

```

# type
> sum :: Num a => [a] -> a
> sum [] = 0
> sum (el:els) = el + sum els

# qsort
> qsort [] = []
qsort (el:els) = qsort small ++ [el] ++ qsort large
  where
  small = [a | a <- els, a <= el]
  large = [a | a <- els, a > el]

# precedence
Functions have precedence over all other operators.

f a (b + 1) means f (a, b + 1)
f a b + 1 means (f a b) + 1
f x * g y means f (x) * g (y)

# types
Bool, Char 'a', String "asd"
Int, Integer, Float

# list
sequence of elements of the same type

['b', 'c'] :: [Char]
[False, True], [True] :: [Bool]

head [1,2,3] = 1
tail [1,2,3] = [2:3]

head :: [a] -> a
tail :: [a] -> [a]

[1,2] : [3,4] = [1,2,3,4]

# tuple
may be of different types

> (False, 'a', 'b') :: (Bool, Char, Char)

# function
mapping from type a to type b

> not :: Bool -> Bool
> isDigit :: Char -> Bool
> add :: (Integer, Integer) -> Integer

# curried functions
functions with several parameters can also return functions as results

> add2 :: Integer -> (Integer -> Integer)
> add2 x y = x + y
> add2 5 :: Integer -> Integer

functions taking arguments one at a time are called curried functions

ex. of partial application

> dropCh :: Int -> [Char] -> [Char]
> dropCh 0 string = string
> dropCh i [] = []
> dropCh i (el:els) = dropChar (i-1) els
> drop3Chars :: [Char] -> [Char]
> drop3Chars = dropCh 3

T1 -> T2 -> T3 means T1 -> (T2 -> T3)
f a b c means ((f a) b) c

# polymorphic types
length :: [a] -> Int
length [1,2]
length "abc"

# overloaded functions
a polymorphic function is overloaded if its type contains one or more class constraints

ex. type a is constrnd' to class Num

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

# basic haskell classes
Eq
  ==, /=
Ord
  <, >, <=, >=
  max, min
Show
  show :: a -> String
Read
  read :: String -> a
Num
  (+), (*), (-) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a
Integral
  quot, rem :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  div, mod :: a -> a -> a
  divMod :: a -> a -> (a, a)
Fractional
  (/) :: a -> a -> a
  recip :: a -> a
Float
RealFrac
  truncate, round :: (Integral b) => a -> b
  ceiling, floor :: (Integral b) => a -> b
  properFract :: (Integral b) => a -> (b, a)

# operators precedence
9 not, negate
  !!
  (!!) :: [a] -> Int -> a
  [1,2,3] !! 1 returns 2
8 ^, ^^, **
7 *, /, 'div', 'quot', 'rem', 'mod'
6 -
5 :, ++
  1:2:[] = 1:[2] = [1,2]
  (++) :: [a] -> [a] -> [a]
4 =, /=, <, <=, >, >=
3 &&
2 ||

# conditional expression
> if e1 then e2 else e3

# defining functions
> abs n = if n >= 0 then n else -n

> signum n = if n > 0 then 1
  else if n == 0 then 0
  else -1

with guarded conditions
> abs n | n >= 0 = n
  | otherwise = -n

> signum n | n > 0 = 1
  | n == 0 = 0
  | otherwise = -1

# pattern matching
a pattern is a mapping between a function's param. and its arg. value

> sumPair (a,b) = a + b
> fst (a,_) = a -- defined in Prelude
> snd (_,b) = b -- ditto

> pairdSums :: (Num a) => [(a,a)] -> a
> pairdSums [] = 0
> pairdSums ((a,b):ls) =
  a + b + pairdSums ls

> matrixSum :: (Num a) => [[a]] -> a
> matrixSum [] = 0
> matrixSum ([:rs] = matrixSum rs
> matrixSum ((c:cs):rs) =
  c + matrixSum (cs:rs)

[(1,x)] is a list pt for [(Int,a)]
((1,x):ls) is a list pt for [(Int,a)]
(("st",x):(y,z):ls) is pt [(Char),a]

# recursion
> factorial n | n == 0 = 1
  | n > 0 = n * factorial (n-1)

> fibo :: Int -> Int
> fibo 0 = 0
> fibo 1 = 1
> fibo n = fibo (n-2) + fibo (n-1)

> drop :: Integral b => b -> [a] -> [a]
> drop 0 xs = xs
> drop (n+1) [] = []
> drop (n+1) (x:xs) = drop n xs

# local environments
> power8 :: (Num a) => a -> a
> power8 x = let
  > square = x * x
  > power4 = square * square
  > in
  > power4 * power4

> sumPaired :: (Num a) => [a] -> (a,a)
> sumPaired [] = (0,0)
> sumPaired [a] = (a,0)
> sumPaired (a:b:ls) = (a+sumOdd,b+sumEven)
> where (sumOdd,sumEven)=sumPaired ls

# list comprehensions
> [x^2 | x <- [1..5]]

> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]

> zip :: [a] -> [b] -> [(a,b)]
> zip [] _ = []
> zip _ [] = []
> zip (x:xs) (y:ys) = (x,y) : zip xs ys

> sorted xs = and
  [x <= y | (x,y) <- zip xs (tail xs)]

now lists compr. with predicates

> [x | x <- [1..10], mod x 2 == 0]
[2,4,6,8,10]

factors :: (Integral a) => a -> [a]
factors n = [x|x<-[1..n], mod n x == 0]

prime :: (Integral a) => a -> Bool
prime n = factors n == [1,n]

```

high-order functions

take a function as an argument or returns a function as a result

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

lambda expressions

nameless function

```
> odds n = map (\x -> x * 2 + 1)
[0..n-1]
```

sections

```
1+2    can be written as    (+) 1 2
(1+) 2  is the same as      (+2) 1
```

```
(1+)    successor function
(1/)    reciprocation function
(*2)    doubling function
(/2)    halving function
```

list processing functions

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

```
> map (+1) [1,3,5,7]
[2,4,6,8]
> map isDigit ['a','1','b']
[False,True,False]
```

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

```
> map (map (+1)) [[1,2,3],[4,5]]
[[2,3,4],[5,6]]
```

```
> filter :: (a -> Bool) -> [a] -> [a]
> filter p [] = []
> filter p (x:xs)
>   | p x = x : filter p xs
>   | otherwise = filter p xs
```

```
> filter even [1..10]
[2,4,6,8,10]
```

```
> filter p xs = [ x | x <- xs, p x ]
```

```
all :: (a -> Bool) -> [a] -> Bool
any :: (a -> Bool) -> [a] -> Bool
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

foldr

```
sum [] = 0
sum (x:xs) = x + sum xs
```

```
product [] = 1
product (x:xs) = x * product xs
```

```
and [] = True
and (x:xs) = x && and xs
```

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

```
> sum = foldr (+) 0
> product = foldr (*) 1
> and = foldr (&&) True
```

```
> foldr (+) 0 [1,2,3]
```

```
> len = foldr (\ _ n -> 1 + n) 0
> len list =
>   foldr (\ _ n -> 1 + n) 0 list
```

now foldl

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

function composition

```
odd n = not (even n)
twice f x = f (f x)
sumsqreven ns =
  sum (map (^2) (filter even ns))
```

```
odd = not . even
twice f = f . f
sumsqreven = sum . map (^2) . filter
even
```

reasoning on lists

naturally uses mathematical induction

show that a property P(xs) holds for all lists xs

```
- base case P([])
- P(xs) => P(x:xs)
```

example: for this,

```
> reverse [] = []
> reverse (x:xs) = reverse xs ++ [x]
```

show that the following property holds:

```
> reverse (xs++ys) =
  reverse ys++reverse xs
```

make substitutions for both cases.

deforestation on lists

eliminate temporary lists

```
> list23 xs = list2 (list3 xs)
> list23' (x:xs) = 6*x : list23' xs
```

removing appends

```
- append. requires traversing the list
- ++ increases complexity
- f a b ++ y becomes f' a b y
```

example

```
> reverse [] = []
> reverse (x:xs) = reverse xs ++ [x]
> reverse' xs y = reverse xs ++ y
```

base case

```
> reverse' [] y = reverse [] ++ y
>               = [] ++ y
>               = y
```

inductive case

```
> reverse' (x:xs) y
  = reverse (x:xs) ++ y
  = reverse xs ++ [x] ++ y
  = reverse xs ++ (x:[]) ++ y
  = reverse xs ++ x : [] ++ y
  = reverse xs ++ (x:y)
  = reverse' xs (x:y)
```

putting everything together

```
reverse' xs = reverse' xs []
  where
    reverse' [] y = y
    reverse' (x:xs) y = reverse' xs (x:y)
```

reducing the number of passes

include intermediate results as parameters

```
> average xs = sum xs / fromInt
(length xs)
```

becomes

```
> average xs = av xs 0 0
  where
    av [] s n = s / fromInt n
    av (x:xs) s n = av xs (s+x) (n+1)
```

Ex: first gets last

```
ex51 :: (Eq a) => [a] -> [a]
ex51 [] = []
ex51 (el:els) = els++[el]
```

Ex: concat lists

```
concatLists :: [[a]] -> [a]
concatLists [] = []
concatLists (x:xs) =
  x ++ concatLists xs
```

Ex: member of list

```
member :: Eq a => a -> [a] -> Bool
member n [] = False
member n (x:xs)
  | n == x = True
  | otherwise = member n xs
```

Ex: split list

```
split :: [a] -> ([a],[a])
split [] = ([],[])
split [a] = ([a],[])
split (x:y:ls) = (x:a,y:b)
  where (a,b) = split ls
```

Ex: replicate

```
rep :: Int -> a -> [a]
rep n a = [ a | x <- [1..n]]
```

Ex. pythagore

```
pyths :: Int -> [(Int,Int,Int)]
pyths n = [(x,y,z) | x <- [1..n],
  y <- [1..n],
  z <- [1..n],
  x*x + y*y == z*z]
```

Ex. scalar product

```
scalarproduct xs ys =
  sum [x * y | (x,y) <- zip xs ys]
```

Ex. list comprehension, map and filter

```
> [f x | x <- xs, p x]
> map f (filter p xs)
```

Ex. all, takeWhile

```
_all :: (a -> Bool) -> [a] -> Bool
_all _ [] = True
_all p (x:xs) =
  p x == True && _all p xs
```

```
_takeWhile :: (a -> Bool) -> [a] -> [a]
_takeWhile _ [] = []
_takeWhile p (el:els)
  | p el = el : _takeWhile p els
  | otherwise = []
```