and accuracy.

```
Prelude Complex> 1e1000 :+ 1e1000
Infinity :+ Infinity
Prelude Complex> 1e-1000 :+ 1e-1000
0.0 :+ 0.0
```

[4] GHC is the Glasgow Haskell Compiler suite and GHCi is its interactive/interpreter program.

[5] For a repetition of the complex number system, see ???? below.

\newpage

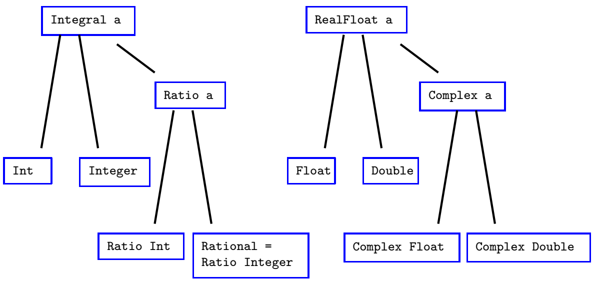
1.2 The eight standard number types

1.2.1

Let us take the next step towards the hierachy of figure 2. In definition 1.1.1, we started with our four "sorts" of numbers:

`Integral` `RealFloat` `Ratio` `Complex`

Let us now get down to the proper number types in Haskell. It turns out, that the two primitive sorts `Integral` and `RealFloat` each split into two different types. And since each `Ratio` number is a composition of two `Integral` numbers, we have two types for `Ratio` as well. Similarly for `Complex` numbers, which are pairs of `RealFloat` numbers. So alltogether, our four sorts split into eight proper Haskell types and these are the standard number types. In the end, we have a new picture



But let us introduce the types for each sort at a time.

1.2.2 Definition the standard Integral number types

There are two `Integral` data types:

(a) `data Int = minBound ... -1 | 0 | 1 ... maxBound`
Fixed sized integers `Int`, ranging from `minBound` to `maxBound`, depending on the implementation. `Int` is very similar to the `int` type from C.

(b) `data Integer = ... -1 | 0 | 1 ...`
Integers of arbitrary size.

`Integral` itself is a type class

```
class Integral a where ... {- defined later on -}
```

and thus has two instances

```
instance Integral Int     where ...
instance Integral Integer where ...
```

1.2.3 Remark

(1) From a purely functional point of view, this duality of types is absurd. `Integer` comprises all the members of `Int` and is safer, because it doesn't interrupt or misbehaves due to unexpected overflows. But of course, `Int` is introduced into the language because it enables the use of built–in processor arithmetic, which is way faster.[6] All "syntactical" operations in Haskell that involve numbers also use `Int` instead of `Integer`. For example,

```
length :: [a] -> Int    or    (!!) :: [a] -> Int -> a
```

So if speed is not totally irrelevant and the values are certain to stay in a reasonable range, then `Int` should be the first choice.

(2) The actual bounds of `Int` are depending on the implementation. But the Haskell Report demands at least

minBound $\leq -2^{29} =$ -536870912

maxBound $\geq 2^{29}-1 =$ 536870911

For example, on my own system (Debian Linux on an Intel Pentium Dual CPU) and with the GHC interpreter (version 6.8.2) I obtain[7]

```
> minBound ::  Int
-2147483648
> maxBound ::  Int
2147483647
```

1.2.4 Remark

If you need to write a program that involves `Integral` numbers, you may know in advance which type suits you more: either `Int` for fast functions and compatibility with the list function arguments or `Integer` for real large numbers. And in that case, you can fix the type everywhere by adding a type declaration to every definition; which is good programming style anyway.
For example, suppose we need a simple `triple` function, where say `triple 5` is `15`. If we know in advance, that we only operate on small `Integral` numbers, we should use this version

```
triple :: Int -> Int
triple n = 3 * n
```

However, if we need the real integers $\mathbb{Z}$ without any limits, we may rather use

```
triple :: Integer -> Integer
triple n = 3 * n
```

But note, that once the type is fixed, all values and results are bound to that type and type mixes lead to error messages, even if all types are `Integral`. For example, both the following inputs are fine:

```
> let { x = 5 ::  Int ; y = 6 ::  Int } in x + y
11 ::  Int
> let { x = 5 ::  Integer ; y = 6 ::  Integer } in x + y
11 ::  Integer
```

but this won't work and produces an error message

```
> let { x = 5 ::  Integer ; y = 6 ::  Int } in x + y
..... error ......
```

1.2.5 Definition the standard RealFloat number types

There are two `RealFloat` data types

(a) `data Float = ...`
Single precision floating point numbers, with a range depending on the implementation, very similar to the `float` type in C.

(b) `data Double = ...`
Double precision floating point numbers, with a range depending on the implementation, very similar to the `double` type in C.

`RealFloat` itself is a type class

```
class RealFloat a where ... {- defined later on -}
```

and thus has two instances

```
instance RealFloat Float  where ...
instance RealFloat Double where ...
```

[6] However, see also exercise 1.2.12, showing real Haskell systems may show some unexpected behavior in this respect.

[7] `minBound::(Bounded a) => a` is a class member of the `Bounded` class and just asking the interpreter for `minBound` itself, without the type constraint `minBound::Int`, does interrupt with an "unresolved overloading" message.

1.2.6 Example

The following session gives an impression of the difference between `Float` and `Double`.

```
> 1.23456789012345678901234567890 :: Float
1.234568 :: Float
> 1.23456789012345678901234567890 :: Double
1.23456789012346 :: Double
```

1.2.7 Definition — the standard `Ratio` number types

`Ratio` is made of `Integral` number pairs, its definition is a parameterized data type

```
data (Integral a) => Ratio a = a%a
```

And with `Integral` comprising two standard types, `Ratio` has two standard types as well:

(a) `Ratio Int`
number pairs `x%y` with `x` and `y` in the range of `Int`.

(b) `Ratio Integer`
rational numbers `x%y` of with `x` and `y` of arbitrary size. This type is provided with an own name by the type declaration

```
type Rational = Ratio Integer
```

(Note, that the `Ratio` module has to be imported/loaded in order to make full use of `Ratio` numbers.)

1.2.8 Example

Both, the numerator `n` and denumerator `d` in `n%d` have to be of the same type. In Hugs (and similar for the GHC interpreter) we have

```
Hugs> :load Ratio                    {- or :module Ratio to import the module -}
Ratio> (123 :: Int) % (456 :: Int)
41 % 152 :: Ratio Int
Ratio> (123 :: Integer) % (456 :: Integer)
41 % 152 :: Ratio Integer
Ratio> (123 :: Integer) % (456 :: Int)
ERROR - Type error in application ...
```

Of course, instead of typing each component with say

```
Ratio> (123 :: Integer) % (456 :: Integer)
41 % 152 :: Ratio Integer
```

we may as well type it like this

```
Ratio> 123 % 456 :: Ratio Integer
41 % 152 :: Ratio Integer
```

which is of course just type synonym for

```
Ratio> 123 % 456 :: Rational
41 % 152 :: Rational
```

1.2.9 Definition — the standard `Complex` number types

`Complex` is made of `RealFloat` number pairs, its definition is a parameterized data type

```
data (RealFloat a) => Complex a = a :+ a
```

And with `RealFloat` comprising two standard types, `Complex` has two standard types as well:

(a) `Complex Float`

(b) `Complex Double`

(Note, that the `Complex` module has to be imported/loaded.)

1.2.10 Example

To demonstrate the difference between the two standard `Complex` types, consider the following session (with Hugs or GHC):

```
> :module Complex                    {- don't forget to :module or :load -}
> 1.23456789012345678890:+0.9876543210987654321::Complex Float
1.234568 :+ 0.9876543 :: Complex Float
> 1.23456789012345678890:+0.9876543210987654321::Complex Double
1.23456789012346 :+ 0.987654321098765 :: Complex Double
```

1.2.11 Remark

Note the conservative choice of the names for the standard types:

- it preserves the legacy of C and its successor language:

type in C	same type in Haskell
`int`	`Int`
`float`	`Float`
`double`	`Double`

- At least two types have the full potential of their counterparts in mathematics

number system in mathematics	same type in Haskell
the integers $\mathbb{Z}$	`Integer`
the rational numbers $\mathbb{Q}$	`Rational`

Also note the difference between the four sorts again:

- The composed sorts `(Ratio a)` and `(Complex a)` are *data types*, although with a parameter type `a`. These data types are well–defined by now.
- The primitive sorts `(Integral a)` and `(RealFloat a)` are actually more complicated *type classes* and their proper definition is still to come.

So by now we really need to turn from the lower *type* part of figure 2 to upper *type class* part.

\newpage

1.2.12 Exercise

Suppose, we have two versions of a simple addition of positive `Integral` numbers defined in identical fashion, but for the two `Integral` types:

```
addInt :: Int -> Int -> Int
addInt n m = if n < 0 then m else addInt (n-1) (m+1)

addInteger :: Integer -> Integer -> Integer
addInteger n m = if n < 0 then m else addInteger (n-1) (m+1)
```

In the Hugs and GHC interpreter, we can ask for additional information about the speed and memory for each call with the "`:set +s`" command. Hugs answers with "reductions and cells" and GHC returns information about "seconds and bytes". We can also enforce a type expression with each value output with the "`:set +t`". An example dialog with Hugs is then given by

```
> addInt 12345 12345
24691 :: Int
(209915 reductions, 281951 cells)
> addInteger 12345 12345
24691 :: Integer
(209915 reductions, 358082 cells)
```

Thus a (small) space advantage for the `Int` version, as expected. But the GHC interpreter shows the following behavior:

```
> addInteger 123456 123456
246913 :: Integer                 {- actually, this output line is two lines -}
(0.36 secs, 22699272 bytes)
> addInt 123456 123456
246913 :: Int                     {- again, the actual output comes in two lines -}
(0.38 secs, 22696400 bytes)
```

Strangely and against all earlier reasoning, the `Integer` version is faster! [8]

[8] I don't have an explanation for this behavior. If this is an example of a general pattern, it would contradict the whole justification for the distinction between the two `Integral` types so far.

\newpage

1.3 Type classes and type class instances

1.3.1 Introduction

Modern mathematics emerged with the understanding, that values such as numbers cannot be described appropriately as such, but only as elements of a *structure* or *algebra*[9]. It also turned out, that even a structure can hardly be defined as such. Instead, it is often described as a *model* or *instance* of a *theory*. An algebraic theory only specifies the *signature* of the algebra and a couple of rules, called *axioms*, that have to hold.
For example, the *theory of rings* defines a *ring* as a set R together with two constants 0 and 1 and functions $+, -, \cdot : R \times R \longrightarrow R$, such that a couple of axioms have to hold (such as the associativity of $+$ and $\cdot$, etc.). Many different, but important algebras are rings, i.e. they are *models* of this theory, in particular the integers together with their usual arithmetic operations.
Haskell has adopted this design, but here, theories are called *type classes* and their models are called *instances*. For example, there is a type class called `Num` and all eight standard numeric classes like `Int`, `Rational` or `Complex Float` are instances of `Num`.

1.3.2 Definition

A TYPE CLASS comprises

(a) A *class name*, which is a Haskell identifier with an initial capital letter (e.g. `Eq`, `Ord`, `Show`)

(b) A *signature*, comprising

- a distinguished, but arbitrary type variable, say `a`
- some type declarations for values (constants, functions), based on already well–defined types and `a`

(c) Some *axioms* or *rules*, i.e. statements about the values that have to be satisfied.

The actual syntax for a type class definition of name `C` on type variable `a` is

```
class C a where
  {- declarations -}
  {- axioms -}
```

This may be read as "type `a` is of class `C`, if the following declarations are defined and axioms are satisfied".

Suppose, a type class named `C` is defined as just described. An INSTANCE of `C` is an actual type `T` that takes the place of the variable `a`. In general, this instantiation is done with an `instance` declaration and an explicit definition for all value declarations in the definition of `C`. The general form to do that is

```
instance C T where
  {- value definitions -}
```

For some type classes however, there is a shortcut. When `T` is defined as a `data` or `newtype`, Haskell can DERIVE all the value definitions for `T` by putting a `deriving` statement immediately after the definition of `T`:

```
data T = ...
  deriving C
```

But this only works for some standard Haskell type classes (namely `Eq`, `Ord`, `Enum`, `Bounded`, `Read`, and `Show`). In all other situations, all the values have to be defined explicitly in the instance declaration.

1.3.3 the Eq type class

The Haskell `Prelude`[10] contains the following class definition

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y  =  not (x == y)
  x == y  =  not (x /= y)
```

This is saying, that a type `a` can only be an `Eq` type, if there are two functions defined on `a`, namely

`(==) :: a -> a -> Bool` and `(/=) :: a -> a -> Bool`

so that all `x`, `y` of type `a` satisfy the two axioms

$$\texttt{x /= y} = \begin{cases} \texttt{True} & \text{if } \texttt{x == y} \text{ is } \texttt{False} \\ \texttt{False} & \text{if } \texttt{x == y} \text{ is } \texttt{True} \end{cases}$$

$$\texttt{x == y} = \begin{cases} \texttt{True} & \text{if } \texttt{x /= y} \text{ is } \texttt{False} \\ \texttt{False} & \text{if } \texttt{x /= y} \text{ is } \texttt{True} \end{cases}$$

---

[9] *** maybe obsolete *** Many text books in mathematical logic and abstract algebra define a *structure* as a couple of carrier sets, together with a couple of functions and relations, defined on these sets. An *algebraic structure* or *algebra* is then a structure that has exactly one carrier set and no relations, but only functions (including constants, which are subsumed as nullary functions). In Haskell, there are no relations anyway. There, a relation say $R : X \rightsquigarrow Y$ takes the form $R : X \times Y \longrightarrow \mathbf{Bool}$, or more often the curryfied version $R : X \longrightarrow Y \longrightarrow \mathbf{Bool}$.

[10] see *The Haskell Report, 6.3.1*

\newpage

1.3.4 Example — an example Eq instance

Suppose, we define a type `Binum` (for *binary numerals*[11]) a the data type

```
data Binum = BIN [Bool]
```

Note, that Haskell has no default operation to test the equality of two values of the same data type. An input like

```
> BIN [True,False] == BIN [True,False]
```

doesn't answer with `True` or `False`, but with an error message. We really have to define the equality.
The easy way to do this is the `deriving` statement immediately after the data type declaration:

```
data Binum = BIN [Bool] deriving Eq
```

If we do that, then the two functions `(==)` and `(/=)` are available on `Binum` values as well, and we e.g. obtain as expected

```
> BIN [True,False] == BIN [True,False]
True
```

At this point and for this example data type, the explicit declaration of a "trivial" function like the equality `(==)` may seem an unnecessary burden on the programmer. But a default equality definition would rather restrict our freedom and needs in many other cases. For example, consider two `Rational` numbers. We will certainly expect Haskell to do the following

```
> 6%8 == 3%4
True
```

because $\frac{6}{8} = \frac{3}{4}$. But a default equality would find the two values to be different. The same goes for two `Float` numbers, say `12.34e2` and `1.234E3`, which ought to be equal.
Anyway, we can also turn `Binum` into an `Eq` instance by means of an explicit `instance` declaration. For example, by

```
instance Eq NatBinum where
  BIN []         == BIN []          =  True
  BIN []         == BIN _           =  False
  BIN _          == BIN []          =  False
  BIN (False:xL) == BIN (True:_)    =  False
  BIN (True:xL)  == BIN (False:yL)  =  False
  BIN (False:xL) == BIN (False:yL)  =  BIN xL == BIN yL
  BIN (True:xL)  == BIN (True:yL)   =  BIN xL == BIN yL
```

Note, that we only need to define `(==)`, because the second function `(/=)` of `Eq` is fully specified by the axioms, i.e. the class definition of `Eq` itself.

1.3.5 Example — standard Eq instances

***** CONTINUE HERE *****

1.3.6 Remark

- ***** type classes are algebraic theories, instances are models or algebras *****
- ***** multiple type classes *****
- ***** instances can only be defined by data or newtype *****
- ***** alternative to the instance declaration *****

1.3.7 — the Num type class

***** CONTINUE HERE *****

***** CONTINUE HERE *****

[11] A *binary numeral* has the form $\beta_n \ldots \beta_2\beta_1\beta_0$, where each β is either 0 or 1, and where $\beta_n \ldots \beta_2\beta_1\beta_0$ itself denotes the number $\beta_n \cdot 2^n + \beta_{n\text{-}1} \cdot 2^{n\text{-}1} + \ldots + \beta_2 \cdot 2^2 + \beta_1 \cdot 2^1 + \beta_0 \cdot 2^0$. For example the binary numeral 1101 is the (decimal) number 13. In our `Binum` implementation of binary numerals, 0 and 1 are represented by `False` and `True`, respectively.

\newpage

1.3.8 Introduction

Maybe the best way to understand type classes is a repetition of some basic concept of modern algebra. What has become a *type class* in Haskell is very much what is traditionally called an *algebraic theory*. The *instances* of type classes are very much the *models* in mathematical jargon.

1.3.9 Definition

An ALGEBRAIC STRUCTURE or ALGEBRA $\mathfrak{A}$ is given by

$$\mathfrak{A} = \langle A, c_1, \ldots, c_n, f_1, \ldots, f_m \rangle$$

where

- A is the CARRIER set
- the CONSTANTS $c_1, \ldots, c_n$ are distinguished elements of A
- each of the FUNCTIONS $f_1, \ldots, f_m$ is an ordinary function on A, i.e. $f_i : A \times \ldots \times A \longrightarrow A$

1.3.10 Example

(1) An algebra is given by $\langle \mathbb{Z}, 0, 1, +, \cdot \rangle$, the integers $\mathbb{Z}$ are the carrier set, there are two constants 0 and 1 and two binary functions $+ : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ and $\cdot : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$.

(2) $\langle \mathbb{Z}, 0, +, \mathbf{negate} \rangle$ is also an algebra, where $\mathbb{Z}, 0$ and $+$ are as in (1), and $\mathbf{negate} : \mathbb{Z} \longrightarrow \mathbb{Z}$ is the unary function that turns each $n \in \mathbb{Z}$ into -n.

(3) $\langle \mathbb{Z}, \leq \rangle$ is not an algebra, because in mathematics $\leq$ is not a function, but a relation[12]

1.3.11 Definition

An ALGEBRAIC THEORY $\mathfrak{T}$ comprises

- an (algebraic) SIGNATURE, made of symbols $\mathbf{A}, \mathbf{c_1}, \ldots, \mathbf{c_n}, \mathbf{f_1}, \ldots, \mathbf{f_m}$, where
  - $\mathbf{A}$ is a carrier set symbol
  - the $\mathbf{c_1}, \ldots, \mathbf{cn}$ are symbols for constants
  - the $\mathbf{f_1}, \ldots, \mathbf{f_m}$ are symbols for functions

  and where each function symbol has an own type expression of the form

  $$\mathbf{f_i} : A \times \ldots \times A \longrightarrow A$$

- A set of AXIOMS, which are formulas on the given signature, making certain statements that are supposed to hold.

1.3.12 Example

Standard examples of algebraic theories are the following.

- The theory of monoids.

  The signature: A *monoid* has the form $\langle \mathbf{M}, \mathbf{e}, \circ \rangle$, where $\mathbf{M}$ stands for a carrier set, $\mathbf{e}$ is a constant of $\mathbf{M}$, called the *neutral element*, and $\circ : \mathbf{M} \times \mathbf{M} \longrightarrow \mathbf{M}$ is a binary function, usually written in infix notation.

  The axioms:

  The associativity axiom: $\forall x, y, z \in M \,.\, (x \circ y) \circ z = x \circ (y \circ z)$

  The neutral element axiom: $\forall x \in M \,.\, x \circ \mathbf{e} = \mathbf{e} \circ x = x$.

- The theory of groups.

  A *group* has the form ***** CONTINUE HERE *****

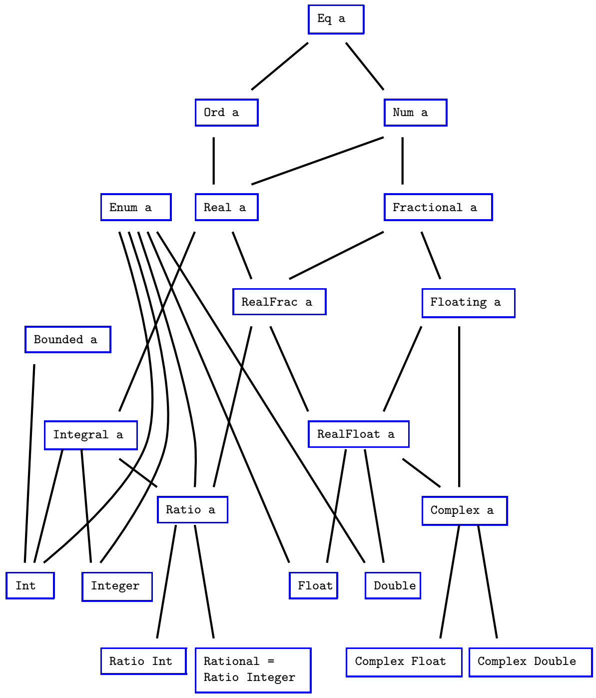
[12] A *structure* in general is made of carriers and constants, functions and relations on these carrier classes. An *algebraic structure* in particular is a structure with exactly one carrier class and no relations other than equality.

\newpage

***** At this point, the real introduction of all numeric functions and type classes really takes off. But the available text so far is not in a decent shape. The following is then the conclusion: *****

1.3.13 the standard numeric type classes and types

The whole hierachy of type classes and types make a complex picture.



The two type classes `Show` and `Read` are excluded for now (see ???? below). They don't belong to the mathematical aspect of the number system, but deal with the string conversion of numeral representation.

***** CONTINUE HERE *****

Figure 1: The number system of Haskell 98, as given in the `Prelude`, `Ratio`, `Complex` and `Numeric` modules

(1) Type classes from the `Prelude`

(a) Equality and Ordered classes

```
class  Eq a  where
    (==), (/=) :: a -> a -> Bool
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

(b) Enumeration and Bounded classes

```
class  Enum a  where
    succ, pred     :: a -> a
    toEnum         :: Int -> a
    fromEnum       :: a -> Int
    enumFrom       :: a -> [a]                -- [n..]
    enumFromThen   :: a -> a -> [a]           -- [n,n'..]
    enumFromTo     :: a -> a -> [a]           -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]      -- [n,n'..m]
class  Bounded a  where
    minBound       :: a
    maxBound       :: a
```

(c) Numeric classes

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)     :: a -> a -> a
    negate            :: a -> a
    abs, signum       :: a -> a
    fromInteger       :: Integer -> a
class  (Num a, Ord a) => Real a  where
    toRational        ::  a -> Rational
class  (Real a, Enum a) => Integral a  where
    quot, rem         :: a -> a -> a
    div, mod          :: a -> a -> a
    quotRem, divMod   :: a -> a -> (a,a)
    toInteger         :: a -> Integer
class  (Num a) => Fractional a  where
    (/)               :: a -> a -> a
    recip             :: a -> a
    fromRational      :: Rational -> a
class  (Fractional a) => Floating a  where
    pi                   :: a
    exp, log, sqrt       :: a -> a
    (**), logBase        :: a -> a -> a
    sin, cos, tan        :: a -> a
    asin, acos, atan     :: a -> a
    sinh, cosh, tanh     :: a -> a
    asinh, acosh, atanh  :: a -> a
class  (Real a, Fractional a) => RealFrac a  where
    properFraction    :: (Integral b) => a -> (b,a)
    truncate, round   :: (Integral b) => a -> b
    ceiling, floor    :: (Integral b) => a -> b
class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix        :: a -> Integer
    floatDigits       :: a -> Int
    floatRange        :: a -> (Int,Int)
    decodeFloat       :: a -> (Integer,Int)
    encodeFloat       :: Integer -> Int -> a
    exponent          :: a -> Int
    significand       :: a -> a
    scaleFloat        :: Int -> a -> a
    isNaN             :: a -> Bool
    isInfinite        :: a -> Bool
    isDenormalized    :: a -> Bool
    isNegativeZero    :: a -> Bool
    IEEE              :: a -> Bool
    atan2             :: a -> a -> a
```

(2) Types and instance declarations

(a) from `Prelude`

(i) Bound integers

```
data  Int  =  minBound ... -1 | 0 | 1 ... maxBound
instance  Eq       Int  where ...
instance  Ord      Int  where ...
instance  Num      Int  where ...
instance  Real     Int  where ...
instance  Integral Int  where ...
instance  Enum     Int  where ...
instance  Bounded  Int  where ...
instance  Show     Int  where ...
instance  Read     Int  where ...
```

(ii) integers of arbitrary size

```
data  Integer  =  ... -1 | 0 | 1 ...
instance  Eq       Integer  where ...
instance  Ord      Integer  where ...
instance  Num      Integer  where ...
instance  Real     Integer  where ...
instance  Integral Integer  where ...
instance  Enum     Integer  where ...
instance  Show     Integer  where ...
instance  Read     Integer  where ...
```

(iii) single precision floating numbers

```
data  Float  =  ...
instance  Eq         Float  where ...
instance  Ord        Float  where ...
instance  Num        Float  where ...
instance  Real       Float  where ...
instance  Fractional Float  where ...
instance  Floating   Float  where ...
instance  RealFrac   Float  where ...
instance  RealFloat  Float  where ...
instance  Enum       Float  where ...
instance  Show       Float  where ...
instance  Read       Float  where ...
```

(iii) double precision floating numbers

```
data  Double  =  ...
instance  Eq         Double  where ...
instance  Ord        Double  where ...
instance  Num        Double  where ...
instance  Real       Double  where ...
instance  Fractional Double  where ...
instance  Floating   Double  where ...
instance  RealFrac   Double  where ...
instance  RealFloat  Double  where ...
instance  Enum       Double  where ...
instance  Show       Double  where ...
instance  Read       Double  where ...
```

(b) from `Ratio`

(i) The `Ratio a` type

```
data  (Integral a) => Ratio a = ...
instance  (Integral a)         => Eq         (Ratio a)  where ...
instance  (Integral a)         => Ord        (Ratio a)  where ...
instance  (Integral a)         => Num        (Ratio a)  where ...
instance  (Integral a)         => Real       (Ratio a)  where ...
instance  (Integral a)         => Fractional (Ratio a)  where ...
instance  (Integral a)         => RealFrac   (Ratio a)  where ...
instance  (Integral a)         => Enum       (Ratio a)  where ...
instance  (Read a,Integral a)  => Read       (Ratio a)  where ...
instance  (Integral a)         => Show       (Ratio a)  where ...
```

(ii) The `Rational` type

```
type  Rational =  Ratio Integer
```

(c) from `Complex`

(i) The `Complex a` type

```
data  (RealFloat a)  => Complex a = !a :+ !a
instance  (RealFloat a) => Eq         (Complex a)  where ...
instance  (RealFloat a) => Read       (Complex a)  where ...
instance  (RealFloat a) => Show       (Complex a)  where ...
instance  (RealFloat a) => Num        (Complex a)  where ...
instance  (RealFloat a) => Fractional (Complex a)  where ...
instance  (RealFloat a) => Floating   (Complex a)  where ...
```

(3) Numeric functions

(a) from `Prelude`

```
subtract     :: (Num a) => a -> a -> a
even         :: (Integral a) => a -> Bool
odd          :: (Integral a) => a -> Bool
gcd          :: (Integral a) => a -> a -> a
lcm          :: (Integral a) => a -> a -> a
(^)          :: (Num a, Integral b) => a -> b -> a
(^^)         :: (Fractional a, Integral b) => a -> b -> a
fromIntegral :: (Integral a, Num b) => a -> b
realToFrac   :: (Real a, Fractional b) => a -> b
minimum      :: (Ord a) => [a] -> a
maximum      :: (Ord a) => [a] -> a
sum          :: (Num a) => [a] -> a
product      :: (Num a) => [a] -> a
numericEnumFrom       :: (Fractional a) => a -> [a]
numericEnumFromThen   :: (Fractional a) => a -> a -> [a]
numericEnumFromTo     :: (Fractional a, Ord a) => a -> a -> [a]
numericEnumFromThenTo :: (Fractional a, Ord a) => a -> a -> a -> [a]
```

(b) from `Ratio`

```
(%)            :: (Integral a) => a -> a -> Ratio a
numerator      :: (Integral a) => Ratio a -> a
denominator    :: (Integral a) => Ratio a -> a
approxRational :: (RealFrac a) => a -> a -> Rational
```

(c) from `Complex`

```
realPart   :: (RealFloat a) => Complex a -> a
imagPart   :: (RealFloat a) => Complex a -> a
conjugate  :: (RealFloat a) => Complex a -> Complex a
mkPolar    :: (RealFloat a) => a -> a -> Complex a
cis        :: (RealFloat a) => a -> Complex a
polar      :: (RealFloat a) => Complex a -> (a,a)
magnitude  :: (RealFloat a) => Complex a -> a
phase      :: (RealFloat a) => Complex a -> a
```

(d) from `Numeric`

```
fromRat        :: (RealFloat a) => Rational -> a
showSigned     :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS
showIntAtBase  :: Integral a => a -> (Int -> Char) -> a -> ShowS
showInt        :: Integral a => a -> ShowS
showOct        :: Integral a => a -> ShowS
showHex        :: Integral a => a -> ShowS
readSigned     :: (Real a) => ReadS a -> ReadS a
readInt        :: (Integral a) => a->(Char->Bool)->(Char->Int)->ReadS a
readDec        :: (Integral a) => ReadS a
readOct        :: (Integral a) => ReadS a
readHex        :: (Integral a) => ReadS a
showEFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showGFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFloat      :: (RealFloat a) => a -> ShowS
floatToDigits  :: (RealFloat a) => Integer -> a -> ([Int], Int)
readFloat      :: (RealFrac a) => ReadS a
lexDigits      :: ReadS String
```

(4) operator preferences of all the mentioned operators

```
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`, %
infixl 6  +, -, :+
infix  4  ==, /=, <, <=, >=, >
```

Figure 2: The number system of Haskell 98

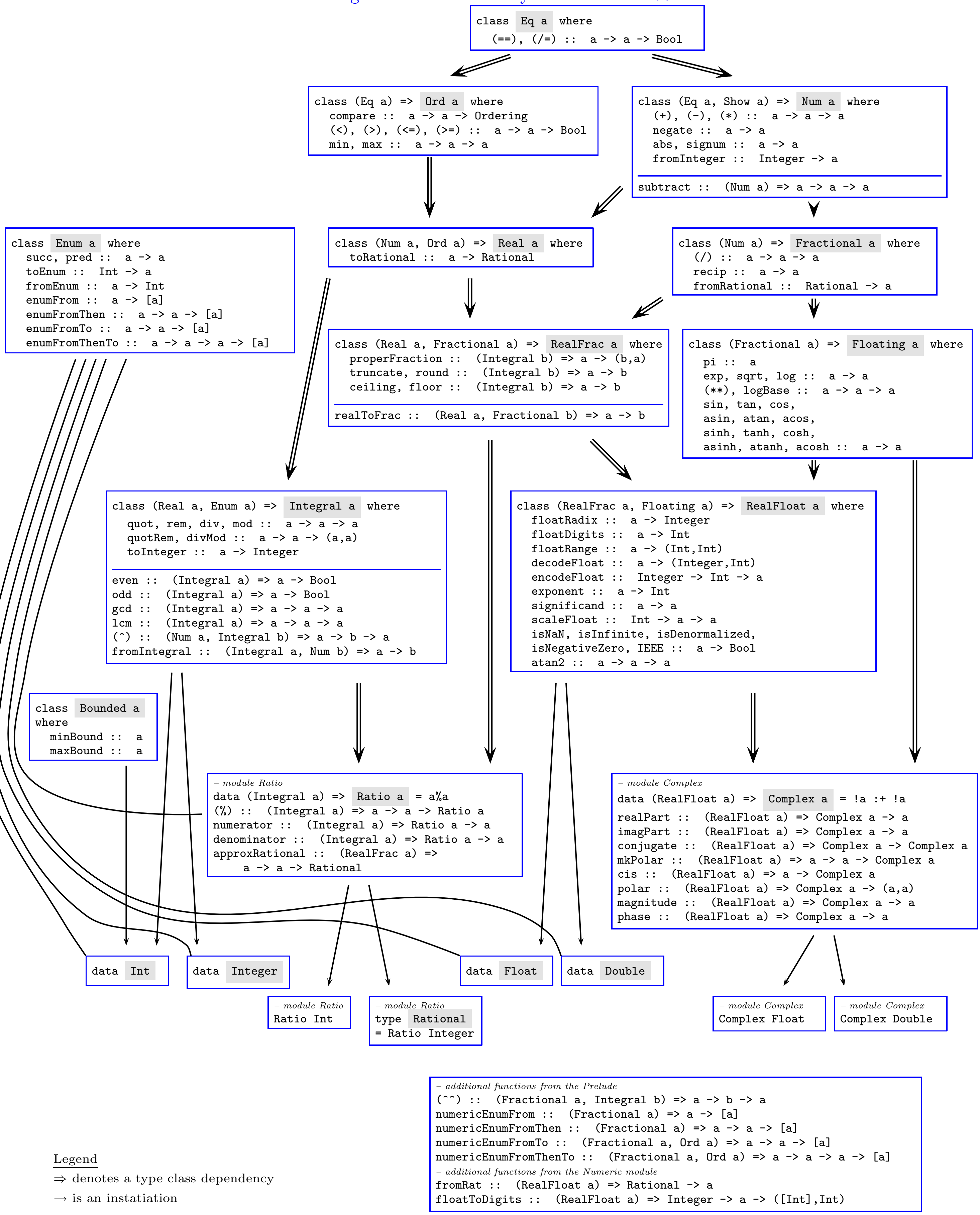


Figure 3: The number and string conversions of Haskell 98

***** this picture is not properly drawn, yet, but all items should be mentioned *****

```
type ShowS = String -> String

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS

shows :: (Show a) => a -> ShowS
showChar :: Char -> ShowS
showString :: String -> ShowS
showParen :: Bool -> ShowS -> ShowS
```

```
type ReadS a = String -> [(a,String)]

class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]

reads :: (Read a) => ReadS a
read :: (Read a) => String -> a
readParen :: Bool -> ReadS a -> ReadS a
lex :: ReadS String
```

```
instance Show Int where ...
instance Read Int where ...
instance Show Integer where ...
instance Read Integer where ...
instance Show Float where ...
instance Read Float where ...
instance Show Double where ...
instance Read Double where ...
instance Show () where ...
instance Read () where ...
instance Show Char where ...
instance Read Char where ...
instance (Read a) => Read [a] where ...
instance (Show a, Show b) => Show (a,b) where ...
instance (Read a, Read b) => Read (a,b) where ...
```

```
– module Numeric
showSigned :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS
showIntAtBase :: Integral a => a -> (Int -> Char) -> a -> ShowS
showInt :: Integral a => a -> ShowS
showOct :: Integral a => a -> ShowS
showHex :: Integral a => a -> ShowS
readSigned :: (Real a) => ReadS a -> ReadS a
readInt :: (Integral a) => a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readDec :: (Integral a) => ReadS a
readOct :: (Integral a) => ReadS a
readHex :: (Integral a) => ReadS a
showEFloat :: (RealFloat a) => Maybe Int -> a -> ShowS
showFFloat :: (RealFloat a) => Maybe Int -> a -> ShowS
showGFloat :: (RealFloat a) => Maybe Int -> a -> ShowS
showFloat :: (RealFloat a) => a -> ShowS
readFloat :: (RealFrac a) => ReadS a
lexDigits :: ReadS String
```

Figure 4: The syntax for integers and floating point numbers

An INTEGER LITERAL has the form *integer*, which is defined by the following grammar:

digit	→	*ascDigit* | *uniDigit*
ascDigit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
uniDigit	→	any Unicode decimal digit
octit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
hexit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`
decimal	→	*digit* { *digit* }
octal	→	*octit* { *octid* }
hexadecimal	→	*hexit* { *hexit* }
integer	→	*decimal*
	|	`0o` *octal* | `0O` *octal*
	|	`0x` *hexadecimal* | `0X` *hexadecimal*

An FLOAT LITERAL has the form *float*, which is defined by the following grammar:

digit	→	*ascDigit* | *uniDigit*
ascDigit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
uniDigit	→	any Unicode decimal digit
decimal	→	*digit* { *digit* }
float	→	*decimal* `.` *decimal* [*exponent*]
	|	*decimal* *exponent*
exponent	→	(`e` | `E`) [`+` | `-`] *decimal*