```haskell
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
-- The (:) operator is built-in syntax, and cannot legally be given
-- a fixity declaration; but its fixity is given by: infixr 5  :
infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`

class  Eq a  where
    (==), (/=) :: a -> a -> Bool
     -- Minimal complete definition: (==) or (/=)

class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
    -- Minimal complete definition: (<=) or compare

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]
    -- Minimal complete definition: toEnum, fromEnum

class  Bounded a  where
    minBound, maxBound :: a

class  (Eq a, Show a) => Num a  where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a
    -- Minimal complete definition: All, except negate or (-)

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
    -- Minimal complete definition: quotRem, toInteger

class  (Num a) => Fractional a  where
    (/)              :: a -> a -> a
    recip            :: a -> a
    fromRational     :: Rational -> a
    -- Minimal complete definition: fromRational and (recip or (/))

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
    -- Minimal complete definition: pi, exp, log, sin, cos, sinh, cosh,
    --   asin, acos, atan, asinh, acosh, atanh
```

```haskell
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
    -- Minimal complete definition: properFraction

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, isInfinite, isDenormalized, isNegativeZero, isIEEE :: a -> Bool
    atan2            :: a -> a -> a
    -- Minimal complete definition:
    -- All except exponent, significand, scaleFloat, atan2

-- Numeric functions
subtract         :: (Num a) => a -> a -> a
even, 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

-- Monadic classes
class  Functor f  where
    fmap              :: (a -> b) -> f a -> f b

class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a
    -- Minimal complete definition: (>>=), return
sequence       :: Monad m => [m a] -> m [a]
sequence_      :: Monad m => [m a] -> m ()

-- The xxxM functions take list arguments, but lift the function or
-- list element to a monad type
mapM             :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as        =  sequence (map f as)
mapM_            :: Monad m => (a -> m b) -> [a] -> m ()
(=<<)            :: Monad m => (a -> m b) -> m a -> m b

-- Trivial type
data  ()  =  ()  deriving (Eq, Ord, Enum, Bounded)

-- identity function
id               :: a -> a
id x             =  x

-- constant function
const            :: a -> b -> a
const x _        =  x

-- function composition
(.)              :: (b -> c) -> (a -> b) -> a -> c
f . g            =  \ x -> f (g x)

-- flip f  takes its (first) two arguments in the reverse order of f.
flip             :: (a -> b -> c) -> b -> a -> c
flip f x y       =  f y x
```

```
seq :: a -> b -> b  -- Primitive

-- right-associating infix application operators
-- (useful in continuation-passing style)
($), ($!) :: (a -> b) -> a -> b
f $  x    =  f x
f $! x    =  x `seq` f x


-- Boolean type
data  Bool  =  False | True     deriving (Eq, Ord, Enum, Read, Show, Bounded)

-- Boolean functions
(&&), (||)        :: Bool -> Bool -> Bool
not               :: Bool -> Bool
otherwise         :: Bool


-- Character type
data Char = ... 'a' | 'b' ... -- Unicode values

instance  Eq Char  where
instance  Ord Char  where
instance  Enum Char  where
instance  Bounded Char  where

type  String = [Char]


-- Maybe type
data  Maybe a  =  Nothing | Just a      deriving (Eq, Ord, Read, Show)

maybe                  :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x

instance  Functor Maybe  where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)
instance  Monad Maybe  where
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing
    return           =  Just
    fail s           =  Nothing

-- Either type
data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)

either                 :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  =  f x
either f g (Right y) =  g y

-- IO type
data IO a = ...  -- abstract

instance  Functor IO where ...
instance Monad IO where ...

-- Ordering type
data  Ordering  =  LT | EQ | GT deriving (Eq, Ord, Enum, Read, Show, Bounded)

data  Int  =  minBound ... -1 | 0 | 1 ... maxBound
  instance Eq, Ord, Num, Real, Enum, Integral, Bounded
data  Integer  =  ... -1 | 0 | 1 ...
  instance Eq, Ord, Num, Real, Enum, Integral
data  Float
  instance Eq, Ord, Num, Real, Fractional, Floating, RealFrac, RealFloat
data  Double
  instance Eq, Ord, Num, Real, Fractional, Floating, RealFrac, RealFloat
```

```
-- The Enum instances for Floats and Doubles are slightly unusual.
-- The 'toEnum' function truncates numbers to Int.  The definitions
-- of enumFrom and enumFromThen allow floats to be used in arithmetic
-- series: [0,0.1 .. 0.95].  However, roundoff errors make these somewhat
-- dubious.  This example may have either 10 or 11 elements, depending on
-- how 0.1 is represented.
instance  Enum Float  where
instance  Enum Double  where


-- Lists
data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
-- Not legal Haskell; for illustration only

instance Functor [] where
    fmap = map
instance  Monad []  where
    m >>= k          = concat (map k m)
    return x         = [x]
    fail s           = []

-- Tuples
data  (a,b)   =  (a,b)    deriving (Eq, Ord, Bounded)
data  (a,b,c) =  (a,b,c)  deriving (Eq, Ord, Bounded)
-- Not legal Haskell; for illustration only

-- component projections for pairs, not provided for triples, quadruples, etc.
fst              :: (a,b) -> a
snd              :: (a,b) -> b

-- curry converts an uncurried function to a curried function;
-- uncurry converts a curried function to a function on pairs.
curry            :: ((a, b) -> c) -> a -> b -> c
uncurry          :: (a -> b -> c) -> ((a, b) -> c)

-- Misc functions
-- until p f  yields the result of applying f until p holds.
until            :: (a -> Bool) -> (a -> a) -> a -> a
until p f x
     | p x       =  x
     | otherwise =  until p f (f x)

-- asTypeOf is a type-restricted version of const.  It is usually used
-- as an infix operator, and its typing forces its first argument
-- (which is usually overloaded) to have the same type as the second.
asTypeOf         :: a -> a -> a
asTypeOf         =  const

-- error stops execution and displays an error message
error            :: String -> a
error            =  primError

-- It is expected that compilers will recognize this and insert error
-- messages that are more appropriate to the context in which undefined
-- appears.
undefined        :: a
undefined        =  error "Prelude.undefined"
```

```
-------------------------------- PreludeList --------------------------------
infixl 9  !!
infixr 5  ++
infix  4  `elem`, `notElem`

-- Map and append
map       :: (a -> b) -> [a] -> [b]
(++)      :: [a] -> [a] -> [a]
filter    :: (a -> Bool) -> [a] -> [a]
concat    :: [[a]] -> [a]
concatMap :: (a -> [b]) -> [a] -> [b]

-- head and tail extract the first element and remaining elements,
-- respectively, of a list, which must be non-empty.  last and init
-- are the dual functions working from the end of a finite list,
-- rather than the beginning.
head             :: [a] -> a
tail             :: [a] -> [a]
last             :: [a] -> a
init             :: [a] -> [a]
null             :: [a] -> Bool

-- length returns the length of a finite list as an Int.
length           :: [a] -> Int

-- List index (subscript) operator, 0-origin
(!!)                :: [a] -> Int -> a

-- foldl, applied to a binary operator, a starting value (typically the
-- left-identity of the operator), and a list, reduces the list using
-- the binary operator, from left to right:
--  foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
-- foldl1 is a variant that has no starting value argument, and  thus must
-- be applied to non-empty lists.  scanl is similar to foldl, but returns
-- a list of successive reduced values from the left:
--      scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
-- Note that  last (scanl f z xs) == foldl f z xs.
-- scanl1 is similar, again without the starting element:
--      scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]
foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs

foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  =  foldl f x xs
foldl1 _ []      =  error "Prelude.foldl1: empty list"

scanl            :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs     =  q : (case xs of []   -> []
                                    x:xs -> scanl f (f q x) xs)

scanl1           :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs)  =  scanl f x xs
scanl1 _ []      =  []

-- foldr, foldr1, scanr, and scanr1 are the right-to-left duals of the
-- above functions.
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr1           :: (a -> a -> a) -> [a] -> a
foldr1 f [x]     =  x
foldr1 f (x:xs)  =  f x (foldr1 f xs)
foldr1 _ []      =  error "Prelude.foldr1: empty list"

scanr             :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     =  [q0]
scanr f q0 (x:xs) =  let qs@(q:_)=scanr f q0 xs in f x q : qs
```

```
scanr1          :: (a -> a -> a) -> [a] -> [a]
scanr1 f []     =  []
scanr1 f [x]    =  [x]
scanr1 f (x:xs) =  let qs@(q:_)=scanr1 f x in f x q : qs

-- iterate f x returns an infinite list of repeated applications of f to x:
-- iterate f x == [x, f x, f (f x), ...]
iterate          :: (a -> a) -> a -> [a]

-- repeat x is an infinite list, with x the value of every element.
repeat           :: a -> [a]

-- replicate n x is a list of length n with x the value of every element
replicate        :: Int -> a -> [a]

-- cycle ties a finite list into a circular one, or equivalently,
-- the infinite repetition of the original list.  It is the identity
-- on infinite lists.
cycle            :: [a] -> [a]
cycle []         =  error "Prelude.cycle: empty list"
cycle xs         =  xs' where xs' = xs ++ xs'

-- take n, applied to a list xs, returns the prefix of xs of length n,
-- or xs itself if n > length xs.  drop n xs returns the suffix of xs
-- after the first n elements, or [] if n > length xs.  splitAt n xs
-- is equivalent to (take n xs, drop n xs).
take                   :: Int -> [a] -> [a]
drop                   :: Int -> [a] -> [a]
splitAt                :: Int -> [a] -> ([a],[a])

-- takeWhile, applied to a predicate p and a list xs, returns the longest
-- prefix (possibly empty) of xs of elements that satisfy p.  dropWhile p xs
-- returns the remaining suffix.  span p xs is equivalent to
-- (takeWhile p xs, dropWhile p xs), while break p uses the negation of p.
takeWhile               :: (a -> Bool) -> [a] -> [a]
dropWhile               :: (a -> Bool) -> [a] -> [a]
span, break             :: (a -> Bool) -> [a] -> ([a],[a])

-- lines breaks a string up into a list of strings at newline characters.
-- The resulting strings do not contain newlines.  Similary, words
-- breaks a string up into a list of words, which were delimited by
-- white space.  unlines and unwords are the inverse operations.
-- unlines joins lines with terminating newlines, and unwords joins
-- words with separating spaces.
lines            :: String -> [String]
words            :: String -> [String]
unlines          :: [String] -> String
unwords          :: [String] -> String

-- reverse xs returns the elements of xs in reverse order. xs must be finite.
reverse          :: [a] -> [a]

-- and returns the conjunction of a Boolean list.  For the result to be
-- True, the list must be finite; False, however, results from a False
-- value at a finite index of a finite or infinite list.  or is the
-- disjunctive dual of and.
and, or          :: [Bool] -> Bool

-- Applied to a predicate and a list, any determines if any element
-- of the list satisfies the predicate.  Similarly, for all.
any, all         :: (a -> Bool) -> [a] -> Bool

-- elem is the list membership predicate, usually written in infix form,
-- e.g., x `elem` xs.  notElem is the negation.
elem, notElem    :: (Eq a) => a -> [a] -> Bool

-- lookup key assocs looks up a key in an association list.
lookup           :: (Eq a) => a -> [(a,b)] -> Maybe b
```

```haskell
-- sum and product compute the sum or product of a finite list of numbers.
sum, product     :: (Num a) => [a] -> a

-- maximum and minimum return the maximum or minimum value from a list,
-- which must be non-empty, finite, and of an ordered type.
maximum, minimum :: (Ord a) => [a] -> a

-- zip takes two lists and returns a list of corresponding pairs.  If one
-- input list is short, excess elements of the longer list are discarded.
-- zip3 takes three lists and returns a list of triples.  Zips for larger
-- tuples are in the List library
zip              :: [a] -> [b] -> [(a,b)]
zip3             :: [a] -> [b] -> [c] -> [(a,b,c)]

-- The zipWith family generalises the zip family by zipping with the
-- function given as the first argument, instead of a tupling function.
-- For example, zipWith (+) is applied to two lists to produce the list
-- of corresponding sums.
zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith3         :: (a->b->c->d) -> [a]->[b]->[c]->[d]

-- unzip transforms a list of pairs into a pair of lists.
unzip            :: [(a,b)] -> ([a],[b])
unzip3           :: [(a,b,c)] -> ([a],[b],[c])
```

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

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

class  Show a  where
    showsPrec        :: Int -> a -> ShowS
    show             :: a -> String
    showList         :: [a] -> ShowS
    -- Mimimal complete definition: show or showsPrec
reads             :: (Read a) => ReadS a
reads             =  readsPrec 0
shows             :: (Show a) => a -> ShowS
shows             =  showsPrec 0
read              :: (Read a) => String -> a
read s            =  case [x | (x,t) <- reads s, ("","") <- lex t] of
                          [x] -> x
                          []  -> error "Prelude.read: no parse"
                          _   -> error "Prelude.read: ambiguous parse"
showChar          :: Char -> ShowS
showString        :: String -> ShowS
showParen         :: Bool -> ShowS -> ShowS
showParen b p     =  if b then showChar '(' . p . showChar ')' else p
readParen         :: Bool -> ReadS a -> ReadS a
readParen b g     =  if b then mandatory else optional
                     where optional r  = g r ++ mandatory r
                           mandatory r = [(x,u) | ("(",s) <- lex r,
                                                  (x,t)   <- optional s,
                                                  (")",u) <- lex t    ]
-- This lexer is not completely faithful to the Haskell lexical syntax.
-- Limitations: Qualified names are not handled properly
--              Octal & hexidecimal numerics aren't recognized as single token
--              Comments are not treated properly
lex             :: ReadS String
lex ""          =  [("","")]
lex (c:s)
   | isSpace c =  lex (dropWhile isSpace s)
lex ('\'':s)    =  [('\'':ch++"'", t) | (ch,'\'':t) <- lexLitChar s, ch /= "'" ]
lex ('"':s)     =  [('"':str, t)      | (str,t) <- lexString s]
  where
    lexString ('"':s) = [("\"",s)]
    lexString s = [(ch++str, u) | (ch,t)<-lexStrItem s, (str,u)<-lexString t ]
    lexStrItem ('\\':'&':s) =  [("\\&",s)]
    lexStrItem ('\\':c:s)
               | isSpace c =  [("\\&",t) | '\\':t<-[dropWhile isSpace s]]
    lexStrItem s           =  lexLitChar s
lex (c:s) | isSingle c = [([c],s)]
          | isSym c    = [(c:sym,t)         | (sym,t) <- [span isSym s]]
          | isAlpha c  = [(c:nam,t)         | (nam,t) <- [span isIdChar s]]
          | isDigit c  = [(c:ds++fe,t)      | (ds,s)  <- [span isDigit s],
                                              (fe,t)  <- lexFracExp s     ]
          | otherwise  = []    -- bad character
  where
     isSingle c =  c `elem` ",;()[]{}_`"
     isSym c    =  c `elem` "!@#$%&*+./<=>?\\^|:-~"
     isIdChar c =  isAlphaNum c || c `elem` "_'"
     lexFracExp ('.':c:cs) | isDigit c = [('.':ds++e,u) |
                             (ds,t)<-lexDigits (c:cs), (e,u)  <- lexExp t]
     lexFracExp s  = lexExp s
     lexExp (e:s) | e `elem` "eE"
                    = [(e:c:ds,u) | (c:t)  <- [s], c `elem` "+-",
                                                (ds,u) <- lexDigits t] ++
                      [(e:ds,t)   | (ds,t) <- lexDigits s]
     lexExp s = [("",s)]
instance Read a Show jsou vsechny zatim definovane typy krome funkci
```

```haskell
----------------------------------- PreludeIO -----------------------------------
type  FilePath = String
data IOError    -- The internals of this type are system dependent
instance  Show IOError  where ...
instance  Eq IOError  where ...

ioError    :: IOError -> IO a
userError  :: String -> IOError
catch      :: IO a -> (IOError -> IO a) -> IO a
putChar    :: Char -> IO ()
putStr     :: String -> IO ()
putStrLn   :: String -> IO ()
print      :: Show a => a -> IO ()
getChar    :: IO Char
getLine    :: IO String
getContents:: IO String
interact   :: (String -> String) -> IO ()
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
-- raises an exception instead of an error
readIO   :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: no parse")
              _   -> ioError (userError "Prelude.readIO: ambiguous parse")
readLn :: Read a => IO a
readLn =  do l <- getLine
             r <- readIO l
             return r

------------------------------------- Ratio -------------------------------------
infixl 7  %

data  (Integral a) => Ratio a = ...
type  Rational =  Ratio Integer

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

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 ...

------------------------------------ Complex ------------------------------------
infix  6  :+

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

realPart, 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, phase  :: (RealFloat a) => Complex 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 ...
```

```
---------------------------------- List ---------------------------------------
infix 5 \\

elemIndex         :: Eq a => a -> [a] -> Maybe Int
elemIndices       :: Eq a => a -> [a] -> [Int]
find              :: (a -> Bool) -> [a] -> Maybe a
findIndex         :: (a -> Bool) -> [a] -> Maybe Int
findIndices       :: (a -> Bool) -> [a] -> [Int]
nub               :: Eq a => [a] -> [a]
nubBy             :: (a -> a -> Bool) -> [a] -> [a]
delete            :: Eq a => a -> [a] -> [a]
deleteBy          :: (a -> a -> Bool) -> a -> [a] -> [a]
(\\)              :: Eq a => [a] -> [a] -> [a]
deleteFirstsBy    :: (a -> a -> Bool) -> [a] -> [a] -> [a]
union             :: Eq a => [a] -> [a] -> [a]
unionBy           :: (a -> a -> Bool) -> [a] -> [a] -> [a]


intersect         :: Eq a => [a] -> [a] -> [a]
intersectBy       :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersperse       :: a -> [a] -> [a]
transpose         :: [[a]] -> [[a]]
partition         :: (a -> Bool) -> [a] -> ([a],[a])
group             :: Eq a => [a] -> [[a]]
groupBy           :: (a -> a -> Bool) -> [a] -> [[a]]
inits             :: [a] -> [[a]]
tails             :: [a] -> [[a]]
isPrefixOf        :: Eq a => [a] -> [a] -> Bool
isSuffixOf        :: Eq a => [a] -> [a] -> Bool
mapAccumL         :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR         :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
unfoldr           :: (b -> Maybe (a,b)) -> b -> [a]
sort              :: Ord a => [a] -> [a]
sortBy            :: (a -> a -> Ordering) -> [a] -> [a]
insert            :: Ord a => a -> [a] -> [a]
insertBy          :: (a -> a -> Ordering) -> a -> [a] -> [a]
maximumBy         :: (a -> a -> Ordering) -> [a] -> a
minimumBy         :: (a -> a -> Ordering) -> [a] -> a
genericLength     :: Integral a => [b] -> a
genericTake       :: Integral a => a -> [b] -> [b]
genericDrop       :: Integral a => a -> [b] -> [b]
genericSplitAt    :: Integral a => a -> [b] -> ([b],[b])
genericIndex      :: Integral a => [b] -> a -> b
genericReplicate  :: Integral a => a -> b -> [b]

zip4              :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip5              :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip6              :: [a] -> [b] -> [c] -> [d] -> [e] -> [f]
                        -> [(a,b,c,d,e,f)]
zip7              :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
                        -> [(a,b,c,d,e,f,g)]
zipWith4          :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith5          :: (a->b->c->d->e->f) ->
                     [a]->[b]->[c]->[d]->[e]->[f]
zipWith6          :: (a->b->c->d->e->f->g) ->
                     [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith7          :: (a->b->c->d->e->f->g->h) ->
                     [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
unzip4            :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip5            :: [(a,b,c,d,e)] -> ([a],[b],[c],[d],[e])
unzip6            :: [(a,b,c,d,e,f)] -> ([a],[b],[c],[d],[e],[f])
unzip7            :: [(a,b,c,d,e,f,g)] -> ([a],[b],[c],[d],[e],[f],[g])
```

```
---------------------------------- 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


---------------------------------- Maybe -------------------------------------
isJust, isNothing     :: Maybe a -> Bool
fromJust              :: Maybe a -> a
fromMaybe             :: a -> Maybe a -> a
listToMaybe           :: [a] -> Maybe a
maybeToList           :: Maybe a -> [a]
catMaybes             :: [Maybe a] -> [a]
mapMaybe              :: (a -> Maybe b) -> [a] -> [b]


---------------------------------- Char --------------------------------------
isAscii, isLatin1, isControl, isPrint, isSpace, isUpper :: Char -> Bool
isLower, isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum :: Char -> Bool

toUpper, toLower         :: Char -> Char

digitToInt :: Char -> Int
intToDigit :: Int -> Char

ord        :: Char -> Int
chr        :: Int  -> Char

lexLitChar  :: ReadS String
readLitChar :: ReadS Char
showLitChar :: Char -> ShowS


---------------------------------- System ------------------------------------
data ExitCode = ExitSuccess | ExitFailure Int
                deriving (Eq, Ord, Read, Show)

getArgs      :: IO [String]
getProgName  :: IO String
getEnv       :: String -> IO String
system       :: String -> IO ExitCode
exitWith     :: ExitCode -> IO a
exitFailure  :: IO a


---------------------------------- CPUTime -----------------------------------
getCPUTime         :: IO Integer  ---v pikosekundach---
cpuTimePrecision   :: Integer     ---kolik nejmene se umi odmerit---
```

```
----------------------------------- Random -------------------------------------
class RandomGen g where
  genRange :: g -> (Int, Int)
  next     :: g -> (Int, g)
  split    :: g -> (g, g)

---------------- A standard instance of RandomGen -----------
data StdGen = ... -- Abstract

instance RandomGen StdGen where ...
instance Read      StdGen where ...
instance Show      StdGen where ...

mkStdGen :: Int -> StdGen

---------------- The Random class ---------------------------
class Random a where
   randomR :: RandomGen g => (a, a) -> g -> (a, g)
   random  :: RandomGen g => g -> (a, g)

   randomRs :: RandomGen g => (a, a) -> g -> [a]
   randoms  :: RandomGen g => g -> [a]

   randomRIO :: (a,a) -> IO a
   randomIO  :: IO a

instance Random Int     where ...
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...

---------------- The global random generator ----------------
newStdGen    :: IO StdGen
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a

----------------------------------- Ix -----------------------------------------
class  Ord a => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) -> a -> Int
    inRange     :: (a,a) -> a -> Bool
    rangeSize   :: (a,a) -> Int

instance Ix Char, Ix Int, Ix Integer, Ix (a, b), ..., Ix Bool, Ix Ordering

----------------------------------- Array --------------------------------------
infixl 9  !, //
data  (Ix a)    => Array a b = ... -- Abstract

array           :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray       :: (Ix a) => (a,a) -> [b] -> Array a b
(!)             :: (Ix a) => Array a b -> a -> b
bounds          :: (Ix a) => Array a b -> (a,a)
indices         :: (Ix a) => Array a b -> [a]
elems           :: (Ix a) => Array a b -> [b]
assocs          :: (Ix a) => Array a b -> [(a,b)]
accumArray      :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)] -> Array a b
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
accum           :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)] -> Array a b
ixmap           :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c -> Array a c

instance                             Functor (Array a) where ...
instance  (Ix a, Eq b)            => Eq   (Array a b)  where ...
instance  (Ix a, Ord b)           => Ord  (Array a b)  where ...
instance  (Ix a, Show a, Show b) => Show (Array a b)  where ...
instance  (Ix a, Read a, Read b) => Read (Array a b)  where ...
```