HASKELL

Function	Type signature	Example usage	Output
even n	a -> Bool	even 12	TRUE
odd n	a -> Bool	odd 12	FALSE
isDigit	Char -> Bool	isDigit ‘1’	FALSE
isUpper	Char -> Bool	isUpper ‘A’	TRUE
succ	Enum a => a -> a	succ ‘a’	‘b'
pred	Enum a => a -> a	pred 5	4
sum	Num a => [a] -> a	sum [1,2,3,4]	10
product	Num a => [a] -> a	product [1,2,3,4]	24
ord	Char -> Int	ord ‘a’	97
chr	Int -> Char	chr 10	\n’
:t	Shows type of expression	Prelude> :t ‘a’	Char
:i	Check class implentation	Prelude> :i Eq	...
flip f x y	(a -> b -> c) -> b -> a -> c	flip (/) 1 2	2 / 1 = 2.0
(++) xs ys	[a] -> [a] -> [a]	[1,2,3] ++ [3,2]	[1,2,3,3,2]
head xs	[a] -> a	head “Hello!”	‘H’
last	[a] -> a	last “Hello!”	‘!’
tail	[a] -> a	tail “Hello!”	“ello!”
length	[a] -> Int	length [1,2,3]	3
map f xs	(a -> b) -> [a] -> [b]	map abs [-1,-3,4]	[1,3,4]
reverse xs	[a] -> [a]	reverse [1..5]	[5,4,3,2,1]
elem x xs	Eq a => a -> [a] -> Bool	elem 1 [1,2,3]	TRUE

Function	Type signature	Example usage	Output
foldl f s xs	(a -> b -> a) -> a -> [b] -> a	foldl (/) 64 [4,2,4]	2.0
foldl1 f xs	(a -> a -> a) -> [a] -> a	foldl1 (/) [64,4,2,8]	1.0
foldr f s xs	(a -> b -> b) -> b -> [a] -> b	foldr (/) 2 [12,24,4]	1.0
foldr1 f xs	(a -> a -> a) -> [a] -> a	foldr1 (/) [12,24,4]	2.0
concat xs	[[a]] -> [a]	concat [[1,2],[3]]	[1,2,3]
concatMap f xs	(a -> [b]) -> [a] -> [b]	concatMap (\x -> x + 2) [1,3,5]	[3,5,7]
any p xs	(a -> Bool) -> [a] -> Bool	any (==1) [1,2,3]	TRUE
all p xs	(a -> Bool) -> [a] -> Bool	all (==1) [1,2,3]	FALSE
maximum	Ord a => [a] -> a	maximum [3,6,4,1,2]	6
minimum	Ord a => [a] -> a	minimum [3,6,4,1,2]	1
scanl f x xs	(a -> b -> a) -> a -> [b] -> a	scanl (/) 6 [3,2]	[6,3,1]
scanl1 f xs	(a -> a -> a) -> [a] -> [a]	scanl1 max [3,6,2]	[3,6,6]
scanr f x xs	(a -> b -> b) -> b -> [a] -> b	scanr (+) 5 [1,3]	[9,8,5]
scanr1 f xs	(a -> a -> a) -> [a] -> [a]	scanr1 (+) [1,2,3]	[6,5,3]
iterate f x	(a -> a) -> a -> [a]	take 3 (iterate (2*) 1)	[1,2,4]
repeat x	a -> [a]	take 3 (repeat ‘A’)	“AAA”
take n xs	Int -> [a] -> [a]	take 2 [1,2,3,4]	[1,2]
drop n xs	Int -> [a] -> [a]	drop 2 [1,2,3,4]	[3,4]
splitAt n xs	Int -> [a] -> ([a],[a])	splitAt 1 “Hi”	(“H”,” i”)
takeWhile p xs	(a -> Bool) -> [a] -> [a]	takeWhile (<3) [1,3,2]	[1]
dropWhile p xs	(a -> Bool) -> [a] -> [a]	dropWhile (<3) [2,5,1]	[5,1]
infixl p f	Sets left infix & precedence	infixl 7 *	...
infixr	Sets right infix & precendence	infixr 0 `(->)`	...

Function	Type signature	Example usage	Output
words xs	String -> [String]	words “a \t b”	[“a”,”b”]
unwords xs	[String] -> String	unwords [“a”,“b”]	“a b”
lookup n dict	Eq a => a -> [(a,b)] -> Maybe b	lookup ‘a’ [(‘a’, 0),(‘b’,1)]	Just 0
filter p xs	(a -> bool) -> [a] -> [a]	filter odd [3,6]	[3]
xs !! i	[a] -> Int -> a	[1,2,3] !! 0	1
sort xs	Ord a => [a] -> [a]	sort “Zvon”	“Znov”
insert n xs	Ord a => a -> [a] -> [a]	insert 2 [1,5]	[1,2,5]
zip xs ys	[a] -> [b] -> [(a,b)]	zip [1,2] [9,8]	[(1,9), (2,8)]
zipWith f xs ys	(a -> b -> c) -> [a] -> [b] -> c	zipWith (+) [1,2] [3,4]	[4,6]
unzip dict	[(a,b)] -> ([a],[b])	unzip [(1,2),(2,3)]	([1,2], [2,3])
maybe x f res	a -> (b -> a) -> Maybe b -> a	maybe 5 negate (lookup 1 [(2,10)]	5
fromJust x	Maybe a -> a	fromJust (Maybe 1)	1
uncurry f x y	((a,b) -> c) -> a -> b -> c	curry fst 2 3	2
show x	Show a => a -> String	show 12	“12”
id x	a -> a	id 3	3

DATA TYPES

data *TypeName TypeVar* = *Constructor TypeVar*
| *Constructor TypeVar*

```
data Maybe a = Nothing | Just a
data List a = Nil | Cons a (List a)
  deriving (Eq, Show)
data Stack a = Empty | Stack [a]
data Tree a = Empty | Node (Tree a) a (Tree a)
data Either a b = Left a | Right b
type Position = (Float, Float)
```

CLASSES & INSTANCES

Class	Uses	Member functions
Eq	Equality testing	/=, ==
Ord	Ordering	>, <, >=, <=
Show	Display strings	show
Enum	Enumeration	e.g. [1..10]
Num	Real and whole (integral) numbers	+, -, *, /, fromInteger

CODING CONVENTIONS

```
power x n
  | n == 0 = 1
  | even n = k * k
  | odd n  = x * k * k
  where k = power x (n `div` 2)

insertionSort [] = []
insertionSort (x : xs)
  = insert x (insertionSort xs)

merge [] [] = []
merge [] (x : xs) = x : xs
merge (x : xs) [] = x : xs
merge xl@(x : xs) yl@(y : ys)
  | x < y = x : merge xs yl
  | otherwise = y : merge xl

mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs
  = merge (mergeSort xs')
(mergeSort xs'')
  where
  (xs', xs'') = splitAt (length
xs `div` 2) xs
```

```
instance Ord (List a) where
  Nil <= _ = True
  (Cons x xs) <= (Cons x' xs')
    | x < x'  = True
    | x > x'  = False
    | x == x' = xs <= xs'
  _ <= _ = False (catch-all)

class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: a -> a

instance Num Exp where
  (+) = Add
  (-) = Sub
  (*) = Mul
  fromInteger
       = Const . fromInteger
```

Force type: `3 * 6 + 7 :: Exp`
returns `Add (Mul (Const 3)…`

```
[(x, y) | x <- [1..3], y <- [1..x]]

let x = 3 in x + 4

ints n = ints' 1 where
  ints' k
    | k > n = []
    | otherwise = k : ints' (k + 1)
ints n = ints' n [] where
  ints' k ks
    | k == 0 = ks
    | otherwise = ints' (k-1) (k:ks)
```

Insert items into an ordered tree

```
insertTree :: Int -> Tree Int -> Tree Int
insertTree n Empty
  = Leaf n
insertTree n (Leaf n)
  | n <= x    = Node (Leaf n) x Empty
  | otherwise = Node Empty x (Leaf n)
insertTree n (Node t1 x t2)
  | n <= x    = Node (insert n t1) x t2
  | otherwise = Node t1 x (insert' n t2)
```

Construct an ordered tree

```
buildTree :: [Int] -> Tree Int
buildTree = foldr insert Empty
```

Flatten a tree (naive)

```
flatten :: Tree a -> [a]
flatten Empty
  = []
flatten (Leaf x)
  = [x]
flatten (Node t1 x t2)
  = flatten t1 ++ (x : flatten t2)
```

Flatten a tree (accumulating parameter)

```
flatten :: Tree a -> [a]
flatten Empty
  = []
flatten tree
  = flatten' tree []
  where
    flatten' Empty acc
      = acc
    flatten' (Node t1 x t2) acc
      = flatten' t1 (x : flatten' t2 acc)
```

Point free notation

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

Extend second argument

```
comp' :: (b -> x -> y) -> (a -> b)
      -> (a -> x -> y)
comp' = (.)
```

Extend first argument

```
comp'' :: (b -> c) -> (a -> x -> b)
       -> (a -> x -> c)
comp'' f g x = f . g x
-- comp'' = (.) . (.)
```