2.9. list.ak

use aiken/builtin

use aiken/bytearray

use aiken/int

/// Determine if all elements of the list satisfy the given predicate.

///

/// Note: an empty list always satisfies the predicate.

///

/// ```aiken

/// list.all([], fn(n) { n > 0 }) == True

/// list.all([1, 2, 3], fn(n) { n > 0 }) == True

/// list.all([1, 2, 3], fn(n) { n == 2 }) == False

/// ```

pub fn all(self: List<a>, predicate: fn(a) -> Bool) -> Bool {

when self is {

[] -> True

[x, ..xs] -> predicate(x) && all(xs, predicate)

}

}

test all\_1() {

all([1, 2, 3], fn(n) { n > 0 }) == True

}

test all\_2() {

all([1, 2, 3], fn(n) { n > 42 }) == False

}

test all\_3() {

all([], fn(n) { n == 42 }) == True

}

/// Determine if at least one element of the list satisfies the given predicate.

///

/// Note: an empty list never satisfies the predicate.

///

/// ```aiken

/// list.any([], fn(n) { n > 2 }) == False

/// list.any([1, 2, 3], fn(n) { n > 0 }) == True

/// list.any([1, 2, 3], fn(n) { n == 2 }) == True

/// list.any([1, 2, 3], fn(n) { n < 0 }) == False

/// ```

pub fn any(self: List<a>, predicate: fn(a) -> Bool) -> Bool {

when self is {

[] -> False

[x, ..xs] -> predicate(x) || any(xs, predicate)

}

}

test any\_1() {

any([1, 2, 3], fn(n) { n > 0 }) == True

}

test any\_2() {

any([1, 2, 3], fn(n) { n > 42 }) == False

}

test any\_3() {

any([], fn(n) { n == 42 }) == False

}

/// Count how many items in the list satisfy the given predicate.

///

/// ```aiken

/// list.count([], fn(a) { a > 2}) == 0

/// list.count([1, 2, 3], fn(a) { n > 0 }) == 3

/// list.count([1, 2, 3], fn(a) { n >= 2 }) == 2

/// list.count([1, 2, 3], fn(a) { n > 5 }) == 0

/// ```

pub fn count(self: List<a>, predicate: fn(a) -> Bool) -> Int {

foldr(

self,

0,

fn(item, total) {

if predicate(item) {

total + 1

} else {

total

}

},

)

}

test count\_empty() {

count([], fn(a) { a > 2 }) == 0

}

test count\_all() {

count([1, 2, 3], fn(a) { a > 0 }) == 3

}

test count\_some() {

count([1, 2, 3], fn(a) { a >= 2 }) == 2

}

test count\_none() {

count([1, 2, 3], fn(a) { a > 5 }) == 0

}

/// Return Some(item) at the index or None if the index is out of range. The index is 0-based.

///

/// ```aiken

/// list.at([1, 2, 3], 1) == Some(2)

/// list.at([1, 2, 3], 42) == None

/// ```

pub fn at(self: List<a>, index: Int) -> Option<a> {

when self is {

[] -> None

[x, ..xs] ->

if index == 0 {

Some(x)

} else {

at(xs, index - 1)

}

}

}

test at\_1() {

at([1, 2, 3], -1) == None

}

test at\_2() {

at([], 0) == None

}

test at\_3() {

at([1, 2, 3], 3) == None

}

test at\_4() {

at([1], 0) == Some(1)

}

test at\_5() {

at([1, 2, 3], 2) == Some(3)

}

/// Merge two lists together.

///

/// ```aiken

/// list.concat([], []) == []

/// list.concat([], [1, 2, 3]) == [1, 2, 3]

/// list.concat([1, 2, 3], [4, 5, 6]) == [1, 2, 3, 4, 5, 6]

/// ```

pub fn concat(left: List<a>, right: List<a>) -> List<a> {

when left is {

[] -> right

[x, ..xs] ->

[x, ..concat(xs, right)]

}

}

test concat\_1() {

concat([1, 2, 3], [4, 5, 6]) == [1, 2, 3, 4, 5, 6]

}

test concat\_2() {

concat([1, 2, 3], []) == [1, 2, 3]

}

test concat\_3() {

concat([], [1, 2, 3]) == [1, 2, 3]

}

/// Remove the first occurrence of the given element from the list.

///

/// ```aiken

/// list.delete([1, 2, 3, 1], 1) == [2, 3, 1]

/// list.delete([1, 2, 3], 14) == [1, 2, 3]

/// ```

pub fn delete(self: List<a>, elem: a) -> List<a> {

when self is {

[] ->

[]

[x, ..xs] ->

if x == elem {

xs

} else {

[x, ..delete(xs, elem)]

}

}

}

test delete\_1() {

delete([], 42) == []

}

test delete\_2() {

delete([1, 2, 3, 1], 1) == [2, 3, 1]

}

test delete\_3() {

delete([1, 2, 3], 14) == [1, 2, 3]

}

test delete\_4() {

delete([2], 2) == []

}

/// Remove the first occurrence of each element of the second list from the first one.

///

/// ```

/// list.difference(["h", "e", "l", "l", "o"], ["l", "e", "l"]) == ["h", "o"]

/// list.difference([1, 2, 3, 4, 5], [1, 1, 2]) == [3, 4, 5]

/// list.difference([1, 2, 3], []) == [1, 2, 3]

/// ```

pub fn difference(self: List<a>, with: List<a>) -> List<a> {

when with is {

[] -> self

[x, ..xs] -> difference(delete(self, x), xs)

}

}

test difference\_1() {

difference(["h", "e", "l", "l", "o"], ["l", "e", "l"]) == ["h", "o"]

}

test difference\_2() {

difference([1, 2, 3, 4, 5], [1, 1, 2]) == [3, 4, 5]

}

test difference\_3() {

difference([1, 2, 3], []) == [1, 2, 3]

}

test difference\_4() {

difference([], [1, 2, 3]) == []

}

/// Drop the first `n` elements of a list.

///

/// ```aiken

/// list.drop([1, 2, 3], 2) == [3]

/// list.drop([], 42) == []

/// list.drop([1, 2, 3], 42) == []

/// ```

pub fn drop(self: List<a>, n: Int) -> List<a> {

if n <= 0 {

self

} else {

when self is {

[] ->

[]

[\_x, ..xs] -> drop(xs, n - 1)

}

}

}

test drop\_1() {

drop([], 42) == []

}

test drop\_2() {

drop([1, 2, 3], 2) == [3]

}

/// Returns the suffix of the given list after removing all elements that satisfy the predicate.

///

/// ```aiken

/// list.drop\_while([1, 2, 3], fn(x) { x < 2 }) == [2, 3]

/// list.drop\_while([], fn(x) { x > 2 }) == []

/// list.drop\_while([1, 2, 3], fn(x) { x == 3 }) == [1, 2, 3]

/// ```

pub fn drop\_while(self: List<a>, predicate: fn(a) -> Bool) -> List<a> {

when self is {

[] ->

[]

[x, ..xs] ->

if predicate(x) {

drop\_while(xs, predicate)

} else {

self

}

}

}

test drop\_while\_1() {

drop\_while([], fn(x) { x > 2 }) == []

}

test drop\_while\_2() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

drop\_while(xs, fn(x) { x > 5 }) == [5, 4, 3, 2, 1]

}

test drop\_while\_3() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

drop\_while(xs, fn(x) { x == 42 }) == xs

}

test drop\_while\_4() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

drop\_while(xs, fn(x) { x < 42 }) == []

}

/// Produce a list of elements that satisfy a predicate.

///

/// ```aiken

/// list.filter([1, 2, 3], fn(x) { x >= 2 }) == [2, 3]

/// list.filter([], fn(x) { x > 2 }) == []

/// list.filter([1, 2, 3], fn(x) { x == 3 }) == [3]

/// ```

pub fn filter(self: List<a>, predicate: fn(a) -> Bool) -> List<a> {

when self is {

[] ->

[]

[x, ..xs] ->

if predicate(x) {

[x, ..filter(xs, predicate)]

} else {

filter(xs, predicate)

}

}

}

test filter\_1() {

filter([], fn(x) { x > 0 }) == []

}

test filter\_2() {

let xs =

[1, 2, 3, 4, 5, 6]

filter(xs, fn(x) { builtin.mod\_integer(x, 2) == 0 }) == [2, 4, 6]

}

test filter\_3() {

let filter\_foldr =

fn(xs, f) {

foldr(

xs,

[],

fn(x, ys) {

if f(x) {

[x, ..ys]

} else {

ys

}

},

)

}

let is\_odd =

fn(n) { builtin.mod\_integer(n, 2) != 0 }

filter\_foldr([1, 2, 3], is\_odd) == filter([1, 2, 3], is\_odd)

}

/// Produce a list of transformed elements that satisfy a predicate.

///

/// ```aiken

/// let transform = fn(x) { if x % 2 == 0 { None } else { Some(3\*x) } }

/// list.filter\_map([1, 2, 3], transform) == [3, 9]

/// ```

pub fn filter\_map(self: List<a>, predicate: fn(a) -> Option<b>) -> List<b> {

when self is {

[] ->

[]

[x, ..xs] ->

when predicate(x) is {

None -> filter\_map(xs, predicate)

Some(y) ->

[y, ..filter\_map(xs, predicate)]

}

}

}

test filter\_map\_1() {

filter\_map([], fn(\_) { Some(42) }) == []

}

test filter\_map\_2() {

filter\_map(

[1, 2, 3, 4, 5, 6],

fn(x) {

if builtin.mod\_integer(x, 2) != 0 {

Some(3 \* x)

} else {

None

}

},

) == [3, 9, 15]

}

/// Find the first element satisfying the given predicate, if any.

///

/// ```aiken

/// list.find([1, 2, 3], fn(x) { x == 2 }) == Some(2)

/// list.find([4, 5, 6], fn(x) { x == 2 }) == None

/// ```

pub fn find(self: List<a>, predicate: fn(a) -> Bool) -> Option<a> {

when self is {

[] -> None

[x, ..xs] ->

if predicate(x) {

Some(x)

} else {

find(xs, predicate)

}

}

}

test find\_1() {

find([1, 2, 3], fn(x) { x == 1 }) == Some(1)

}

test find\_2() {

find([1, 2, 3], fn(x) { x > 42 }) == None

}

test find\_3() {

find([], fn(\_) { True }) == None

}

/// Map elements of a list into a new list and flatten the result.

///

/// ```aiken

/// list.flat\_map([1, 2, 3], fn(a) { [a, 2\*a] }) == [1, 2, 2, 4, 3, 6]

/// ```

pub fn flat\_map(self: List<a>, with: fn(a) -> List<b>) -> List<b> {

foldr(self, [], fn(x, xs) { concat(with(x), xs) })

}

test flat\_map\_1() {

flat\_map([], fn(a) { [a] }) == []

}

test flat\_map\_2() {

flat\_map([1, 2, 3], fn(a) { [a, a] }) == [1, 1, 2, 2, 3, 3]

}

/// Reduce a list from left to right.

///

/// ```aiken

/// list.foldl([1, 2, 3], 0, fn(n, total) { n + total }) == 6

/// list.foldl([1, 2, 3], [], fn(x, xs) { [x, ..xs] }) == [3, 2, 1]

/// ```

pub fn foldl(self: List<a>, zero: b, with: fn(a, b) -> b) -> b {

when self is {

[] -> zero

[x, ..xs] -> foldl(xs, with(x, zero), with)

}

}

test foldl\_1() {

foldl([], 0, fn(\_, \_) { 1 }) == 0

}

test foldl\_2() {

foldl([1, 2, 3, 4, 5], 0, fn(n, total) { n + total }) == 15

}

test foldl\_3() {

foldl([1, 2, 3, 4], [], fn(x, xs) { [x, ..xs] }) == [4, 3, 2, 1]

}

/// Reduce a list from right to left.

///

/// ```aiken

/// list.foldr([1, 2, 3], 0, fn(n, total) { n + total }) == 6

/// list.foldr([1, 2, 3], [], fn(x, xs) { [x, ..xs] }) == [1, 2, 3]

/// ```

pub fn foldr(self: List<a>, zero: b, with: fn(a, b) -> b) -> b {

when self is {

[] -> zero

[x, ..xs] -> with(x, foldr(xs, zero, with))

}

}

test foldr\_1() {

foldr([1, 2, 3, 4, 5], 0, fn(n, total) { n + total }) == 15

}

test foldr\_2() {

foldr(

[1, 2, 3],

"",

fn(n, \_str) {

if builtin.mod\_integer(n, 2) == 0 {

"foo"

} else {

"bar"

}

},

) == "bar"

}

test foldr\_3() {

foldr([1, 2, 3, 4], [], fn(x, xs) { [x, ..xs] }) == [1, 2, 3, 4]

}

/// Return all elements except the last one.

///

/// ```aiken

/// list.init([]) == None

/// list.init([1, 2, 3]) == Some([1, 2])

/// ```

pub fn init(self: List<a>) -> Option<List<a>> {

when self is {

[] -> None

\_ -> Some(do\_init(self))

}

}

fn do\_init(self: List<a>) -> List<a> {

when self is {

[] -> fail @"unreachable"

[\_] ->

[]

[x, ..xs] ->

[x, ..do\_init(xs)]

}

}

test init\_1() {

init([]) == None

}

test init\_2() {

init([1]) == Some([])

}

test init\_3() {

init([1, 2, 3, 4]) == Some([1, 2, 3])

}

/// Figures out whether a list contain the given element.

///

/// ```aiken

/// list.has([1, 2, 3], 2) == True

/// list.has([1, 2, 3], 14) == False

/// list.has([], 14) == False

/// ```

pub fn has(self: List<a>, elem: a) -> Bool {

when self is {

[] -> False

[x, ..xs] ->

if x == elem {

True

} else {

has(xs, elem)

}

}

}

test has\_1() {

has([1, 2, 3], 1) == True

}

test has\_2() {

has([1, 2, 3], 14) == False

}

test has\_3() {

has([], 14) == False

}

/// Gets the index of an element of a list, if any. Otherwise, returns None.

///

/// ```aiken

/// list.index\_of([1, 5, 2], 2) == Some(2)

/// list.index\_of([1, 7, 3], 4) == None

/// list.index\_of([1, 0, 9, 6], 6) == 3

/// list.index\_of([], 6) == None

/// ```

pub fn index\_of(self: List<a>, elem: a) -> Option<Int> {

do\_index\_of(self, elem, 0)

}

fn do\_index\_of(self: List<a>, elem: a, i: Int) -> Option<Int> {

when self is {

[] -> None

[x, ..xs] ->

if x == elem {

Some(i)

} else {

do\_index\_of(xs, elem, i + 1)

}

}

}

test index\_of\_1() {

index\_of([1, 5, 2], 2) == Some(2)

}

test index\_of\_2() {

index\_of([1, 7, 3], 4) == None

}

test index\_of\_3() {

index\_of([1, 0, 9, 6], 6) == Some(3)

}

test index\_of\_4() {

index\_of([], 6) == None

}

/// Get the first element of a list

///

/// ```aiken

/// list.head([1, 2, 3]) == Some(1)

/// list.head([]) == None

/// ```

pub fn head(self: List<a>) -> Option<a> {

when self is {

[] -> None

\_ -> Some(builtin.head\_list(self))

}

}

test head\_1() {

head([1, 2, 3]) == Some(1)

}

test head\_2() {

head([]) == None

}

/// Like [`foldr`](#foldr), but also provides the position (0-based) of the elements when iterating.

///

/// ```aiken

/// let group = fn(i, x, xs) { [(i, x), ..xs] }

/// list.indexed\_foldr(["a", "b", "c"], [], group) == [

/// (0, "a"),

/// (1, "b"),

/// (2, "c")

/// ]

/// ```

pub fn indexed\_foldr(

self: List<a>,

zero: result,

with: fn(Int, a, result) -> result,

) -> result {

do\_indexed\_foldr(0, self, zero, with)

}

fn do\_indexed\_foldr(

n: Int,

self: List<a>,

zero: result,

with: fn(Int, a, result) -> result,

) -> result {

when self is {

[] -> zero

[x, ..xs] -> with(n, x, do\_indexed\_foldr(n + 1, xs, zero, with))

}

}

test indexed\_foldr\_1() {

indexed\_foldr([], 0, fn(i, x, xs) { i + x + xs }) == 0

}

test indexed\_foldr\_2() {

let letters =

["a", "b", "c"]

indexed\_foldr(letters, [], fn(i, x, xs) { [(i, x), ..xs] }) == [

(0, "a"),

(1, "b"),

(2, "c"),

]

}

/// List [`map`](#map) but provides the position (0-based) of the elements while iterating.

///

/// ```aiken

/// list.indexed\_map([1, 2, 3], fn(i, x) { i + x }) == [1, 3, 5]

/// ```

pub fn indexed\_map(self: List<a>, with: fn(Int, a) -> result) -> List<result> {

do\_indexed\_map(0, self, with)

}

fn do\_indexed\_map(

n: Int,

self: List<a>,

with: fn(Int, a) -> result,

) -> List<result> {

when self is {

[] ->

[]

[x, ..xs] ->

[with(n, x), ..do\_indexed\_map(n + 1, xs, with)]

}

}

test indexed\_map\_1() {

indexed\_map([], fn(i, \_n) { i }) == []

}

test indexed\_map\_2() {

indexed\_map(

[4, 8, 13, 2],

fn(i, n) {

if n == 8 {

n

} else {

i

}

},

) == [0, 8, 2, 3]

}

/// Checks whether a list is empty.

///

/// ```aiken

/// list.is\_empty([]) == True

/// list.is\_empty([1, 2, 3]) == False

/// ```

pub fn is\_empty(self: List<a>) -> Bool {

when self is {

[] -> True

\_ -> False

}

}

test is\_empty\_1() {

is\_empty([]) == True

}

test is\_empty\_2() {

is\_empty([1, 2, 3]) == False

}

/// Get the last in the given list, if any.

///

/// ```aiken

/// list.last([]) == None

/// list.last([1, 2, 3]) == Some(3)

/// ```

pub fn last(self: List<a>) -> Option<a> {

when self is {

[] -> None

[x] -> Some(x)

[\_, ..xs] -> last(xs)

}

}

test last\_1() {

last([]) == None

}

test last\_2() {

last([1]) == Some(1)

}

test last\_3() {

last([1, 2, 3, 4]) == Some(4)

}

/// Get the number of elements in the given list.

///

/// ```aiken

/// list.length([]) == 0

/// list.length([1, 2, 3]) == 3

/// ```

pub fn length(self: List<a>) -> Int {

when self is {

[] -> 0

[\_, ..xs] -> 1 + length(xs)

}

}

test length\_1() {

length([]) == 0

}

test length\_2() {

length([1, 2, 3]) == 3

}

/// Apply a function to each element of a list.

///

/// ```aiken

/// list.map([1, 2, 3, 4], fn(n) { n + 1 }) == [2, 3, 4, 5]

/// ```

pub fn map(self: List<a>, with: fn(a) -> result) -> List<result> {

when self is {

[] ->

[]

[x, ..xs] ->

[with(x), ..map(xs, with)]

}

}

test map\_1() {

map([], fn(n) { n + 1 }) == []

}

test map\_2() {

map([1, 2, 3, 4], fn(n) { n + 1 }) == [2, 3, 4, 5]

}

/// Apply a function of two arguments, combining elements from two lists.

///

/// Note: if one list is longer, the extra elements are dropped.

///

/// ```aiken

/// list.map2([1, 2, 3], [1, 2], fn(a, b) { a + b }) == [2, 4]

/// ```

pub fn map2(

self: List<a>,

bs: List<b>,

with: fn(a, b) -> result,

) -> List<result> {

when self is {

[] ->

[]

[x, ..xs] ->

when bs is {

[] ->

[]

[y, ..ys] ->

[with(x, y), ..map2(xs, ys, with)]

}

}

}

test map2\_1() {

map2([], [1, 2, 3], fn(a, b) { a + b }) == []

}

test map2\_2() {

map2([1, 2, 3], [1, 2], fn(a, b) { a + b }) == [2, 4]

}

test map2\_3() {

map2([42], [1, 2, 3], fn(\_a, b) { Some(b) }) == [Some(1)]

}

/// Apply a function of three arguments, combining elements from three lists.

///

/// Note: if one list is longer, the extra elements are dropped.

///

/// ```aiken

/// list.map3([1, 2, 3], [1, 2], [1, 2, 3], fn(a, b, c) { a + b + c }) == [3, 6]

/// ```

pub fn map3(

self: List<a>,

bs: List<b>,

cs: List<c>,

with: fn(a, b, c) -> result,

) -> List<result> {

when self is {

[] ->

[]

[x, ..xs] ->

when bs is {

[] ->

[]

[y, ..ys] ->

when cs is {

[] ->

[]

[z, ..zs] ->

[with(x, y, z), ..map3(xs, ys, zs, with)]

}

}

}

}

test map3\_1() {

map3([], [], [1, 2, 3], fn(a, b, c) { a + b + c }) == []

}

test map3\_2() {

map3([1, 2, 3], [1, 2], [1, 2, 3], fn(a, b, c) { a + b + c }) == [3, 6]

}

/// Add an element in front of the list. Sometimes useful when combined with

/// other functions.

///

/// ```aiken

/// list.push([2, 3], 1) == [1, ..[2, 3]] == [1, 2, 3]

/// ```

pub fn push(self: List<a>, elem: a) -> List<a> {

[elem, ..self]

}

test push\_1() {

push([], 0) == [0]

}

test push\_2() {

push([2, 3], 1) == [1, 2, 3]

}

/// Construct a list of a integer from a given range.

///

/// ```aiken

/// list.range(0, 3) == [0, 1, 2, 3]

/// list.range(-1, 1) == [-1, 0, 1]

/// ```

pub fn range(from: Int, to: Int) -> List<Int> {

if from > to {

[]

} else {

[from, ..range(from + 1, to)]

}

}

test range\_1() {

range(0, 3) == [0, 1, 2, 3]

}

test range\_2() {

range(-1, 1) == [-1, 0, 1]

}

/// Construct a list filled with n copies of a value.

///

/// ```aiken

/// list.repeat("na", 3) == ["na", "na", "na"]

/// ```

pub fn repeat(elem: a, n\_times: Int) -> List<a> {

if n\_times <= 0 {

[]

} else {

[elem, ..repeat(elem, n\_times - 1)]

}

}

test repeat\_1() {

repeat(42, 0) == []

}

test repeat\_2() {

repeat(14, 3) == [14, 14, 14]

}

/// Return the list with its elements in the reserve order.

///

/// ```aiken

/// list.reverse([1, 2, 3]) == [3, 2, 1]

/// ```

pub fn reverse(self: List<a>) -> List<a> {

foldl(self, [], fn(x, xs) { [x, ..xs] })

}

test reverse\_1() {

reverse([]) == []

}

test reverse\_2() {

reverse([1, 2, 3]) == [3, 2, 1]

}

/// Returns a tuple with all elements that satisfy the predicate at first

/// element, and the rest as second element.

///

/// ```aiken

/// list.partition([1, 2, 3, 4], fn(x) { x % 2 == 0 }) == ([2, 4], [1, 3])

/// ```

pub fn partition(self: List<a>, predicate: fn(a) -> Bool) -> (List<a>, List<a>) {

when self is {

[] -> ([], [])

[x, ..xs] -> {

let (left, right) = partition(xs, predicate)

if predicate(x) {

([x, ..left], right)

} else {

(left, [x, ..right])

}

}

}

}

test partition\_1() {

partition([], fn(x) { x > 2 }) == ([], [])

}

test partition\_2() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

partition(xs, fn(x) { x > 5 }) == ([10, 9, 8, 7, 6], [5, 4, 3, 2, 1])

}

test partition\_3() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

partition(xs, fn(x) { x == 42 }) == ([], xs)

}

test partition\_4() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

partition(xs, fn(x) { x < 42 }) == (xs, [])

}

test partition\_5() {

partition([1, 2, 3, 4], fn(x) { x % 2 == 0 }) == ([2, 4], [1, 3])

}

/// Extract a sublist from the given list using 0-based indexes. Negative

/// indexes wrap over, so `-1` refers to the last element of the list.

///

/// ```aiken

/// list.slice([1, 2, 3, 4, 5, 6], from: 2, to: 4) == [3, 4, 5]

/// list.slice([1, 2, 3, 4, 5, 6], from: -2, to: -1) == [5, 6]

/// list.slice([1, 2, 3, 4, 5, 6], from: 1, to: -1) == [2, 3, 4, 5, 6]

/// ```

pub fn slice(self: List<a>, from: Int, to: Int) {

let (i, l) =

if from >= 0 {

(from, None)

} else {

let l = length(self)

(l + from, Some(l))

}

let j =

if to >= 0 {

to - i + 1

} else {

when l is {

Some(l) -> l + to - i + 1

None -> length(self) + to - i + 1

}

}

self

|> drop(i)

|> take(j)

}

test slice\_1() {

slice([1, 2, 3], 0, 2) == [1, 2, 3]

}

test slice\_2() {

slice([1, 2, 3, 4, 5, 6], from: 2, to: 4) == [3, 4, 5]

}

test slice\_3() {

slice([1, 2, 3, 4, 5, 6], from: -2, to: -1) == [5, 6]

}

test slice\_4() {

slice([1, 2, 3, 4, 5, 6], from: 1, to: -1) == [2, 3, 4, 5, 6]

}

test slice\_5() {

slice([1, 2, 3, 4, 5, 6], from: -4, to: -3) == [3, 4]

}

test slice\_6() {

slice([1, 2, 3, 4, 5, 6], from: -2, to: 1) == []

}

/// Sort a list in ascending order using the given comparison function.

///

/// ```aiken

/// use aiken/int

///

/// sort([3, 1, 4, 0, 2], int.compare) == [0, 1, 2, 3, 4]

/// sort([1, 2, 3], int.compare) == [1, 2, 3]

/// ```

pub fn sort(self: List<a>, compare: fn(a, a) -> Ordering) -> List<a> {

when self is {

[] ->

[]

[x, ..xs] -> insert(sort(xs, compare), x, compare)

}

}

fn insert(self: List<a>, e: a, compare: fn(a, a) -> Ordering) -> List<a> {

when self is {

[] ->

[e]

[x, ..xs] ->

if compare(e, x) == Less {

[e, ..self]

} else {

[x, ..insert(xs, e, compare)]

}

}

}

test sort\_1() {

let xs =

[6, 7, 5, 4, 1, 3, 9, 8, 0, 2]

sort(xs, int.compare) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

}

test sort\_2() {

let xs =

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sort(xs, int.compare) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

}

test sort\_3() {

let xs =

[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

sort(xs, int.compare) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

}

test sort\_4() {

sort([], int.compare) == []

}

/// Cut a list in two, such that the first list contains the given number of /

/// elements and the second list contains the rest.

///

/// Fundamentally equivalent to (but more efficient):

///

/// ```aiken

/// // span(xs, n) == (take(xs, n), drop(xs, n))

/// span([1, 2, 3, 4, 5], 3) == ([1, 2, 3], [4, 5])

/// ```

pub fn span(self: List<a>, n: Int) -> (List<a>, List<a>) {

when self is {

[] -> ([], [])

[x, ..xs] ->

if n <= 0 {

([], self)

} else {

let (left, right) = span(xs, n - 1)

([x, ..left], right)

}

}

}

test span\_1() {

span([], 2) == ([], [])

}

test span\_2() {

span([1, 2, 3], 2) == ([1, 2], [3])

}

test span\_3() {

span([1, 2, 3], -1) == ([], [1, 2, 3])

}

test span\_4() {

span([1, 2, 3], 42) == ([1, 2, 3], [])

}

/// Get elements of a list after the first one, if any.

///

/// ```aiken

/// list.tail([]) == None

/// list.tail([1, 2, 3]) == Some([2, 3])

/// ```

pub fn tail(self: List<a>) -> Option<List<a>> {

when self is {

[] -> None

[\_, ..xs] -> Some(xs)

}

}

test tail\_1() {

tail([1, 2, 3]) == Some([2, 3])

}

test tail\_2() {

tail([]) == None

}

/// Get the first `n` elements of a list.

///

/// ```aiken

/// list.take([1, 2, 3], 2) == [1, 2]

/// list.take([1, 2, 3], 14) == [1, 2, 3]

/// ```

pub fn take(self: List<a>, n: Int) -> List<a> {

if n <= 0 {

[]

} else {

when self is {

[] ->

[]

[x, ..xs] ->

[x, ..take(xs, n - 1)]

}

}

}

test take\_1() {

take([], 42) == []

}

test take\_2() {

take([1, 2, 3], 2) == [1, 2]

}

/// Returns the longest prefix of the given list where all elements satisfy the predicate.

///

/// ```aiken

/// list.take\_while([1, 2, 3], fn(x) { x > 2 }) == []

/// list.take\_while([1, 2, 3], fn(x) { x < 2 }) == [1]

/// ```

pub fn take\_while(self: List<a>, predicate: fn(a) -> Bool) -> List<a> {

when self is {

[] ->

[]

[x, ..xs] ->

if predicate(x) {

[x, ..take\_while(xs, predicate)]

} else {

[]

}

}

}

test take\_while\_1() {

take\_while([], fn(x) { x > 2 }) == []

}

test take\_while\_2() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

take\_while(xs, fn(x) { x > 5 }) == [10, 9, 8, 7, 6]

}

test take\_while\_3() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

take\_while(xs, fn(x) { x == 42 }) == []

}

test take\_while\_4() {

let xs =

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

take\_while(xs, fn(x) { x < 42 }) == xs

}

/// Removes duplicate elements from a list.

///

/// ```aiken

/// list.unique([1, 2, 3, 1]) == [1, 2, 3]

/// ```

pub fn unique(self: List<a>) -> List<a> {

when self is {

[] ->

[]

[x, ..xs] ->

[x, ..unique(filter(xs, fn(y) { y != x }))]

}

}

test unique\_1() {

unique([]) == []

}

test unique\_2() {

let xs =

[1, 2, 3, 1, 1, 3, 4, 1, 2, 3, 2, 4, 5, 6, 7, 8, 9, 10, 9]

unique(xs) == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

}

/// Decompose a list of tuples into a tuple of lists.

///

/// ```

/// list.unzip([(1, "a"), (2, "b")]) == ([1, 2], ["a", "b"])

/// ```

pub fn unzip(self: List<(a, b)>) -> (List<a>, List<b>) {

when self is {

[] -> ([], [])

[(a, b), ..xs] -> {

let (a\_tail, b\_tail) = unzip(xs)

([a, ..a\_tail], [b, ..b\_tail])

}

}

}

test unzip\_1() {

unzip([]) == ([], [])

}

test unzip\_2() {

unzip([(1, "a"), (2, "b")]) == ([1, 2], ["a", "b"])

}

/// Combine two lists together.

///

/// Note: if one list is longer, the extra elements are dropped.

///

/// ```aiken

/// list.zip([1, 2], ["a", "b", "c"]) == [(1, "a"), (2, "b")]

/// ```

pub fn zip(self: List<a>, bs: List<b>) -> List<(a, b)> {

when self is {

[] ->

[]

[x, ..xs] ->

when bs is {

[] ->

[]

[y, ..ys] ->

[(x, y), ..zip(xs, ys)]

}

}

}

test zip\_1() {

zip([], [1, 2, 3]) == []

}

test zip\_2() {

zip([1, 2, 3], []) == []

}

test zip\_3() {

zip([1, 2], ["a", "b", "c"]) == [(1, "a"), (2, "b")]

}

/// Reduce a list from left to right using the accumulator as left operand.

/// Said differently, this is [`foldl`](#foldl) with callback arguments swapped.

///

/// ```aiken

/// list.reduce([#[1], #[2], #[3]], #[0], bytearray.concat) == #[0, 1, 2, 3]

/// list.reduce([True, False, True], False, fn(b, a) { or { b, a } }) == True

/// ```

pub fn reduce(self: List<a>, zero: b, with: fn(b, a) -> b) -> b {

foldl(self, zero, flip(with))

}

test reduce\_1() {

reduce([], 0, fn(n, total) { n + total }) == 0

}

test reduce\_2() {

reduce([1, 2, 3], 0, fn(n, total) { n + total }) == 6

}

test reduce\_3() {

reduce([True, False, True], False, fn(left, right) { left || right }) == True

}

test reduce\_4() {

reduce(

[#[1], #[2], #[3]],

#[9],

fn(left, right) { bytearray.concat(left, right) },

) == #[9, 1, 2, 3]

}