2.11. option.ak

/// Picks the first element which is not None. If there's no such element, return None.

///

/// ```aiken

/// option.choice([]) == None

/// option.choice([Some(14), Some(42)]) == Some(14)

/// option.choice([None, Some(42)]) == Some(42)

/// option.choice([None, None]) == None

/// ```

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

when self is {

[] -> None

[head, ..others] ->

when head is {

None -> choice(others)

\_ -> head

}

}

}

test choice\_1() {

Some(1) == choice([Some(1), Some(2)])

}

test choice\_2() {

None == choice([])

}

test choice\_3() {

Some(1) == choice([None, Some(1)])

}

/// Provide a default value, turning an optional value into a normal value.

///

/// ```aiken

/// option.or\_else(None, "aiken") == "aiken"

/// option.or\_else(Some(42), 14) == 42

/// ```

pub fn or\_else(self: Option<a>, default: a) -> a {

when self is {

None -> default

Some(a) -> a

}

}

test or\_else\_1() {

or\_else(None, "aiken") == "aiken"

}

test or\_else\_2() {

or\_else(Some(42), 14) == 42

}

/// Like [`or\_else`](#or\_else) but allows returning an `Option`.

/// This is effectively mapping the error branch.

///

/// ```aiken

/// option.or\_try(None, fn(\_) { Some("aiken") }) == Some("aiken")

/// option.or\_try(Some(42), fn(\_) { Some(14) }) == Some(42)

/// ```

pub fn or\_try(self: Option<a>, compute\_default: fn() -> Option<a>) -> Option<a> {

when self is {

None -> compute\_default()

\_ -> self

}

}

test or\_try\_1() {

or\_try(None, fn() { Some("aiken") }) == Some("aiken")

}

test or\_try\_2() {

or\_try(Some(42), fn() { fail }) == Some(42)

}

/// Apply a function to the inner value of an [`Option`](#option)

///

/// ```aiken

/// option.map(None, fn(n) { n \* 2 }) == None

/// option.map(Some(14), fn(n) { n \* 2 }) == Some(28)

/// ```

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

when self is {

None -> None

Some(a) -> Some(with(a))

}

}

test map\_1() {

map(None, fn(\_) { Void }) == None

}

test map\_2() {

map(Some(14), fn(n) { n + 1 }) == Some(15)

}

/// Combine two [`Option`](#option) together.

///

/// ```aiken

/// type Foo {

/// Foo(Int, Int)

/// }

///

/// option.map2(Some(14), Some(42), Foo) == Some(Foo(14, 42))

/// option.map2(None, Some(42), Foo) == None

/// option.map2(Some(14), None, Foo) == None

/// ```

pub fn map2(

opt\_a: Option<a>,

opt\_b: Option<b>,

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

) -> Option<result> {

when opt\_a is {

None -> None

Some(a) ->

when opt\_b is {

None -> None

Some(b) -> Some(with(a, b))

}

}

}

test map2\_1() {

map2(None, Some(42), fn(\_, \_) { 14 }) == None

}

test map2\_2() {

map2(Some(42), None, fn(\_, \_) { 14 }) == None

}

test map2\_3() {

map2(Some(14), Some(42), fn(a, b) { (a, b) }) == Some((14, 42))

}

/// Combine three [`Option`](#option) together.

///

/// ```aiken

/// type Foo {

/// Foo(Int, Int, Int)

/// }

///

/// option.map3(Some(14), Some(42), Some(1337), Foo) == Some(Foo(14, 42, 1337))

/// option.map3(None, Some(42), Some(1337), Foo) == None

/// option.map3(Some(14), None, None, Foo) == None

/// ```

pub fn map3(

opt\_a: Option<a>,

opt\_b: Option<b>,

opt\_c: Option<c>,

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

) -> Option<result> {

when opt\_a is {

None -> None

Some(a) ->

when opt\_b is {

None -> None

Some(b) ->

when opt\_c is {

None -> None

Some(c) -> Some(with(a, b, c))

}

}

}

}

test map3\_1() {

map3(None, Some(42), None, fn(\_, \_, \_) { 14 }) == None

}

test map3\_2() {

map3(Some(42), None, None, fn(\_, \_, \_) { 14 }) == None

}

test map3\_3() {

map3(Some(14), Some(42), Some(1337), fn(a, b, c) { c - a + b }) == Some(1365)

}

/// Chain together many computations that may fail.

///

/// ```aiken

/// self

/// |> dict.get(policy\_id)

/// |> option.and\_then(dict.get(\_, asset\_name))

/// |> option.or\_else(0)

/// ```

pub fn and\_then(

self: Option<a>,

then: fn(a) -> Option<result>,

) -> Option<result> {

when self is {

None -> None

Some(a) -> then(a)

}

}

fn try\_decrement(n: Int) -> Option<Int> {

if n > 0 {

Some(n - 1)

} else {

None

}

}

test and\_then\_1() {

let result =

None

|> and\_then(try\_decrement)

result == None

}

test and\_then\_2() {

let result =

Some(14)

|> and\_then(try\_decrement)

result == Some(13)

}

test and\_then\_3() {

let result =

Some(0)

|> and\_then(try\_decrement)

result == None

}

/// Converts from `Option<Option<a>>` to `Option<a>`.

///

/// ```aiken

/// option.flatten(Some(Some(42))) == Some(42)

/// option.flatten(Some(None)) == None

/// option.flatten(None) == None

/// ```

///

/// Flattening only removes one level of nesting at a time:

///

/// ```aiken

/// flatten(Some(Some(Some(42)))) == Some(Some(42))

/// Some(Some(Some(42))) |> flatten |> flatten == Some(42)

/// ```

pub fn flatten(opt: Option<Option<a>>) -> Option<a> {

when opt is {

Some(inner) -> inner

None -> None

}

}

test flatten\_1() {

let x: Option<Option<Int>> = Some(Some(6))

Some(6) == flatten(x)

}

test flatten\_2() {

let x: Option<Option<Int>> = Some(None)

None == flatten(x)

}

test flatten\_3() {

let x: Option<Option<Int>> = None

None == flatten(x)

}

test flatten\_4() {

let x: Option<Option<Option<Int>>> = Some(Some(Some(6)))

let result =

x

|> flatten

|> flatten

Some(6) == result

}

/// Asserts whether an option is `Some`, irrespective of the value it contains.

pub fn is\_some(self: Option<a>) -> Bool {

when self is {

Some(\_) -> True

\_ -> False

}

}

test is\_some\_1() {

is\_some(Some(0)) == True

}

test is\_some\_2() {

is\_some(None) == False

}

/// Asserts whether an option is `None`.

pub fn is\_none(self: Option<a>) -> Bool {

when self is {

Some(\_) -> False

\_ -> True

}

}

test is\_none\_1() {

is\_none(Some(0)) == False

}

test is\_none\_2() {

is\_none(None) == True

}