A Tour of the Haskell Prelude

(and a few other basic functions)

Authors: Bernie Pope (original content), Arjan van IJzendoorn (HTML-isation and updates), Clem Baker-Finch (updated for Haskell 98 hierarchical libraries organisation).

This webpage is a HTML version of most of Bernie Pope's paper "A Tour of the Haskell Prelude": http://www.cs.mu.oz.au/~bjpop/papers.html.

To make searching easy I've included a list of functions below. Otherwise, when you look for "map" using your browser, you'll not only find the definition but all its uses, too.

This is not a complete reference for the Haskell Prelude. *It focuses on some of the more basic functions that may be of most use to beginningstudents. Type classes are not covered.*

---

abs, all, and, any, atan, break, ceiling, chr, compare, concat, concatMap, const, cos, digitToInt, div, drop, dropWhile, elem, error, even, exp, filter, flip, floor, foldl, foldl1, foldr, foldr1, fromIntegral, fst, gcd, head, id, init, isAlpha, isDigit, isLower, isSpace, isUpper, iterate, last, lcm, length, lines, log, map, max, maximum, min, minimum, mod, not, notElem, null, odd, or, ord, pi, pred, putStr, product, quot, rem, repeat, replicate, reverse, round, show, sin, snd, sort, span, splitAt, sqrt, subtract, succ, sum, tail, take, takeWhile, tan, toLower, toUpper, truncate, undefined, unlines, until, unwords, words, zip, zipWith, (!!), (.), (*), (**), (^), (^^), (%), (/), (-), (:), (+), (++), (/=), (==), (<), (<=), (>), (>=), (&&), (||)

---

abs

type: `abs :: Num a => a -> a`

description: returns the absolute value of a number.

definition:

```
abs x
  | x >= 0 = x
  | otherwise = -x
```

usage:

```
Prelude> abs (-3)
3
```

all

type: `all :: (a -> Bool) -> [a] -> Bool`

description: applied to a predicate and a list, returns True if all elements of the list satisfy the predicate, and False otherwise. Similar to the function any.

definition:

```
all p xs = and (map p xs)
```

usage:

```
Prelude> all (<11) [1..10]
True
Prelude> all isDigit "123abc"
False
```

and

type: `and :: [Bool] -> Bool`

description: takes the logical conjunction of a list of boolean values (see also `or').

definition:

```
and xs = foldr (&&) True xs
```

usage:

```
Prelude> and [True, True, False, True]
False
Prelude> and [True, True, True, True]
```

```
True
Prelude> and []
True
```

any

type: `any :: (a -> Bool) -> [a] -> Bool`

description: applied to a predicate and a list, returns True if any of the elements of the list satisfy the predicate, and False otherwise. Similar to the function all.

definition: `any p xs = or (map p xs)`

usage:

```
Prelude> any (<11) [1..10]
True
Prelude> any isDigit "123abc"
True
Prelude> any isDigit "alphabetics"
False
```

atan

type: `atan :: Floating a => a -> a`

description: the trigonometric function inverse tan.

definition: `defined internally.`

usage:

```
Prelude> atan pi
1.26263
```

break

type: `break :: (a -> Bool) -> [a] -> ([a],[a])`

description: given a predicate and a list, breaks the list into two lists (returned as a tuple) at the point where the predicate is first satisfied. If the predicate is never satisfied then the first element of the resulting tuple is the entire list and the second element is the empty list ([]).

definition:

```
break p xs
  = span p' xs
    where
    p' x = not (p x)
```

usage:

```
Prelude> break isSpace "hello there fred"
("hello", " there fred")
Prelude> break isDigit "no digits here"
("no digits here","")
```

ceiling

type: `ceiling :: (RealFrac a, Integral b) => a -> b`

description: returns the smallest integer not less than its argument.

usage:

```
Prelude> ceiling 3.8
4
Prelude> ceiling (-3.8)
-3
```

see also: floor

chr

type: `chr :: Int -> Char`

description: applied to an integer in the range 0 -- 255, returns the character whose ascii code is that integer. It is the converse of the function ord. An error will result if chr is applied to an integer outside the correct range. *[Import from* `Data.Char`*]*

definition: `defined internally.`

usage:

```
Prelude> chr 65
'A'
Prelude> (ord (chr 65)) == 65
True
```

see also: ord

compare

type:

```
compare :: Ord a => a -> a -> Ordering
```

description: applied to to values of the same type which have an ordering defined on them, returns a value of type Ordering which will be: EQ if the two values are equal; GT if the first value is strictly greater than the second; and LT if the first value is less than or equal to the second value.

definition:

```
compare x y
    | x == y = EQ
    | x <= y = LT
    | otherwise = GT
```

usage:

```
Prelude> compare "aadvark" "zebra"
LT
```

concat

type:

```
concat :: [[a]] -> [a]
```

description: applied to a list of lists, joins them together using the ++ operator.

definition:

```
concat xs = foldr (++) [] xs
```

usage:

```
Prelude> concat [[1,2,3], [4], [], [5,6,7,8]]
[1, 2, 3, 4, 5, 6, 7, 8]
```

concatMap

type:

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

description: given a function which maps a value to a list, and a list of elements of the same type as the value, applies the function to the list and then concatenates the result (thus flattening the resulting list).

definition:

```
concatMap f = concat . map f
```

usage:

```
Prelude> concatMap show [1,2,3,4]
"1234"
```

const

type:

```
const :: const :: a -> b -> a
```

description: creates a constant valued function which always has the value of its first argument, regardless of the value of its second argument.

definition:

```
const k _ = k
```

usage:

```
Prelude> const 12 "lucky"
12
```

cos

type:

```
cos :: Floating a => a -> a
```

description: the trigonometric cosine function, arguments are interpreted to be in radians.

definition:

```
defined internally.
```

usage:

```
Prelude> cos pi
-1.0
Prelude> cos (pi/2)
-4.37114e-08
```

digitToInt

type: `digitToInt :: Char -> Int`

description: converts a digit character into the corresponding integer value of the digit. *[Import from `Data.Char`]*

definition:

```
digitToInt :: Char -> Int
digitToInt c
  | isDigit c            =  fromEnum c - fromEnum '0'
  | c >= 'a' && c <= 'f' =  fromEnum c - fromEnum 'a' + 10
  | c >= 'A' && c <= 'F' =  fromEnum c - fromEnum 'A' + 10
  | otherwise            =  error "Char.digitToInt: not a digit"
```

usage:

```
Prelude> digitToInt '3'
3
```

div

type: `div :: Integral a => a -> a -> a`

description: computes the integer division of its integral arguments.

definition: `defined internally.`

usage:

```
Prelude> 16 `div` 9
1
Prelude> (-12) `div` 5
-3
```

notes: `` `div` `` is integer division such that the result is truncated towards negative infinity.

drop

type: `drop :: Int -> [a] -> [a]`

description: applied to a number and a list, returns the list with the specified number of elements removed from the front of the list. If the list has less than the required number of elements then it returns [].

definition:

```
drop 0 xs            = xs
drop _ []            = []
drop n (_:xs) | n>0  = drop (n-1) xs
drop _ _             = error "PreludeList.drop: negative argument"
```

usage:

```
Prelude> drop 3 [1..10]
[4, 5, 6, 7, 8, 9, 10]
Prelude> drop 4 "abc"
""
```

dropWhile

type: `dropWhile :: (a -> Bool) -> [a] -> [a]`

description: applied to a predicate and a list, removes elements from the front of the list while the predicate is satisfied.

definition:

```
dropWhile p [] = []
dropWhile p (x:xs)
  | p x = dropWhile p xs
  | otherwise = (x:xs)
```

usage:

```
Prelude> dropWhile (<5) [1..10]
[5, 6, 7, 8, 9, 10]
```

elem

type: `elem :: Eq a => a -> [a] -> Bool`

description: applied to a value and a list returns True if the value is in the list and False otherwise. The elements of the list must be of the same type as the value.

definition: `elem x xs = any (== x) xs`

usage:

```
Prelude> elem 5 [1..10]
True
Prelude> elem "rat" ["fat", "cat", "sat", "flat"]
False
```

error

type: `error :: String -> a`

description: applied to a string creates an error value with an associated message. Error values are equivalent to the undefined value (undefined), any attempt to access the value causes the program to terminate and print the string as a diagnostic.

definition: `defined internally.`

usage: `error "this is an error message"`

even

type: `even :: Integral a => a -> Bool`

description: applied to an integral argument, returns True if the argument is even, and False otherwise.

definition: ``even n = n `rem` 2 == 0``

usage:

```
Prelude> even 2
True
Prelude> even (11 * 3)
False
```

exp

type: `exp :: Floating a => a -> a`

description: the exponential function (exp n is equivalent to e^n).

definition: `defined internally.`

usage:

```
Prelude> exp 1
2.71828
```

filter

type: `filter :: (a -> Bool) -> [a] -> [a]`

description: applied to a predicate and a list, returns a list containing all the elements from the argument list that satisfy the predicate.

definition: `filter p xs = [k | k <- xs, p k]`

usage:

```
Prelude> filter isDigit "fat123cat456"
"123456"
```

flip

type: `flip :: (a -> b -> c) -> b -> a -> c`

description: applied to a binary function, returns the same function with the order of the arguments reversed.

definition: `flip f x y = f y x`

usage:

```
Prelude> flip elem [1..10] 5
True
```

floor

type: `floor :: (RealFrac a, Integral b) => a -> b`

description: returns the largest integer not greater than its argument.

usage:

```
Prelude> floor 3.8
3
Prelude> floor (-3.8)
```

```
-4
```

see also: ceiling

foldl

type:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

description: folds up a list, using a given binary operator and a given start value, in a left associative manner.

```
foldl op r [a, b, c] → ((r `op` a) `op` b) `op` c
```

definition:

```
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

usage:

```
Prelude> foldl (+) 0 [1..10]
55
Prelude> foldl (flip (:)) [] [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

foldl1

type:

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

description: folds left over non--empty lists.

definition:

```
foldl1 f (x:xs) = foldl f x xs
```

usage:

```
Prelude> foldl1 max [1, 10, 5, 2, -1]
10
```

foldr

type:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

description: folds up a list, using a given binary operator and a given start value, in a right associative manner.

```
foldr op r [a, b, c] → a `op` (b `op` (c `op` r))
```

definition:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

usage:

```
Prelude> foldr (++) [] ["con", "cat", "en", "ate"]
"concatenate"
```

foldr1

type:

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

description: folds right over non--empty lists.

definition:

```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

usage:

```
Prelude> foldr1 (*) [1..10]
3628800
```

fromIntegral

type:

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

description: Converts from an Integer or Int to a numeric type which is in the class Num.

usage:

```
Prelude> (fromIntegral 10000000000)::Float
1.0e+10
```

fst

type:

```
fst :: (a, b) -> a
```

description: returns the first element of a two element tuple.

definition:

```
fst (x, _) = x
```

usage:

```
Prelude> fst ("harry", 3)
```

```
"harry"
```

gcd

type: `gcd :: Integral a => a -> a -> a`

description: returns the greatest common divisor between its two integral arguments.

definition:

```
gcd 0 0 = error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y = gcd' (abs x) (abs y)
          where
             gcd' x 0 = x
             gcd' x y = gcd' y (x `rem` y)
```

usage:

```
Prelude> gcd 2 10
2
Prelude> gcd (-7) 13
1
```

head

type: `head :: [a] -> a`

description: returns the first element of a non--empty list. If applied to an empty list an error results.

definition: `head (x:_) = x`

usage:

```
Prelude> head [1..10]
1
Prelude> head ["this", "and", "that"]
"this"
```

id

type: `id :: a -> a`

description: the identity function, returns the value of its argument.

definition: `id x = x`

usage:

```
Prelude> id 12
12
Prelude> id (id "fred")
"fred"
Prelude> (map id [1..10]) == [1..10]
True
```

init

type: `init :: [a] -> [a]`

description: returns all but the last element of its argument list. The argument list must have at least one element. If init is applied to an empty list an error occurs.

definition:

```
init [x] = []
init (x:xs) = x : init xs
```

usage:

```
Prelude> init [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

isAlpha

type: `isAlpha :: Char -> Bool`

description: applied to a character argument, returns True if the character is alphabetic, and False otherwise. *[Import from* `Data.Char`*]*

definition: `isAlpha c = isUpper c || isLower c`

usage:

```
Prelude> isAlpha 'a'
True
Prelude> isAlpha '1'
```

```
False
```

isDigit

type: `isDigit :: Char -> Bool`

description: applied to a character argument, returns True if the character is a numeral, and False otherwise. *[Import from `Data.Char`]*

definition:
```
isDigit c = c >= '0' && c <= '9'
```

usage:
```
Prelude> isDigit '1'
True
Prelude> isDigit 'a'
False
```

isLower

type: `isLower :: Char -> Bool`

description: applied to a character argument, returns True if the character is a lower case alphabetic, and False otherwise. *[Import from `Data.Char`]*

definition:
```
isLower c = c >= 'a' && c <= 'z'
```

usage:
```
Prelude> isLower 'a'
True
Prelude> isLower 'A'
False
Prelude> isLower '1'
False
```

isSpace

type: `isSpace :: Char -> Bool`

description: returns True if its character argument is a whitespace character and False otherwise. *[Import from `Data.Char`]*

definition:
```
isSpace c  = c == ' '  || c == '\t' || c == '\n' ||
               c == '\r' || c == '\f' || c == '\v'
```

usage:
```
Prelude> dropWhile isSpace "   \nhello  \n"
"hello  \n"
```

isUpper

type: `isUpper :: Char -> Bool`

description: applied to a character argument, returns True if the character is an upper case alphabetic, and False otherwise. *[Import from `Data.Char`]*

definition:
```
isUpper c = c >= 'A' && c <= 'Z'
```

usage:
```
Prelude> isUpper 'A'
True
Prelude> isUpper 'a'
False
Prelude> isUpper '1'
False
```

iterate

type: `iterate :: (a -> a) -> a -> [a]`

description: iterate~f~x returns the infinite list [x,~f(x),~f(f(x)),~...].

definition:
```
iterate f x = x : iterate f (f x)
```

usage:
```
Prelude> iterate (+1) 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, .....
```

last

type:

```
last :: [a] -> a
```

description: applied to a non--empty list, returns the last element of the list.

definition:

```
last [x] = x
last (_:xs) = last xs
```

usage:

```
Prelude> last [1..10]
10
```

lcm

type:

```
lcm :: Integral a => a -> a -> a
```

description: returns the least common multiple of its two integral arguments.

definition:

```
lcm _ 0 = 0
lcm 0 _ = 0
lcm x y = abs ((x `quot` gcd x y) * y)
```

usage:

```
Prelude> lcm 2 10
10
Prelude> lcm 2 11
22
```

length

type:

```
length :: [a] -> Int
```

description: returns the number of elements in a finite list.

definition:

```
length [] = 0
length (x:xs) = 1 + length xs
```

usage:

```
Prelude> length [1..10]
10
```

lines

type:

```
lines :: String -> [String]
```

description: applied to a list of characters containing newlines, returns a list of lists by breaking the original list into lines using the newline character as a delimiter. The newline characters are removed from the result.

definition:

```
lines [] = []
lines (x:xs)
  = l : ls
  where
  (l, xs') = break (== '\n') (x:xs)
  ls
     | xs' == [] = []
     | otherwise = lines (tail xs')
```

usage:

```
Prelude> lines "hello world\nit's me,\neric\n"
["hello world", "it's me,", "eric"]
```

log

type:

```
log :: Floating a => a -> a
```

description: returns the natural logarithm of its argument.

definition:

```
defined internally.
```

usage:

```
Prelude> log 1
0.0
Prelude> log 3.2
1.16315
```

map

type:

```
map :: (a -> b) -> [a] -> [b]
```

description: given a function, and a list of any type, returns a list where each element is the result of applying the function to the corresponding element in the input list.

definition:

```
map f xs = [f x | x <- xs]
```

usage:

```
Prelude> map sqrt [1..5]
[1.0, 1.41421, 1.73205, 2.0, 2.23607]
```

max

type:

```
max :: Ord a => a -> a -> a
```

description: applied to two values of the same type which have an ordering defined upon them, returns the maximum of the two elements according to the operator >=.

definition:

```
max x y
  | x >= y = x
  | otherwise = y
```

usage:

```
Prelude> max 1 2
2
```

maximum

type:

```
maximum :: Ord a => [a] -> a
```

description: applied to a non--empty list whose elements have an ordering defined upon them, returns the maximum element of the list.

definition:

```
maximum xs = foldl1 max xs
```

usage:

```
Prelude> maximum [-10, 0 , 5, 22, 13]
22
```

min

type:

```
min :: Ord a => a -> a -> a
```

description: applied to two values of the same type which have an ordering defined upon them, returns the minimum of the two elements according to the operator <=.

definition:

```
min x y
  | x <= y = x
  | otherwise = y
```

usage:

```
Prelude> min 1 2
1
```

minimum

type:

```
minimum :: Ord a => [a] -> a
```

description: applied to a non--empty list whose elements have an ordering defined upon them, returns the minimum element of the list.

definition:

```
minimum xs = foldl1 min xs
```

usage:

```
Prelude> minimum [-10, 0 , 5, 22, 13]
-10
```

mod

type:

```
mod :: Integral a => a -> a -> a
```

description: returns the modulus of its two arguments.

definition:

```
defined internally.
```

usage:

```
Prelude> 16 `mod` 9
7
```

not

type: `not :: Bool -> Bool`

description: returns the logical negation of its boolean argument.

definition:
```
not True = False
not False = True
```

usage:
```
Prelude> not (3 == 4)
True
Prelude> not (10 > 2)
False
```

notElem

type: `notElem :: Eq a => a -> [a] -> Bool`

description: returns `True` if its first argument is *not* an element of the list as its second argument.

usage:
```
Prelude> 3 `notElem` [1,2,3]
False
Prelude> 4 `notElem` [1,2,3]
True
```

null

type: `null :: [a] -> Bool`

description: returns True if its argument is the empty list ([]) and False otherwise.

definition:
```
null [] = True
null (_:_) = False
```

usage:
```
Prelude> null []
True
Prelude> null (take 3 [1..10])
False
```

odd

type: `odd :: Integral a => a -> Bool`

description: applied to an integral argument, returns True if the argument is odd, and False otherwise.

definition:
```
odd = not . even
```

usage:
```
Prelude> odd 1
True
Prelude> odd (2 * 12)
False
```

or

type: `or :: [Bool] -> Bool`

description: applied to a list of boolean values, returns their logical disjunction (see also `and').

definition:
```
or xs = foldr (||) False xs
```

usage:
```
Prelude> or [False, False, True, False]
True
Prelude> or [False, False, False, False]
False
Prelude> or []
False
```

ord

type: `ord :: Char -> Int`

description: applied to a character, returns its ascii code as an integer. *[Import from* `Data.Char`*]* *definition:* `defined internally.`

usage:

```
Prelude> ord 'A'
65
Prelude> (chr (ord 'A')) == 'A'
True
```

see also: chr

pi

type: `pi :: Floating a => a`

description: the ratio of the circumference of a circle to its diameter.

definition: `defined internally.`

usage:

```
Prelude> pi
3.14159
Prelude> cos pi
-1.0
```

pred

type: `pred :: Enum a => a -> a`

description: applied to a value of an enumerated type returns the predecessor (previous value in the enumeration) of its argument. If its argument is the first value in an enumeration an error will occur.

usage:

```
Prelude> pred 1
0
Prelude> pred True
False
```

putStr

type: `putStr :: String -> IO ()`

description: takes a string as an argument and returns an I/O action as a result. A side-effect of applying putStr is that it causes its argument string to be printed to the screen.

definition: `defined internally.`

usage:

```
Prelude> putStr "Hello World\nI'm here!"
Hello World
I'm here!
```

product

type: `product :: Num a => [a] -> a`

description: applied to a list of numbers, returns their product.

definition: `product xs = foldl (*) 1 xs`

usage:

```
Prelude> product [1..10]
3628800
```

quot

type: `quot :: Integral a => a -> a -> a`

description: returns the quotient after dividing the its first integral argument by its second integral argument.

definition: `defined internally.`

usage:

```
Prelude> 16 `quot` 8
2
Prelude> quot 16 9
1
```

rem

type:

```
rem :: Integral a => a -> a -> a
```

description: returns the remainder after dividing its first integral argument by its second integral argument.

definition:

```
defined internally.
```

usage:

```
Prelude> 16 `rem` 8
0
Prelude> rem 16 9
7
```

notes: The following equality holds:

```
(x `quot` y)*y + (x `rem` y) == x
```

repeat

type:

```
repeat :: a -> [a]
```

description: given a value, returns an infinite list of elements the same as the value.

definition:

```
repeat x
  = xs
  where xs = x:xs
```

usage:

```
Prelude> repeat 12
[12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12 ....
```

replicate

type:

```
replicate :: Int -> a -> [a]
```

description: given an integer (positive or zero) and a value, returns a list containing the specified number of instances of that value.

definition:

```
replicate n x = take n (repeat x)
```

usage:

```
Prelude> replicate 3 "apples"
["apples", "apples", "apples"]
```

reverse

type:

```
reverse :: [a] -> [a]
```

description: applied to a finite list of any type, returns a list of the same elements in reverse order.

definition:

```
reverse = foldl (flip (:)) []
```

usage:

```
Prelude> reverse [1..10]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

round

type:

```
round :: (RealFrac a, Integral b) => a -> b
```

description: rounds its argument to the nearest integer.

usage:

```
Prelude> round 3.2
3
Prelude> round 3.5
4
Prelude> round (-3.2)
-3
```

show

type:

```
show :: Show a => a -> String
```

description: converts a value (which must be a member of the Show class), to its string representation.

definition:

```
defined internally.
```

usage:

```
Prelude> "six plus two equals " ++ (show (6 + 2))
"six plus two equals 8"
```

sin

type:
```
sin :: Floating a => a -> a
```

description: the trigonometric sine function, arguments are interpreted to be in radians.

definition:
```
defined internally.
```

usage:
```
Prelude> sin (pi/2)
1.0
Prelude> ((sin pi)^2) + ((cos pi)^2)
1.0
```

snd

type:
```
snd :: (a, b) -> b
```

description: returns the second element of a two element tuple.

definition:
```
snd (_, y) = y
```

usage:
```
Prelude> snd ("harry", 3)
3
```

sort

type:
```
sort :: Ord a => [a] -> [a]
```

description: sorts its argument list in ascending order. The items in the list must be in the class Ord. *[Import from* `Data.List`*]*

usage:
```
List> sort [1, 4, -2, 8, 11, 0]
[-2,0,1,4,8,11]
```

span

type:
```
span :: (a -> Bool) -> [a] -> ([a],[a])
```

description: given a predicate and a list, splits the list into two lists (returned as a tuple) such that elements in the first list are taken from the head of the list while the predicate is satisfied, and elements in the second list are the remaining elements from the list once the predicate is not satisfied.

definition:
```
span p [] = ([],[])
span p xs@(x:xs')
  | p x = (x:ys, zs)
  | otherwise = ([],xs)
    where (ys,zs) = span p xs'
```

usage:
```
Prelude> span isDigit "123abc456"
("123", "abc456")
```

splitAt

type:
```
splitAt :: Int -> [a] -> ([a],[a])
```

description: given an integer (positive or zero) and a list, splits the list into two lists (returned as a tuple) at the position corresponding to the given integer. If the integer is greater than the length of the list, it returns a tuple containing the entire list as its first element and the empty list as its second element.

definition:
```
splitAt 0 xs = ([],xs)
splitAt _ [] = ([],[])
splitAt n (x:xs)
  | n > 0 = (x:xs',xs'')
    where
    (xs',xs'') = splitAt (n-1) xs
splitAt _ _ = error "PreludeList.splitAt: negative argument"
```

usage:
```
Prelude> splitAt 3 [1..10]
([1, 2, 3], [4, 5, 6, 7, 8, 9, 10])
```

```
Prelude> splitAt 5 "abc"
("abc", "")
```

sqrt

type: `sqrt :: Floating a => a -> a`

description: returns the square root of a number.

definition: `sqrt x = x ** 0.5`

usage:

```
Prelude> sqrt 16
4.0
```

subtract

type: `subtract :: Num a => a -> a -> a`

description: subtracts its first argument from its second argument.

definition: `subtract = flip (-)`

usage:

```
Prelude> subtract 7 10
3
```

succ

type: `succ :: Enum a => a -> a`

description: applied to a value of an enumerated type returns the successor (next value in the enumeration) of its argument. If its argument is the last value in an enumeration an error will occur.

definition: `defined internally.`

usage:

```
Prelude> succ 'a'
'b'
Prelude> succ False
True
```

sum

type: `sum :: Num a => [a] -> a`

description: computes the sum of a finite list of numbers.

definition: `sum xs = foldl (+) 0 xs`

usage:

```
Prelude> sum [1..10]
55
```

tail

type: `tail :: [a] -> [a]`

description: applied to a non--empty list, returns the list without its first element.

definition: `tail (_:xs) = xs`

usage:

```
Prelude> tail [1,2,3]
[2,3]
Prelude> tail "hugs"
"ugs"
```

take

type: `take :: Int -> [a] -> [a]`

description: applied to an integer (positive or zero) and a list, returns the specified number of elements from the front of the list. If the list has less than the required number of elements, take returns the entire list.

definition:

```
take 0 _ = []
take _ []= []
take n (x:xs)
```

```
   | n > 0 = x : take (n-1) xs
take _ _ = error "PreludeList.take: negative argument"
```

usage:

```
Prelude> take 4 "goodbye"
"good"
Prelude> take 10 [1,2,3]
[1,2,3]
```

takeWhile

type:

```
takewhile :: (a -> Bool) -> [a] -> [a]
```

description: applied to a predicate and a list, returns a list containing elements from the front of the list while the predicate is satisfied.

definition:

```
takeWhile p [] = []
takeWhile p (x:xs)
   | p x = x : takeWhile p xs
   | otherwise = []
```

usage:

```
Prelude> takeWhile (<5) [1, 2, 3, 10, 4, 2]
[1, 2, 3]
```

tan

type:

```
tan :: Floating a => a -> a
```

description: the trigonometric function tan, arguments are interpreted to be in radians.

definition:

```
defined internally.
```

usage:

```
Prelude> tan (pi/4)
1.0
```

toLower

type:

```
toLower :: Char -> Char
```

description: converts an uppercase alphabetic character to a lowercase alphabetic character. If this function is applied to an argument which is not uppercase the result will be the same as the argument unchanged. *[Import from `Data.Char`]*

definition:

```
toLower c
   | isUpper c = toEnum (fromEnum c - fromEnum 'A' + fromEnum 'a')
   | otherwise = c
```

usage:

```
Prelude> toLower 'A'
'a'
Prelude> toLower '3'
'3'
```

toUpper

type:

```
toUpper :: Char -> Char
```

description: converts a lowercase alphabetic character to an uppercase alphabetic character. If this function is applied to an argument which is not lowercase the result will be the same as the argument unchanged. *[Import from `Data.Char`]*

definition:

```
toUpper c
   | isLower c = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
   | otherwise = c
```

usage:

```
Prelude> toUpper 'a'
'A'
Prelude> toUpper '3'
'3'
```

truncate

type:

```haskell
truncate :: (RealFrac a, Integral b) => a -> b
```

description: drops the fractional part of a floating point number, returning only the integral part.

usage:

```
Prelude> truncate 3.2
3
Prelude> truncate (-3.2)
-3
```

undefined

type:

```haskell
undefined :: a
```

description: an undefined value. It is a member of every type.

definition:

```haskell
undefined
    | False = undefined
```

unlines

type:

```haskell
unlines :: [String] -> String
```

description: converts a list of strings into a single string, placing a newline character between each of them. It is the converse of the function lines.

definition:

```haskell
unlines xs
  = concat (map addNewLine xs)
  where
  addNewLine l = l ++ "\n"
```

usage:

```
Prelude> unlines ["hello world", "it's me,", "eric"]
"hello world\nit's me,\neric\n"
```

until

type:

```haskell
until :: (a -> Bool) -> (a -> a) -> a -> a
```

description: given a predicate, a unary function and a value, it recursively re--applies the function to the value until the predicate is satisfied. If the predicate is never satisfied until will not terminate.

definition:

```haskell
until p f x
  | p x = x
  | otheriwise = until p f (f x)
```

usage:

```
Prelude> until (>1000) (*2) 1
1024
```

unwords

type:

```haskell
unwords :: [String] -> String
```

description: concatenates a list of strings into a single string, placing a single space between each of them.

definition:

```haskell
unwords [] = []
unwords ws
  = foldr1 addSpace ws
  where
  addSpace w s = w ++ (' ':s)
```

usage:

```
Prelude> unwords ["the", "quick", "brown", "fox"]
"the quick brown fox"
```

words

type:

```haskell
words :: String -> [String]
```

description: breaks its argument string into a list of words such that each word is delimited by one or more whitespace characters.

definition:

```haskell
words s
  | findSpace == [] = []
  | otherwise = w : words s''
  where
  (w, s'') = break isSpace findSpace
  findSpace = dropWhile isSpace s
```

usage:

```
Prelude> words "the quick brown\n\nfox"
["the", "quick", "brown", "fox"]
```

zip

type: `zip :: [a] -> [b] -> [(a,b)]`

description: applied to two lists, returns a list of pairs which are formed by tupling together corresponding elements of the given lists. If the two lists are of different length, the length of the resulting list is that of the shortest.

definition:

```
zip xs ys
  = zipWith pair xs ys
  where
  pair x y = (x, y)
```

usage:

```
Prelude> zip [1..6] "abcd"
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

zipWith

type: `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

description: applied to a binary function and two lists, returns a list containing elements formed be applying the function to corresponding elements in the lists.

definition:

```
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

usage:

```
Prelude> zipWith (+) [1..5] [6..10]
[7, 9, 11, 13, 15]
```

(!!)

description: given a list and a number, returns the element of the list whose position is the same as the number.

usage:

```
Prelude> [1..10] !! 0
1
Prelude> "a string" !! 3
't'
```

notes: the valid subscripts for a list l are: 0 .. (length l) - 1. Therefore, negative subscripts are not allowed, nor are subsripts greater than one less than the length of the list argument. Subscripts out of this range will result in a program error.

(.)

description: composes two functions into a single function.

usage:

```
Prelude> (sqrt . sum ) [1,2,3,4,5]
3.87298
```

notes: `(f.g.h) x` is equivalent to `f (g (h x))`.

(**)

description: raises its first argument to the power of its second argument. The arguments must be in the `Floating` numerical type class, and the result will also be in that class.

usage:

```
Prelude> 3.2**pi
38.6345
```

(^)

description: raises its first argument to the power of its second argument. The first argument must be a member of the `Num` type class, and the second argument must be a member of the `Integral` type class. The result will be of the same type as the first argument.

usage:

```
Prelude> 3.2^4
104.858
```

(^^)

description: raises its first argument to the power of its second argument. The first argument must be a member of the `Fractional` type class, and the second argument must be a member of the `Integral` type class. The result will be of the same type as the first argument.

usage:

```
Prelude> 3.142^^4
97.4596
```

(%)

description: takes two numbers in the `Integral` type class and returns the most simple ratio of the two.

usage:

```
Prelude> 20 % 4
5 % 1
Prelude> (5 % 4)^2
25 % 16
```

(*)

description: returns the multiple of its two arguments.

usage:

```
Prelude> 6 * 2.0
12.0
```

(/)

description: returns the result of dividing its first argument by its second. Both arguments must in the type class `Fractional`.

usage:

```
Prelude> 12.0 / 2
6.0
```

(+)

description: returns the addition of its arguments.

usage:

```
Prelude> 3 + 4
7
Prelude> (4 % 5) + (1 % 5)
1 % 1
```

(-)

description: returns the substraction of its second argument from its first.

usage:

```
Prelude> 4 - 3
1
Prelude> 4 - (-3)
7
```

(:)

description: prefixes an element onto the front of a list.

usage:

```
Prelude> 1:[2,3]
[1,2,3]
Prelude> True:[]
[True]
Prelude> 'h':"askell"
"haskell"
```

(++)

description: appends its second list argument onto the end of its first list argument.

usage:

```
Prelude> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
Prelude> "foo " ++ "was" ++ " here"
"foo was here"
```

(/=)

description: is `True` if its first argument is not equal to its second argument, and `False` otherwise. Equality is defined by the == operator. Both of its arguments must be in the `Eq` type class.

usage:

```
Prelude> 3 /= 4
True
Prelude> [1,2,3] /= [1,2,3]
False
```

(==)

description: is `True` if its first argument is equal to its second argument, and `False` otherwise. Equality is defined by the == operator. Both of its arguments must be in the `Eq`

usage:

```
Prelude> 3 == 4
False
Prelude> [1,2,3] == [1,2,3]
True
```

(<)

description: returns `True` if its first argument is strictly less than its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

usage:

```
Prelude> 1 < 2
True
Prelude> 'a' < 'z'
True
Prelude> True < False
False
```

(<=)

description: returns `True` if its first argument is less than or equal to its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

usage:

```
Prelude> 3 <= 4
True
Prelude> 4 <= 4
True
Prelude> 5 <= 4
False
```

(>)

.

description: returns `True` if its first argument is strictly greater than its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`

usage:

```
Prelude> 2 > 1
True
Prelude> 'a' > 'z'
False
Prelude> True > False
True
```

(>=)

description: returns `True` if its first argument is greater than or equal to its second argument, and `False` otherwise. Both arguments must be in the type class `Ord`.

usage:

```
Prelude> 4 >= 3
True
Prelude> 4 >= 4
True
Prelude> 4 >= 5
False
```

(&&)

description: returns the logical conjunction of its two boolean arguments.

usage:

```
Prelude> True && True
True
Prelude> (3 < 4) && (4 < 5) && False
False
```

(||)

description: returns the logical disjunction of its two boolean arguments.

usage:

```
Prelude> True || False
True
Prelude> (3 < 4) || (4 > 5) || False
True
```