{-
  This is a list of selected functions from the standard Haskell modules:

  Prelude
  Data.List
  Data.Maybe
  Data.Char
-
}

-- standard type classes

class Eq a where
  show :: a -> String

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

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
  toInteger       :: a -> Integer

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

class (Fractional a) => Floating a where
  exp, log, sqrt   :: a -> a
  sin, cos, tan    :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round :: (Integral b) => a -> b
  ceiling, floor  :: (Integral b) => a -> b

-- numerical functions

even, odd        :: (Integral a) => a -> Bool
  even n          = n `rem` 2 == 0
  odd             = not . even

-- monadic functions

sequence       :: Monad m => [m] -> m [a]
sequence       = foldr mcons (return [])
  where mcons p q = do x <- p; xs <- q; return (x:xs)

sequence_      :: Monad m => [m a] -> m 
sequence_ xs   = do sequence xs; return ()

-- functions on functions

id              :: a -> a
  id x            = x

const           :: a -> b -> a
  const x_        = x

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

flip            :: (a -> b -> c) -> b -> a -> c
  flip f x y     = f y x

($)             :: (a -> b) -> a -> b
  f $ x          = f x

-- standard type classes

class Eq a where
  show :: a -> String

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

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
  toInteger       :: a -> Integer

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

class (Fractional a) => Floating a where
  exp, log, sqrt   :: a -> a
  sin, cos, tan    :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round :: (Integral b) => a -> b
  ceiling, floor  :: (Integral b) => a -> b

-- numerical functions

even, odd        :: (Integral a) => a -> Bool
  even n          = n `rem` 2 == 0
  odd             = not . even

-- monadic functions

sequence       :: Monad m => [m] -> m [a]
sequence       = foldr mcons (return [])
  where mcons p q = do x <- p; xs <- q; return (x:xs)

sequence_      :: Monad m => [m a] -> m 
sequence_ xs   = do sequence xs; return ()

-- functions on functions

id              :: a -> a
  id x            = x

const           :: a -> b -> a
  const x_        = x

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

flip            :: (a -> b -> c) -> b -> a -> c
  flip f x y     = f y x

($)             :: (a -> b) -> a -> b
  f $ x          = f x
-- functions on Bools

data Bool = False | True

(&&), (||) :: Bool -> Bool -> Bool
True && x = x
False && _ = False
True || _ = True
False || x = x

not :: Bool -> Bool
not True = False
not False = True

-- functions on Maybe

data Maybe a = Nothing | Just a

isJust :: Maybe a -> Bool
isJust (Just a) = True
isJust Nothing = False

isNothing :: Maybe a -> Bool
isNothing = not . isJust

fromJust :: Maybe a -> a
fromJust (Just a) = a

maybeToList :: Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just a) = [a]

listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (a:_:_) = Just a

-- functions on pairs

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

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

curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p = f (fst p) (snd p)

-- functions on lists

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

(+++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (++) [] xs ys

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

head, last :: [a] -> a
head (x:_:) = x
last [x] = x
last (_:xs) = last xs

tail, init :: [a] -> [a]
tail (_:xs) = xs
init [x] = []
init (x:xs) = x : init xs

null :: [a] -> Bool
null [] = True
null (_:_:) = False
length :: [a] -> Int
length [] = 0
length (_:l) = 1 + length l

(||) :: [a] -> Int -> a
(_x:) !! 0 = x
(_x:) !! n = xs !! (n-1)

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

repeat :: a -> [a]
repeat x = xs where xs = x:xs

replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)

cycle :: [a] -> [a]
cycle [] = error "Prelude.cycle: empty list"
cycle xs = xs' where xs' = xs ++ xs

take, drop :: Int -> [a] -> [a]
take n (x:y:xs) | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

drop n xs | n <= 0 = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs

splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) |
| p x = x : takeWhile p xs
| otherwise = []
dropWhile p [] = []
dropWhile p x:s@(x':xs') |
| p x = dropWhile p x's
| otherwise = xs

lines, words :: String -> [String]
-- lines "apa\nbepa\ncepa\n" == ["apa","bepa","cepap""]
-- words "apa bepa\ncepa" == ["apa","bepa","cepap"]

unlines, unwords :: [String] -> String
-- unlines ["apa","bepa","cepap"] == "apa\nbepa\ncepa"
-- unwords ["apa","bepa","cepap"] == "apa bepa cepap"

reverse :: [a] -> [a]
reverse = foldl (flip (:)) []

and, or :: [Bool] -> Bool
and = foldr (&&) True
or = foldr (||) False

any, all :: (a -> Bool) -> [a] -> Bool
any p = or . map p
all p = and . map p

elem, notElem :: (Eq a) => a -> [a] -> Bool
elem x = any (== x)
notElem x = all (/= x)

lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key [] = Nothing
lookup key ((x,y):xs) |
| key == x = Just y
| otherwise = lookup key xs

sum, product :: (Num a) => [a] -> a
sum = foldl (+) 0
product = foldl (*) 1
maximum, minimum :: (Ord a) => [a] -> (a, a)
maximum [] = error "Prelude.maximum: empty list"
maximum xs = foldl1 max xs
minimum [] = error "Prelude.minimum: empty list"
minimum xs = foldl1 min xs
zip :: [a] -> [b] -> [(a, b)]
zip = zipWith (,)
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _ = []
unzip :: [(a, b)] -> ([a], [b])
unzip = foldr (\(a, b) \(as, bs) \(a:as, b:bs) \([])=[],[])
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x \( nub \( y \( y < xs, x /= y)
delete :: Eq a => [a] -> [a]
delete y [] = []
delete y (x:xs) = if x == y then xs else x : delete y xs
(\() =: Eq a => [a] -> [a] -> [a]
(\) = foldl (flip delete)
union :: Eq a => [a] -> [a] -> [a]
union xs ys = xs ++ (ys \( \ys)
intersect :: Eq a => [a] -> [a] -> [a]
intersect xs ys = \( x \( x < xs, x \( 'elem' \ys)
intersperse :: a -> [a] -> [a]
intersperse 0 [1,2,3,4] = [1,0,2,0,3,0,4]
transpose :: [[a]] -> [[a]]
transpose [[1,2,3],[4,5,6]] = [[1,4],[2,5],[3,6]]
partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs, filter (not . p) xs)
group :: Eq a => [a] -> [[a]]
group "aaaaaabbbee" = ["aa","p","aa","bbb","eee"]
isPrefixOf, isSuffixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf [] = True
isPrefixOf _ [] = False
isPrefixOf (x:xs) (y:ys) = x == y && isPrefixOf xs ys
isSuffixOf x y = reverse x \( "isPrefixOf" \) reverse y
sort :: (Ord a) => [a] -> [a]
sort = foldr insert []
insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x <= y then x:y:ys else y:insert x ys

---------------------------------------------------------------

-- functions on Char

toUpper, toLower :: Char -> Char
  -- toUpper 'a' = 'A'
  -- toLower 'Z' = 'z'
digitToInt :: Char -> Int
digitToInt 'B' = 8
intToDigit :: Int -> Char
  -- intToDigit 3 = '3'
ord :: Char -> Int
chr :: Int -> Char

---------------------------------------------------------------