Foundation

This documentation is similar, and partly based on, Haskell's documentation of Prelude.

Constants

```
pi :: Float
```

- The constant π (3.14159265359)

Functional

```
id :: a -> a
```

- Identity function

Composition

```
{.} :: (b -> c) -> (a -> b) -> (a -> c)
```

- Function composition

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

- `flip f` takes its (first) two arguments in the reverse order of `f`.

Boolean

```
not :: Bool -> Bool
```

- Logical NOT

```
{&&} :: Bool -> Bool -> Bool
```

- Logical AND

```
{||} :: Bool -> Bool -> Bool
```

- Logical OR

```
{==} :: a -> a -> Bool
```

- Equality (*built-in*)

```
{/=} :: a -> a -> Bool
```

```
```

- Negated equality

```
{<=} :: a -> a -> Bool
```

- Logical less-than or equals (*built-in*)

```
{<} :: a -> a -> Bool
```

- Logical less-than

```
{>} :: a -> a -> Bool
```

- Logical greater-than

```
{>=} :: a -> a -> Bool
```

- Logical greater-than or equals

Pairs

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

- Pairs two first arguments

```
{<>} :: a -> b -> (a, b)
```

- Infix operator for `asPair`

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

- First element of pair

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

- Second element of pair

Lists

```
{#} :: [a] -> Int -> a
```

- List index

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

```
```

- Length of list

```
empty :: [a] -> Bool
```

- Test for empty list

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

- First element of list

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

- Elements of list, *after* head

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

- Elements of list, *except* head

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

- Last element of list

```
{++} :: [a] -> [a] -> [a]
```

- Join two lists

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

- `map f xs` is the list of results, of applying `f` to each element in `xs`

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

- `fold`, applied to a binary operator, a starting value and a list, reduces the list using the binary operator, from left to right

```
fold1 :: (a -> b -> a) -> [b] -> a
```

- `fold1`, applied to a binary operator and a list, reduces the list using the binary operator, from left to right, with head as starting value

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

- `filter f xs` returns the list of results which holds for the predicate function `f`, where `f` is applied

to each element in `xs`

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

- `take n xs` returns the `n` first elements in `xs`

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

- `drop n xs` returns the elements of `xs`, *after* the `n` first elements

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

- `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

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

- `zipWith` takes a function and two lists, and returns a the function applied to an element of the first list and the corresponding element of the second list. If one input list is short, excess elements of the longer list are discarded

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

- Conjunction of Boolean list

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

- Disjunction of Boolean list

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

- `all f xs` tests if all elements in `xs` hold for predicate functions `f`, where `f` is applied to each element

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

- `any f xs` tests if any of the elements in `xs` hold for predicate functions `f`, where `f` is applied to each element

```
flatten :: [[a]] -> [a]
```

- Concatenates all lists into single list

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

```
```

- `flatMap f xs` applies each list in `xs` to function `f` and flattens the result into single list

```
elem :: a -> [a] -> Bool
```

- `elem x xs` tests if element `x` is in `xs`

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

- Returns list in reverse order

```
union ::  [a] -> [a] -> [a]
```

- Returns the list of all distinct elements in both lists (∪ in set-theory)

```
intersect ::  [a] -> [a] -> [a]
```

- Returns the list of elements which exists in both lists (∩ in set-theory)

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

- Sort list by predicate function

```
sort :: [a] -> [a]
```

- Sort list with quick-sort algorithm

List utilities

```
range :: Int -> Int -> [Int]
```

- `range x y` returns a list of Integer numbers ranging from `x` to `y`

```
rangeStep :: Int -> Int -> Int -> [Int]
```

- `rangeStep x y z` returns a list of Integer numbers ranging from `x`, `x + z`, `x + 2z` up to `y`

Numeric

```
{**} :: Int -> Int -> Int
```

- Exponentiation

```
pow :: Int -> Int -> Int -> Int
```

```
```

- Exponentiation with base

```
sum :: [a] -> a
```

- Sum of elements in list

```
product :: [a] -> a
```

- Product of elements in list

```
min :: a -> a -> a
```

- Smallest of two arguments

```
max :: a -> a -> a
```

- Largest of two arguments

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

- Smallest element in list

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

- Largest element in list

```
even :: Int -> Bool
```

- Test if number is even

```
odd :: Int -> Bool
```

- Test if number is odd

```
negate :: Int -> Int
```

- Negate Integer number

```
negatef :: Float -> Float
```

- Negate Float number

```
abs :: Int -> Int
```

- Absolute value of Integer number

Misc

```
while :: (a -> Bool) -> (a -> a) -> a -> a
```

- `while p f x` applies `f` to `x`, while `p x` is True

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

- `until p f x` applies `f` to `x`, until `p x` becomes False