JFP **13** (1): 157–168, January 2003.
DOI: 10.1017/S0956796803001618 Printed in the United Kingdom

Chapter 14

Numeric Functions

```
module Numeric(fromRat,
               showSigned, showIntAtBase,
               showInt, showOct, showHex,
               readSigned, readInt,
               readDec, readOct, readHex,
               floatToDigits,
               showEFloat, showFFloat, showGFloat, showFloat,
               readFloat, lexDigits) where

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
```

This library contains assorted numeric functions, many of which are used in the standard Prelude.

In what follows, recall the following type definitions from the `Prelude`:

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

14.1 Showing Functions

- `showSigned :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS` converts a possibly-negative `Real` value of type `a` into a string. In the call (`showSigned` *show prec val*), *val* is the value to show, *prec* is the precedence of the enclosing context, and *show* is a function that can show unsigned values.

- `showIntAtBase :: Integral a => a -> (Int -> Char) -> a -> ShowS` shows a *non-negative* `Integral` number using the base specified by the first argument, and the character representation specified by the second.

- `showInt, showOct, showHex :: Integral a => a -> ShowS` show *non-negative* `Integral` numbers in base 10, 8 and 16, respectively.

- `showFFloat, showEFloat, showGFloat`
  `:: (RealFloat a) => Maybe Int -> a -> ShowS` These three functions all show signed `RealFloat` values:

  - `showFFloat` uses standard decimal notation (e.g. `245000`, `0.0015`).
  - `showEFloat` uses scientific (exponential) notation (e.g. `2.45e2`, `1.5e-3`).
  - `showGFloat` uses standard decimal notation for arguments whose absolute value lies between `0.1` and `9,999,999`, and scientific notation otherwise.

  In the call (`showEFloat` *digs val*), if *digs* is `Nothing`, the value is shown to full precision; if *digs* is `Just` *d*, then at most *d* digits after the decimal point are shown. Exactly the same applies to the *digs* argument of the other two functions.

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

- `floatToDigits :: (RealFloat a) => Integer -> a -> ([Int], Int)` converts a base and a value to the representation of the value in digits, plus an exponent. More specifically, if

  $$\texttt{floatToDigits}\ b\ r = \texttt{([}d_1, d_2, ...d_n\texttt{], }\ e\texttt{)}$$

  then the following properties hold:

  - $r = 0.d_1 d_2 ..., d_n \ * \ b^e$
  - $n \geq 0$
  - $d_1 \neq 0$ (when $n > 0$)
  - $0 \leq d_i \leq b - 1$

14.2 Reading Functions

- `readSigned :: (Real a) => ReadS a -> ReadS a` reads a *signed* `Real` value, given a reader for an unsigned value.
- `readInt :: (Integral a) => a -> (Char->Bool) -> (Char->Int) -> ReadS a` reads an *unsigned* `Integral` value in an arbitrary base. In the (`readInt` *base isdig d2i*) call, *base* is the base, *isdig* is a predicate distinguishing valid digits in this base, and *d2i* converts a valid digit character to an `Int`.
- `readFloat :: (RealFrac a) => ReadS a` reads an *unsigned* `RealFrac` value, expressed in decimal scientific notation.
- `readDec, readOct, readHex :: (Integral a) => ReadS a` each read an unsigned number, in decimal, octal, and hexadecimal notation, respectively. In the hexadecimal case, both upper or lower case letters are allowed.
- `lexDigits :: ReadS String` reads a non-empty string of decimal digits.

(NB: `readInt` is the "dual" of `showIntAtBase`, and `readDec` is the "dual" of `showInt`. The inconsistent naming is a historical accident.)

14.3 Miscellaneous

- `fromRat :: (RealFloat a) => Rational -> a` converts a `Rational` value into any type in class `RealFloat`.

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

14.4 Library `Numeric`

```
module Numeric(fromRat,
               showSigned, showIntAtBase,
               showInt, showOct, showHex,
               readSigned, readInt,
               readDec, readOct, readHex,
               floatToDigits,
               showEFloat, showFFloat, showGFloat, showFloat,
               readFloat, lexDigits) where

import Char   ( isDigit, isOctDigit, isHexDigit
              , digitToInt, intToDigit )
import Ratio  ( (%), numerator, denominator )
import Array  ( (!), Array, array )

-- This converts a rational to a floating.  This should be used in the
-- Fractional instances of Float and Double.

fromRat :: (RealFloat a) => Rational -> a
fromRat x =
    if x == 0 then encodeFloat 0 0              -- Handle exceptional cases
    else if x < 0 then - fromRat' (-x)          -- first.
    else fromRat' x

-- Conversion process:
-- Scale the rational number by the RealFloat base until
-- it lies in the range of the mantissa (as used by decodeFloat/encodeFloat).
-- Then round the rational to an Integer and encode it with the exponent
-- that we got from the scaling.
-- To speed up the scaling process we compute the log2 of the number to get
-- a first guess of the exponent.
fromRat' :: (RealFloat a) => Rational -> a
fromRat' x = r
  where b = floatRadix r
        p = floatDigits r
        (minExp0, _) = floatRange r
        minExp = minExp0 - p            -- the real minimum exponent
        xMin = toRational (expt b (p-1))
        xMax = toRational (expt b p)
        p0 = (integerLogBase b (numerator x) -
              integerLogBase b (denominator x) - p) `max` minExp
        f = if p0 < 0 then 1 % expt b (-p0) else expt b p0 % 1
        (x', p') = scaleRat (toRational b) minExp xMin xMax p0 (x / f)
        r = encodeFloat (round x') p'
```

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

```
-- Scale x until xMin <= x < xMax, or p (the exponent) <= minExp.
scaleRat :: Rational -> Int -> Rational -> Rational ->
             Int -> Rational -> (Rational, Int)
scaleRat b minExp xMin xMax p x =
    if p <= minExp then
        (x, p)
    else if x >= xMax then
        scaleRat b minExp xMin xMax (p+1) (x/b)
    else if x < xMin  then
        scaleRat b minExp xMin xMax (p-1) (x*b)
    else
        (x, p)

-- Exponentiation with a cache for the most common numbers.
minExpt = 0::Int
maxExpt = 1100::Int
expt :: Integer -> Int -> Integer
expt base n =
    if base == 2 && n >= minExpt && n <= maxExpt then
        expts!n
    else
        base^n

expts :: Array Int Integer
expts = array (minExpt,maxExpt) [(n,2^n) | n <- [minExpt .. maxExpt]]

-- Compute the (floor of the) log of i in base b.
-- Simplest way would be just divide i by b until it's smaller then b,
-- but that would be very slow!  We are just slightly more clever.
integerLogBase :: Integer -> Integer -> Int
integerLogBase b i =
     if i < b then
        0
     else
        -- Try squaring the base first to cut down the number of divisions.
        let l = 2 * integerLogBase (b*b) i
            doDiv :: Integer -> Int -> Int
            doDiv i l = if i < b then l else doDiv (i `div` b) (l+1)
        in  doDiv (i `div` (b^l)) l

-- Misc utilities to show integers and floats

showSigned :: Real a => (a -> ShowS) -> Int -> a -> ShowS
showSigned showPos p x
  | x < 0     = showParen (p > 6) (showChar '-' . showPos (-x))
  | otherwise = showPos x

-- showInt, showOct, showHex are used for positive numbers only
showInt, showOct, showHex :: Integral a => a -> ShowS
showOct = showIntAtBase  8 intToDigit
showInt = showIntAtBase 10 intToDigit
showHex = showIntAtBase 16 intToDigit
```

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

```
showIntAtBase :: Integral a
              => a              -- base
              -> (Int -> Char)  -- digit to char
              -> a              -- number to show
              -> ShowS
showIntAtBase base intToDig n rest
  | n < 0     = error "Numeric.showIntAtBase: can't show negative numbers"
  | n' == 0   = rest'
  | otherwise = showIntAtBase base intToDig n' rest'
  where
    (n',d) = quotRem n base
    rest'  = intToDig (fromIntegral d) : rest

readSigned :: (Real a) => ReadS a -> ReadS a
readSigned readPos = readParen False read'
                     where read' r  = read'' r ++
                                      [(-x,t) | ("-",s) <- lex r,
                                                (x,t)   <- read'' s]
                           read'' r = [(n,s)  | (str,s) <- lex r,
                                                (n,"")  <- readPos str]

-- readInt reads a string of digits using an arbitrary base.
-- Leading minus signs must be handled elsewhere.

readInt :: (Integral a) => a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readInt radix isDig digToInt s =
   [(foldl1 (\n d -> n * radix + d) (map (fromIntegral . digToInt) ds), r)
          | (ds,r) <- nonnull isDig s ]

-- Unsigned readers for various bases
readDec, readOct, readHex :: (Integral a) => ReadS a
readDec = readInt 10 isDigit    digitToInt
readOct = readInt  8 isOctDigit digitToInt
readHex = readInt 16 isHexDigit digitToInt

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

showEFloat d x =  showString (formatRealFloat FFExponent d x)
showFFloat d x =  showString (formatRealFloat FFFixed d x)
showGFloat d x =  showString (formatRealFloat FFGeneric d x)
showFloat      =  showGFloat Nothing

-- These are the format types.  This type is not exported.

data FFormat = FFExponent | FFFixed | FFGeneric
```

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

```
formatRealFloat :: (RealFloat a) => FFFormat -> Maybe Int -> a -> String
formatRealFloat fmt decs x
  = s
  where
    base = 10
    s = if isNaN x then
            "NaN"
        else if isInfinite x then
            if x < 0 then "-Infinity" else "Infinity"
        else if x < 0 || isNegativeZero x then
            '-' : doFmt fmt (floatToDigits (toInteger base) (-x))
        else
            doFmt fmt (floatToDigits (toInteger base) x)

    doFmt fmt (is, e)
      = let
           ds = map intToDigit is
        in
        case fmt of
          FFGeneric ->
              doFmt (if e < 0 || e > 7 then FFExponent else FFFixed)
                    (is, e)
          FFExponent ->
            case decs of
              Nothing ->
                case ds of
                   []    -> "0.0e0"
                   [d]   -> d : ".0e" ++ show (e-1)
                   d:ds  -> d : '.' : ds ++ 'e':show (e-1)

              Just dec ->
                let dec' = max dec 1 in
                case is of
                  [] -> '0':'.':take dec' (repeat '0') ++ "e0"
                  _ ->
                    let (ei, is') = roundTo base (dec'+1) is
                        d:ds = map intToDigit
                                   (if ei > 0 then init is' else is')
                    in d:'.':ds  ++ "e" ++ show (e-1+ei)
```

```
          FFFixed ->
            case decs of
               Nothing  -- Always prints a decimal point
                 | e > 0     -> take e (ds ++ repeat '0')

                                ++ '.' : mk0 (drop e ds)
                 | otherwise -> "0." ++ mk0 (replicate (-e) '0' ++ ds)
               Just dec ->  -- Print decimal point iff dec > 0
                 let dec' = max dec 0 in
                 if e >= 0 then
                   let (ei, is') = roundTo base (dec' + e) is
                       (ls, rs)  = splitAt (e+ei)
                                             (map intToDigit is')
                   in  mk0 ls ++ mkdot0 rs
                 else
                   let (ei, is') = roundTo base dec'
                                          (replicate (-e) 0 ++ is)
                       d : ds = map intToDigit
                                      (if ei > 0 then is' else 0:is')
                   in  d : mkdot0 ds
            where
              mk0 "" = "0"        -- Print 0.34, not .34
              mk0 s  = s

              mkdot0 "" = ""       -- Print 34, not 34.
              mkdot0 s  = '.' : s  -- when the format specifies no
                                   -- digits after the decimal point

roundTo :: Int -> Int -> [Int] -> (Int, [Int])
roundTo base d is = case f d is of
                (0, is) -> (0, is)
                (1, is) -> (1, 1 : is)
  where b2 = base `div` 2
        f n [] = (0, replicate n 0)
        f 0 (i:_) = (if i >= b2 then 1 else 0, [])
        f d (i:is) =
            let (c, ds) = f (d-1) is
                i' = c + i
            in  if i' == base then (1, 0:ds) else (0, i':ds)
```

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

```
-- Based on "Printing Floating-Point Numbers Quickly and Accurately"
-- by R.G. Burger and R. K. Dybvig, in PLDI 96.
-- The version here uses a much slower logarithm estimator.
-- It should be improved.

-- This function returns a non-empty list of digits (Ints in [0..base-1])
-- and an exponent.  In general, if
--      floatToDigits r = ([a, b, ... z], e)
-- then
--      r = 0.ab..z * base^e
--

floatToDigits :: (RealFloat a) => Integer -> a -> ([Int], Int)
floatToDigits _ 0 = ([], 0)
floatToDigits base x =
    let (f0, e0) = decodeFloat x
        (minExp0, _) = floatRange x
        p = floatDigits x
        b = floatRadix x
        minExp = minExp0 - p            -- the real minimum exponent

        -- Haskell requires that f be adjusted so denormalized numbers
        -- will have an impossibly low exponent.  Adjust for this.
        f :: Integer
        e :: Int
        (f, e) = let n = minExp - e0
                 in  if n > 0 then (f0 `div` (b^n), e0+n) else (f0, e0)

        (r, s, mUp, mDn) =
           if e >= 0 then
               let be = b^e in
               if f == b^(p-1) then
                   (f*be*b*2, 2*b, be*b, b)
               else
                   (f*be*2, 2, be, be)
           else
               if e > minExp && f == b^(p-1) then
                   (f*b*2, b^(-e+1)*2, b, 1)
               else
                   (f*2, b^(-e)*2, 1, 1)
```

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

```
        k = let k0 =
                    if b==2 && base==10 then
                        -- logBase 10 2 is slightly bigger than 3/10 so
                        -- the following will err on the low side.  Ignoring
                        -- the fraction will make it err even more.
                        -- Haskell promises that p-1 <= logBase b f < p.
                        (p - 1 + e0) * 3 `div` 10
                    else
                        ceiling ((log (fromInteger (f+1)) +
                                 fromIntegral e * log (fromInteger b)) /
                                  log (fromInteger base))
                fixup n =
                    if n >= 0 then
                        if r + mUp <= expt base n * s then n else fixup (n+1)
                    else
                        if expt base (-n) * (r + mUp) <= s then n
                                                           else fixup (n+1)
            in  fixup k0
        gen ds rn sN mUpN mDnN =
            let (dn, rn') = (rn * base) `divMod` sN
                mUpN' = mUpN * base
                mDnN' = mDnN * base
            in  case (rn' < mDnN', rn' + mUpN' > sN) of
                (True,  False) -> dn : ds
                (False, True)  -> dn+1 : ds
                (True,  True)  -> if rn' * 2 < sN then dn : ds else dn+1 : ds
                (False, False) -> gen (dn:ds) rn' sN mUpN' mDnN'
        rds =
            if k >= 0 then
                gen [] r (s * expt base k) mUp mDn
            else
                let bk = expt base (-k)
                in  gen [] (r * bk) s (mUp * bk) (mDn * bk)
    in  (map fromIntegral (reverse rds), k)
```

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press

```
-- This floating point reader uses a less restrictive syntax for floating
-- point than the Haskell lexer.  The ‘.’ is optional.
readFloat     :: (RealFrac a) => ReadS a
readFloat r    = [(fromRational ((n%1)*10^^(k-d)),t) | (n,d,s) <- readFix r,
                                                        (k,t)   <- readExp s] ++
                 [ (0/0, t) | ("NaN",t)      <- lex r] ++
                 [ (1/0, t) | ("Infinity",t) <- lex r]
               where
                 readFix r = [(read (ds++ds'), length ds', t)
                             | (ds,d) <- lexDigits r,
                               (ds',t) <- lexFrac d ]

                 lexFrac ('.':ds) = lexDigits ds
                 lexFrac s        = [("",s)]

                 readExp (e:s) | e `elem` "eE" = readExp' s
                 readExp s                     = [(0,s)]

                 readExp' ('-':s) = [(-k,t) | (k,t) <- readDec s]
                 readExp' ('+':s) = readDec s
                 readExp' s       = readDec s

lexDigits        :: ReadS String
lexDigits        =  nonnull isDigit

nonnull          :: (Char -> Bool) -> ReadS String
nonnull p s      =  [(cs,t) | (cs@(_:_),t) <- [span p s]]
```

https://doi.org/10.1017/S0956796803001618 Published online by Cambridge University Press