2.3. bytearray.ak

use aiken/builtin

use aiken/math

use aiken/option

/// Compare two bytearrays lexicographically.

///

/// ```aiken

/// bytearray.compare(#"00", #"FF") == Less

/// bytearray.compare(#"42", #"42") == Equal

/// bytearray.compare(#"FF", #"00") == Greater

/// ```

pub fn compare(left: ByteArray, right: ByteArray) -> Ordering {

if builtin.less\_than\_bytearray(left, right) {

Less

} else if builtin.equals\_bytearray(left, right) {

Equal

} else {

Greater

}

}

/// Combine two `ByteArray` together.

///

/// ```aiken

/// bytearray.concat(left: #[1, 2, 3], right: #[4, 5, 6]) == #[1, 2, 3, 4, 5, 6]

/// ```

pub fn concat(left: ByteArray, right: ByteArray) -> ByteArray {

builtin.append\_bytearray(left, right)

}

test concat\_1() {

concat(#"", #"") == #""

}

test concat\_2() {

concat(#"", #"01") == #"01"

}

test concat\_3() {

concat(#"0102", #"") == #"0102"

}

test concat\_4() {

concat(#"0102", #"0304") == #"01020304"

}

/// Returns the suffix of a `ByteArray` after `n` elements.

///

/// ```aiken

/// bytearray.drop(#[1, 2, 3], n: 2) == #[3]

/// ```

pub fn drop(self: ByteArray, n: Int) -> ByteArray {

builtin.slice\_bytearray(n, builtin.length\_of\_bytearray(self) - n, self)

}

test drop\_1() {

let x = #"01020304050607"

drop(x, 2) == #"0304050607"

}

test drop\_2() {

let x = #"01020304050607"

drop(x, 0) == x

}

test drop\_3() {

let x = #"01"

drop(x, 1) == #""

}

test drop\_4() {

let x = #""

drop(x, 2) == #""

}

/// Left-fold over bytes of a [`ByteArray`](https://aiken-lang.github.io/prelude/aiken.html#ByteArray). Note that every byte given to the callback function is comprised between 0 and 255.

///

/// ```aiken

/// bytearray.foldl(#"acab", 0, fn(byte, acc) { acc \* 256 + byte }) == 44203

/// bytearray.foldl(#[1, 2, 3], #"", flip(bytearray.push)) == #[3, 2, 1]

/// ```

pub fn foldl(

self: ByteArray,

zero: result,

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

) -> result {

do\_foldl(self, zero, builtin.length\_of\_bytearray(self), 0, with)

}

fn do\_foldl(

self: ByteArray,

zero: result,

len: Int,

cursor: Int,

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

) -> result {

if cursor == len {

zero

} else {

do\_foldl(

self,

with(builtin.index\_bytearray(self, cursor), zero),

len,

cursor + 1,

with,

)

}

}

test foldl\_1() {

foldl(#[], 42, fn(byte, acc) { byte + acc }) == 42

}

test foldl\_2() {

foldl(#"acab", 0, fn(byte, acc) { acc \* 256 + byte }) == 44203

}

test foldl\_3() {

foldl(

#"356cf088720a169dae0ce0bb1df8588944389fa43322f0d6ef4ed8c069bfd405",

0,

fn(byte, acc) { acc \* 256 + byte },

) == 24165060555594911913195642527692216679757672038384202527929620681761931383813

}

test foldl\_4() {

foldl(#[1, 2, 3, 4, 5], #"", flip(push)) == #[5, 4, 3, 2, 1]

}

/// Right-fold over bytes of a [`ByteArray`](https://aiken-lang.github.io/prelude/aiken.html#ByteArray). Note that every byte given to the callback function is comprised between 0 and 255.

///

/// ```aiken

/// bytearray.foldr(#"acab", 0, fn(byte, acc) { acc \* 256 + byte }) == 43948

/// bytearray.foldl(#[1, 2, 3], #"", flip(bytearray.push)) == #[1, 2, 3]

/// ```

pub fn foldr(

self: ByteArray,

zero: result,

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

) -> result {

do\_foldr(self, zero, builtin.length\_of\_bytearray(self) - 1, with)

}

fn do\_foldr(

self: ByteArray,

zero: result,

cursor: Int,

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

) -> result {

if cursor < 0 {

zero

} else {

do\_foldr(

self,

with(builtin.index\_bytearray(self, cursor), zero),

cursor - 1,

with,

)

}

}

test foldr\_1() {

foldr(#[], 42, fn(byte, acc) { byte + acc }) == 42

}

test foldr\_2() {

foldr(#"acab", 0, fn(byte, acc) { acc \* 256 + byte }) == 43948

}

test foldr\_3() {

foldr(#[1, 2, 3, 4, 5], #"", flip(push)) == #[1, 2, 3, 4, 5]

}

/// Search the start and end positions of a sub-array in a `ByteArray`.

///

/// ```aiken

/// bytearray.index\_of("Hello, World!", "World") == Some((7, 11))

/// bytearray.index\_of("Hello, World!", "foo") == None

/// bytearray.index\_of("Hello, World!", "!") == Some((12, 12))

/// bytearray.index\_of("Hello, World!", "o") == Some((4, 4))

/// bytearray.index\_of("Hello, World!", "Hello, World!") == Some((0, 12))

/// ```

pub fn index\_of(self: ByteArray, bytes: ByteArray) -> Option<(Int, Int)> {

let offset = length(bytes)

do\_index\_of(self, bytes, 0, offset, length(self))

|> option.map(fn(ix) { (ix, ix + offset - 1) })

}

fn do\_index\_of(

self: ByteArray,

bytes: ByteArray,

cursor: Int,

offset: Int,

size: Int,

) -> Option<Int> {

if cursor + offset > size {

None

} else {

if builtin.slice\_bytearray(cursor, offset, self) == bytes {

Some(cursor)

} else {

do\_index\_of(self, bytes, cursor + 1, offset, size)

}

}

}

test index\_of\_1() {

index\_of("Hello, World!", "World") == Some((7, 11))

}

test index\_of\_2() {

index\_of("Hello, World!", "foo") == None

}

test index\_of\_3() {

index\_of("Hello, World!", "!") == Some((12, 12))

}

test index\_of\_4() {

index\_of("Hello, World!", "o") == Some((4, 4))

}

test index\_of\_5() {

index\_of("Hello, World!", "Hello, World!") == Some((0, 12))

}

/// Returns the number of bytes in a `ByteArray`.

///

/// ```aiken

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

/// ```

pub fn length(self: ByteArray) -> Int {

builtin.length\_of\_bytearray(self)

}

test length\_1() {

length(#"") == 0

}

test length\_2() {

length(#"010203") == 3

}

/// Returns `True` when the given `ByteArray` is empty.

///

/// ```aiken

/// bytearray.is\_empty(#"") == True

/// bytearray.is\_empty(#"00ff") == False

/// ```

pub fn is\_empty(self: ByteArray) -> Bool {

builtin.length\_of\_bytearray(self) == 0

}

test is\_empty\_1() {

is\_empty(#"") == True

}

test is\_empty\_2() {

is\_empty(#"01") == False

}

/// Convert a `String` into a `ByteArray`.

///

/// ```aiken

/// bytearray.from\_string(@"ABC") == #"414243"

/// ```

pub fn from\_string(str: String) -> ByteArray {

builtin.encode\_utf8(str)

}

test from\_string\_1() {

from\_string(@"") == ""

}

test from\_string\_2() {

from\_string(@"ABC") == #"414243"

}

/// Add a byte element in front of a `ByteArray`. When the given byte is

/// greater than 255, it wraps-around. \*\*PlutusV2 behavior\*\* So 256 is mapped to 0, 257 to 1, and so

/// forth.

/// In PlutusV3 this will error instead of wrapping around.

///

/// ```aiken

/// bytearray.push(#"", 0) == #"00"

/// bytearray.push(#"0203", 1) == #"010203"

/// bytearray.push(#"0203", 257) == #"010203"

/// ```

pub fn push(self: ByteArray, byte: Int) -> ByteArray {

builtin.cons\_bytearray(byte, self)

}

test push\_1() {

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

}

test push\_2() {

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

}

test push\_3() {

let x = 257

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

}

/// Reduce bytes in a ByteArray from left to right using the accumulator as left operand.

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

///

/// ```aiken

/// bytearray.reduce(#[1,2,3], #[], bytearray.push) == #[3, 2, 1]

/// ```

pub fn reduce(

self: ByteArray,

zero: result,

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

) -> result {

foldl(self, zero, flip(with))

}

test reduce\_1() {

reduce(#[], #[], push) == #[]

}

test reduce\_2() {

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

}

/// Extract a `ByteArray` as a slice of another `ByteArray`.

///

/// Indexes are 0-based and inclusive.

///

/// ```aiken

/// bytearray.slice(#[0, 1, 2, 3, 4, 5, 6], start: 1, end: 3) == #[1, 2, 3]

/// ```

pub fn slice(self: ByteArray, start: Int, end: Int) -> ByteArray {

builtin.slice\_bytearray(start, end - start + 1, self)

}

test slice\_1() {

slice(#"", 1, 2) == #""

}

test slice\_2() {

slice(#"010203", 1, 2) == #"0203"

}

test slice\_3() {

slice(#"010203", 0, 42) == #"010203"

}

test slice\_4() {

slice(#[0, 1, 2, 3, 4], 0, 3) == #[0, 1, 2, 3]

}

test slice\_5() {

slice(#[0, 1, 2, 3, 4], 1, 2) == #[1, 2]

}

/// Returns the n-length prefix of a `ByteArray`.

///

/// ```aiken

/// bytearray.take(#[1, 2, 3], n: 2) == #[1, 2]

/// ```

pub fn take(self: ByteArray, n: Int) -> ByteArray {

builtin.slice\_bytearray(0, n, self)

}

test take\_1() {

let x = #"01020304050607"

take(x, 2) == #"0102"

}

test take\_2() {

let x = #"01020304050607"

take(x, 0) == #""

}

test take\_3() {

let x = #"01"

take(x, 1) == x

}

test take\_4() {

let x = #"010203"

take(x, 0) == #""

}

/// Convert a `ByteArray` into a `String`.

///

/// <br/>⚠️<pre>WARNING</pre> | This functions fails if the underlying `ByteArray` isn't UTF-8-encoded. <br/>In particular, you cannot convert arbitrary hash digests using this function. <br/>For converting arbitrary `ByteArray`s, use [bytearray.to\_hex](#to\_hex).

/// --- | ---

///

///

/// ```aiken

/// bytearray.to\_string(#"414243") == "ABC"

///

/// bytearray.to\_string(some\_hash) -> fail

/// ```

pub fn to\_string(self: ByteArray) -> String {

builtin.decode\_utf8(self)

}

test to\_string\_1() {

to\_string("") == @""

}

test to\_string\_2() {

to\_string("ABC") == @"ABC"

}

/// Encode a `ByteArray` as a hexidecimal `String`.

///

/// ```aiken

/// use aiken/bytearray

///

/// bytearray.to\_hex("Hello world!") == @"48656c6c6f20776f726c6421"

/// ```

pub fn to\_hex(self: ByteArray) -> String {

self

|> encode\_base16(builtin.length\_of\_bytearray(self) - 1, "")

|> builtin.decode\_utf8

}

// Construct an hex string in reverse order, from the back. The 'builder' is an

// accumulator. It works fast because `index\_bytearray` follows a constant-time cost

// model

fn encode\_base16(bytes: ByteArray, ix: Int, builder: ByteArray) -> ByteArray {

if ix < 0 {

builder

} else {

let byte = builtin.index\_bytearray(bytes, ix)

let msb = byte / 16

let lsb = byte % 16

encode\_base16(

bytes,

ix - 1,

builtin.cons\_bytearray(

msb + if msb < 10 {

48

} else {

87

},

builtin.cons\_bytearray(

lsb + if lsb < 10 {

48

} else {

87

},

builder,

),

),

)

}

}

test to\_hex\_1() {

to\_hex("Hello world!") == @"48656c6c6f20776f726c6421"

}

test to\_hex\_2() {

to\_hex("The quick brown fox jumps over the lazy dog") == @"54686520717569636b2062726f776e20666f78206a756d7073206f76657220746865206c617a7920646f67"

}

/// Checks whether a bit (Most-Significant-Bit first) is set in the given 'ByteArray'.

///

/// For example, consider the following bytearray: `#"8b765f"`. It can also be written as the

/// following bits sequence:

///

/// `8` | `b` | `7` | `6` | `5` | `f`

/// --- | --- | --- | --- | --- | ---

/// `1000` | `1011` | `0111` | `0110` | `0101` | `1111`

///

/// And thus, we have:

///

/// ```aiken

/// test\_bit(#"8b765f", 0) == True

/// test\_bit(#"8b765f", 1) == False

/// test\_bit(#"8b765f", 2) == False

/// test\_bit(#"8b765f", 3) == False

/// test\_bit(#"8b765f", 7) == True

/// test\_bit(#"8b765f", 8) == False

/// test\_bit(#"8b765f", 20) == True

/// test\_bit(#"8b765f", 21) == True

/// test\_bit(#"8b765f", 22) == True

/// test\_bit(#"8b765f", 23) == True

/// ```

pub fn test\_bit(self: ByteArray, ix: Int) -> Bool {

builtin.less\_than\_equals\_bytearray(

#[128],

builtin.cons\_bytearray(

builtin.index\_bytearray(self, ix / 8) \* math.pow2(ix % 8),

"",

),

)

}

test test\_bit\_0() {

test\_bit(#"8b765f", 0)

}

test test\_bit\_1() {

!test\_bit(#"8b765f", 1)

}

test test\_bit\_2() {

!test\_bit(#"8b765f", 2)

}

test test\_bit\_3() {

!test\_bit(#"8b765f", 3)

}

test test\_bit\_7() {

test\_bit(#"8b765f", 7)

}

test test\_bit\_8() {

!test\_bit(#"8b765f", 8)

}

test test\_bit\_20\_21\_22\_23() {

and {

test\_bit(#"8b765f", 20),

test\_bit(#"8b765f", 21),

test\_bit(#"8b765f", 22),

test\_bit(#"8b765f", 23),

}

}