Array To Zipper

Rob Napier

Array

Dictionary

String

Set

SequenceType

CollectionType

Stride

Slice

Generator

ZipGenerator2

MapSequenceGenerator

BidirectionalReverseView

LazyBidirectionalCollection

ExtendedGraphemeClusterLiteralConvertible

Stdlib includes the obvious

```swift
/// Conceptually_, `Array` is an efficient, tail-growable random-access
/// collection of arbitrary elements.
struct Array { ... }
```

Alongside the obscure

```swift
/// Useful mainly when the optimizer's ability to specialize generics
/// outstrips its ability to specialize ordinary closures.
protocol Sink { ... }
```

Generators

Generators

- Encapsulate iteration state
- Provide the next element if there is one

```
protocol GeneratorType {
    typealias Element
    mutating func next() -> Element?
}
```

```swift
struct MyEmptyGenerator<Element>: GeneratorType {
    mutating func next() -> Element? {
        return nil
    }
}

var e = MyEmptyGenerator<Int>()
e.next() // => nil
e.next() // => nil
```

Generators are mutable so no else has to be.

```swift
struct MyGeneratorOfOne<T> : GeneratorType {
    var element: T?
    init(_ element: T?) { self.element = element }

    mutating func next() -> T? {
        let result = self.element
        self.element = nil
        return result
    }
}
```

Requires: `next()` has not been applied to a copy of `self` since the copy was made, and no preceding call to `self.next()` has returned `nil`. Specific implementations of this protocol are encouraged to respond to violations of this requirement by calling `preconditionFailure("...")`.

```swift
struct MyGeneratorOfOne<T> : GeneratorType {
    var element: T?
    var done = false
    init(_ element: T?) { self.element = element }

    mutating func next() -> T? {
        precondition(!done, "Generator exhausted")
        let result = self.element
        self.element = nil
        self.done = (result == nil)
        return result
    }
}
```

```swift
struct NaturalGenerator : GeneratorType {
    var n = 0
    mutating func next() -> Int? {
        return n++
    }
}

var nats = NaturalGenerator()
```

```swift
struct RandomGenerator : GeneratorType {
    let limit: UInt32
    var n: Int

    mutating func next() -> UInt32? {
        if n > 0 {
            --n
            return arc4random_uniform(limit)
        }
        return nil
    }
    init(n: Int, limit: UInt32) {
        self.limit = limit
        self.n = n
    }
}
```

Generic Generators

```swift
struct NaturalGenerator : GeneratorType {
    var n = 0
    mutating func next() -> Int? {
        return n++
    }
}
var nats = NaturalGenerator()
```

```swift
struct NaturalGenerator : GeneratorType {
    var n = 0
    mutating func next() -> Int? {
        return n++
    }
}
var nats = NaturalGenerator()
```

⬇

```swift
var n = 0
var nats = GeneratorOf{ return n++ }
```

```swift
struct GeneratorOf<T> : GeneratorType, SequenceType {
    init(_ nextElement: () -> T?)
    init<G : GeneratorType where G.Element == T>(_ base: G)
...
}
```

```swift
func makeNaturalGenerator() -> GeneratorOf<Int> {
    var n = 0
    return GeneratorOf{ return n++ }
}

func makeNaturalGenerator() -> GeneratorOf<Int> {
    return GeneratorOf(NaturalGenerator())
}
```

Sequences

Sequences

- Can be iterated by `for...in`
- Wraps a Generator

```
protocol SequenceType {
    typealias Generator : GeneratorType
    func generate() -> Generator
}
```

Danger ahead

```
func withoutMinMax<Seq: SequenceType
    where Seq.Generator.Element : Comparable>
    (xs: Seq) -> [Seq.Generator.Element]{

        let mn = minElement(xs)
        let mx = maxElement(xs)

        return filter(xs) { $0 != mn && $0 != mx }
}
```

```swift
struct MyEmptySequence<T> : SequenceType {
    func generate() -> EmptyGenerator<T> {
        return EmptyGenerator()
    }
}

for x in MyEmptySequence<Int>() {
    fatalError("This should never run")
}
```

```swift
struct NaturalSequence : SequenceType {
    func generate() -> GeneratorOf<Int> {
        var n = 0
        return GeneratorOf{ n++ }
    }
}
let nats = NaturalSequence()
```

```swift
let nats = SequenceOf { () -> GeneratorOf<Int> in
    var n = 0
    return GeneratorOf { return n++ }
}
```

Impossible

```
// Drop some elements and return Seq
func drop<Seq: SequenceType>(n: Int, xs: Seq) -> Seq { ... }
```

Possible

```
// Drop some elements and return SequenceOf
func myDrop<Seq: SequenceType>(n: Int, xs: Seq)
    -> SequenceOf<Seq.Generator.Element> {
        var g = xs.generate()
        for _ in 1...n { g.next() }
        return SequenceOf{g}
}
```

Infinite sequences

```
func myCount<Seq: SequenceType>(xs: Seq) -> Int {
    return reduce(xs, 0) { (n, _) in n + 1 }
}
```

Infinite sequences

```
func myCount<Seq: SequenceType>(xs: Seq) -> Int {
    var n = 0
    for _ in xs { ++n }
    return n
}
```

```
func underestimateCount<T : SequenceType>(x: T) -> Int
```

Collections

Collections

- Conforms to SequenceType
- Multipass
- Efficient subscript using some index
- Iterates in subscript order

```
protocol CollectionType : SequenceType {
    typealias Index : ForwardIndexType

    var startIndex: Index { get }
    var endIndex: Index { get }

    subscript (position: Self.Index) -> Self.Generator.Element { get }
}
```

```swift
struct Random {
    subscript (n : UInt32) -> UInt32 {
        return arc4random_uniform(n)
    }
}

let r = Random()
r[100]  // 51
r[1000] // 872
r[100]  // 10
```

Start with Sequence

```
struct MyEmptySequence<T> : SequenceType {
    func generate() -> EmptyGenerator<T> {
        return EmptyGenerator()
    }
}
```

Upgrade to Collection

```
struct MyEmptyCollection<T> : CollectionType {
    func generate() -> EmptyGenerator<T> {
        return EmptyGenerator()
    }
}
```

Index

```
struct MyEmptyCollection<T> : CollectionType {
    func generate() -> EmptyGenerator<T> {
        return EmptyGenerator()
    }
    typealias Index = Int
}
```

Start and end

```
struct MyEmptyCollection<T> : CollectionType {
    func generate() -> EmptyGenerator<T> {
        return EmptyGenerator()
    }
    typealias Index = Int
    let startIndex = 0
    let endIndex = 0
}
```

Subscript

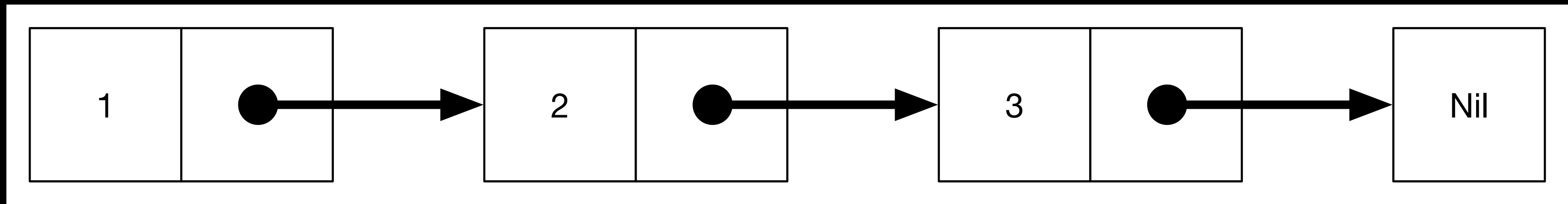
```
struct MyEmptyCollection<T> : CollectionType {
    func generate() -> EmptyGenerator<T> {
        return EmptyGenerator()
    }
    typealias Index = Int
    let startIndex = 0
    let endIndex = 0
    subscript (position: Index) -> T {
        fatalError("Out of bounds")
    }
}
```

Linked List

```swift
final class Box<T> {
    let value: T
    init(_ value: T) { self.value = value }
}

enum List<T> {
    case Cons(Box<T>, Box<List>)
    case Nil
    init(_ first: T, _ rest: List<T>) {
        self = Cons(Box(first), Box(rest))
    }
}

let l = List(1, List(2, List(3, .Nil)))
```



Some helpers

```
extension List {
    func first() -> T? {
        switch self {
        case let Cons(first, _): return first.value
        case Nil: return nil
        }
    }

    func rest() -> List<T> {
        switch self {
        case let Cons(_, rest): return rest.value
        case Nil: return .Nil
        }
    }
}
```

```
extension List : SequenceType {
    func generate() -> GeneratorOf<T> {
        var node = self
        return GeneratorOf {
            let result = node.first()
            node = node.rest()
            return result
        }
    }
}
```

Index design

- Store iteration state: Cursor
- Efficient access

Types of Index

- `ForwardIndexType` ➡ Can only move forward
- `BidirectionalIndexType` ➡ Can move forward or backward
- `RandomAccessIndexType` ➡ Can jump to any index

Integer Subscripting

Not a good List index

```
extension List {
    subscript (i: Int) -> T? {
        return self.nth(i).first()
    }
    func nth (i: Int) -> List {
        var node = self
        for _ in 0 ..< i { node = node.rest() }
        return node
    }
}
```

Indexes

- Need to remember the current iteration location
- Must be able to increment
- Must be able to compare for equality

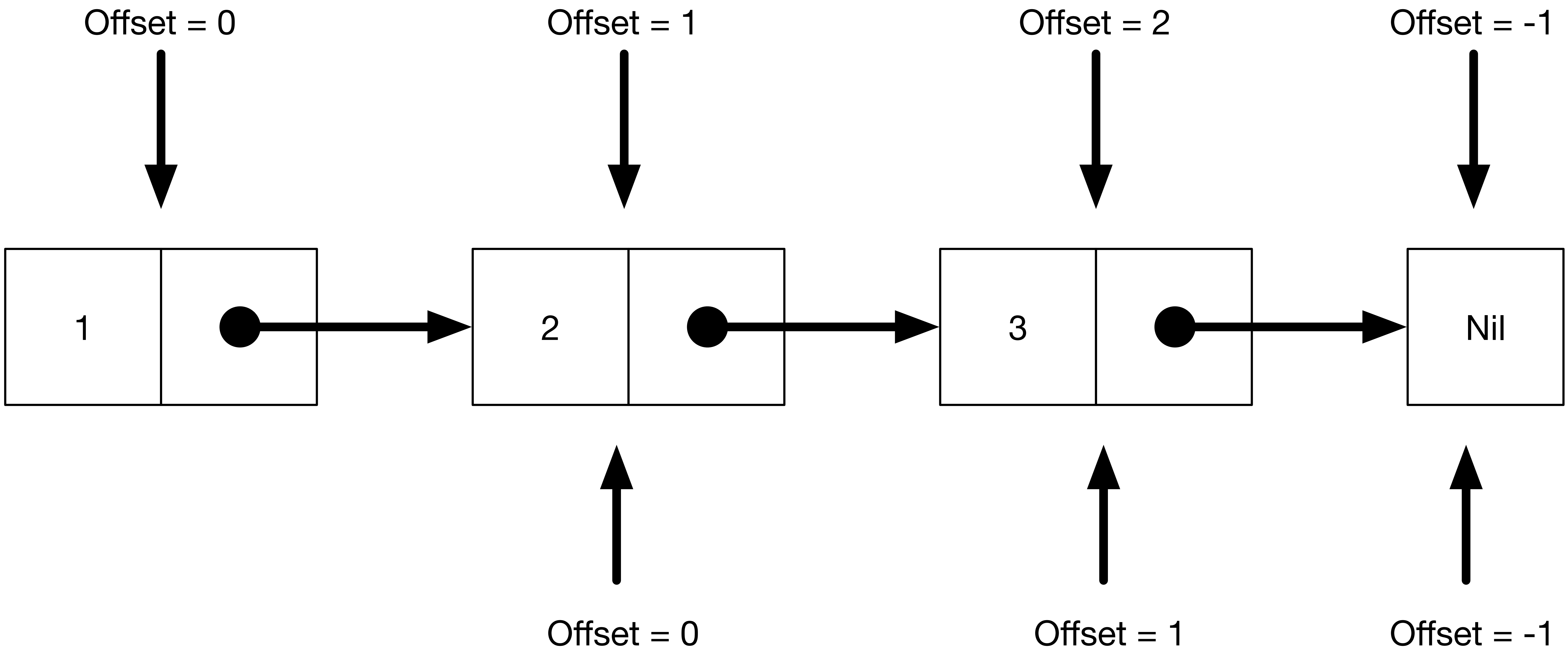
A proper List index

```
struct ListIndex<T> {
    static var End: ListIndex<T> {
        return ListIndex(node: .Nil, offset: -1)
    }
    static func Start(list: List<T>) -> ListIndex<T> {
        return ListIndex(node: list, offset: 0)
    }

    let node: List<T>
    let offset: Int
}
```

```swift
extension ListIndex : ForwardIndexType {
    func successor() -> ListIndex {
        let rest = self.node.rest()
        switch rest {
        case .Cons:
            return ListIndex(node: rest, index: self.index + 1)
        case .Nil:
             return .End
        }
    }
}

func == <T>(lhs: ListIndex<T>, rhs: ListIndex<T>) -> Bool {
    return lhs.index == rhs.index
}
```


Offset = 0
Offset = 1
Offset = 2
Offset = -1
1
2
3
Nil
Offset = 0
Offset = 1
Offset = -1

Now make it a Collection

```
extension List : CollectionType {
    typealias Index = ListIndex<T>
    var startIndex: Index { return .Start(self) }
    var endIndex: Index { return .End }

    subscript (i: Index) -> T {
        return i.node.first()!
    }
}
```

Simplify generate()

```
extension List : SequenceType {
    func generate() -> GeneratorOf<T> {
        var node = self
        return GeneratorOf {
            let result = node.first()
            node = node.rest()
            return result
        }
    }
}
```

Simplify generate()

```
extension List : SequenceType {
    func generate() -> IndexingGenerator<List> {
        return IndexingGenerator(self)
    }
}
```

We get a bunch of helpers

```
find
first
isEmpty
count
```

But can we drop yet?

```
func dropFirst<Seq : Sliceable>(s: Seq) -> Seq.SubSlice
```

-- Nope

Sliceable

- Inherits from CollectionType
- A sub-range of elements can be efficiently extracted
- Slices should be temporary
- Type of SubSlice may be different from collection

Unenforceable Sliceable Requirements

- SubSlice should be Sliceable
- SubSlice should have same element type as collection

Slices in Swift

- String ➡ String
- Array ➡ ArraySlice

https://devforums.apple.com/message/1105132
("Slice, Sliceable")

```swift
struct ListSlice<T> {
    let list: List<T>
    let bounds: Range<List<T>.Index>
    func first() -> T? { return self.startIndex.node.first() }
    func rest() -> List<T> { return self.startIndex.node.rest() }
}
```

Slices are Collections

```
extension ListSlice : CollectionType {
    typealias Index = List<T>.Index
    var startIndex: Index { return self.bounds.startIndex }
    var endIndex: Index { return self.bounds.endIndex }
    subscript (i: Index) -> T { return i.node.first()! }
    func generate() -> IndexingGenerator<ListSlice> {
        return IndexingGenerator(self)
    }
}
```

Slices are Sliceable

```
extension ListSlice : Sliceable {
    typealias SubSlice = ListSlice<T>
    subscript (bounds: Range<Index>) -> SubSlice {
        return ListSlice(list: self.list, bounds: bounds)
    }
}
```

And finally… a sliceable list

```
extension List : Sliceable {
    typealias SubSlice = ListSlice<T>
    subscript (bounds: Range<Index>) -> SubSlice {
        return ListSlice(list: self, bounds: bounds)
    }
}

dropFirst(myList).first() // Second element
```

GeneratorSequence

```
for x in seq { ... }

for x in GeneratorSequence(g) { ... }

for x in SequenceOf({g}) { ... }
```

PermutationGenerator

```
let xs = [3, 1, 4, 1, 5, 9, 2, 6, 5]
let orderedxs = PermutationGenerator(
    elements: xs,
    indices: indices(xs))
```

PermutationGenerator

```
let xs = [3, 1, 4, 1, 5, 9, 2, 6, 5]
let orderedxs = PermutationGenerator(
    elements: xs,
    indices: 0..<xs.count)
```

Just the evens

```
let evenxs = PermutationGenerator(
    elements: xs,
    indices: stride(from: 0, to: xs.count, by: 2))
```

Reverse

```
let reversexs = PermutationGenerator(
    elements: xs,
    indices: reverse(indices(xs)))

// let reversexs = reverse(xs)
```

Concrete Types

Array

- Predictable performance for "normal" usage
- Transparent interoperability with NSArray
- Local-mutation / Non-sharing

Predictable performance for "normal" usage

- "Normal" is like C++ std::vector
  (not Haskell, Scala, ObjC, ...)
- Subscript is O(1)
- Append is O(1), prepend/insert is O(N)

Transparent interoperability

- Sometimes Array is really NSArray
- Sometimes it isn't
- Sometimes it can convert without copy
- Sometimes it can't

ContiguousArray

- Promises to really, really have Swift Array performance
- Loses bridging to ObjC

Local-mutation / Non-sharing

- Swift encourages local mutation
- Swift discourages shared mutable state (sort-of)

Local-mutation / Non-sharing (Performance)

```swift
// O(N)
func addOne(xs: [Int]) -> [Int] {
    return xs + [1]
}

// O(1)
func addOne(inout xs: [Int]) {
    xs.append(1)
}
```

```swift
// O(N^2)
func ones(n: Int) -> [Int] {
    if n == 0 { return [] }
    return [1] + ones(n - 1)
}

// O(N^2) (but currently faster)
reduce(1...n, [Int]()) { (a, _) in a + [1] }

// O(N)
func ones(n: Int) -> [Int] {
    var result: [Int] = []
    for _ in 1...n { result.append(1) }
    return result
}
```

Swift tries to make efficient code more beautiful rather than beautiful code more efficient.

— *Rob's current gut feeling*

Dictionaries

```
struct Dictionary<Key : Hashable, Value> : CollectionType {
    typealias Element = (Key, Value)
    typealias Index = DictionaryIndex<Key, Value>

    subscript(position: DictionaryIndex<Key, Value>) -> (Key, Value) { get }
    subscript(key: Key) -> Value?
}
```

Strings

```
extension String : CollectionType {
    struct Index : BidirectionalIndexType, Comparable, Reflectable {
        subscript (i: String.Index) -> Character { get }
    }
}
```

Ranges and Intervals

```
/// A collection of consecutive discrete index values.
struct Range<T : ForwardIndexType> : Equatable, CollectionType {}

/// A half-open `IntervalType`, which contains its `start` but not its
/// `end`.  Can represent an empty interval.
struct HalfOpenInterval<T : Comparable> : IntervalType, Equatable {}

/// A closed `IntervalType`, which contains both its `start` and its
/// `end`.  Cannot represent an empty interval.
struct ClosedInterval<T : Comparable> : IntervalType, Equatable {}
```

Range

```
/// A collection of consecutive discrete index values.
struct Range<T : ForwardIndexType> : Equatable, CollectionType {}

Range(start: 1, end: 6) // 1, 2, 3, 4, 5
```

Intervals

```
/// A half-open `IntervalType`, which contains its `start` but not its
/// `end`.  Can represent an empty interval.
struct HalfOpenInterval<T : Comparable> : IntervalType, Equatable {}

/// A closed `IntervalType`, which contains both its `start` and its
/// `end`.  Cannot represent an empty interval.
struct ClosedInterval<T : Comparable> : IntervalType, Equatable {}
```

```
HalfOpenInterval(1.0, 6.0) // [1.0, 6.0)
ClosedInterval(1.0, 6.0)   // [1.0, 6.0]

Range(start: 1.0, end: 6.0) // Error
```

"Which function does Swift call?"

http://airspeedvelocity.net/2014/09/20/which-func/

Breaking Swift's brain

```
let ranges = [
    1...2,
    1...2,
    1...2,
    1...2,
    1...2,
    1...2,
    1...2,
    1...2,
    1...2,
    1...2,
]
```

Giving Swift a break

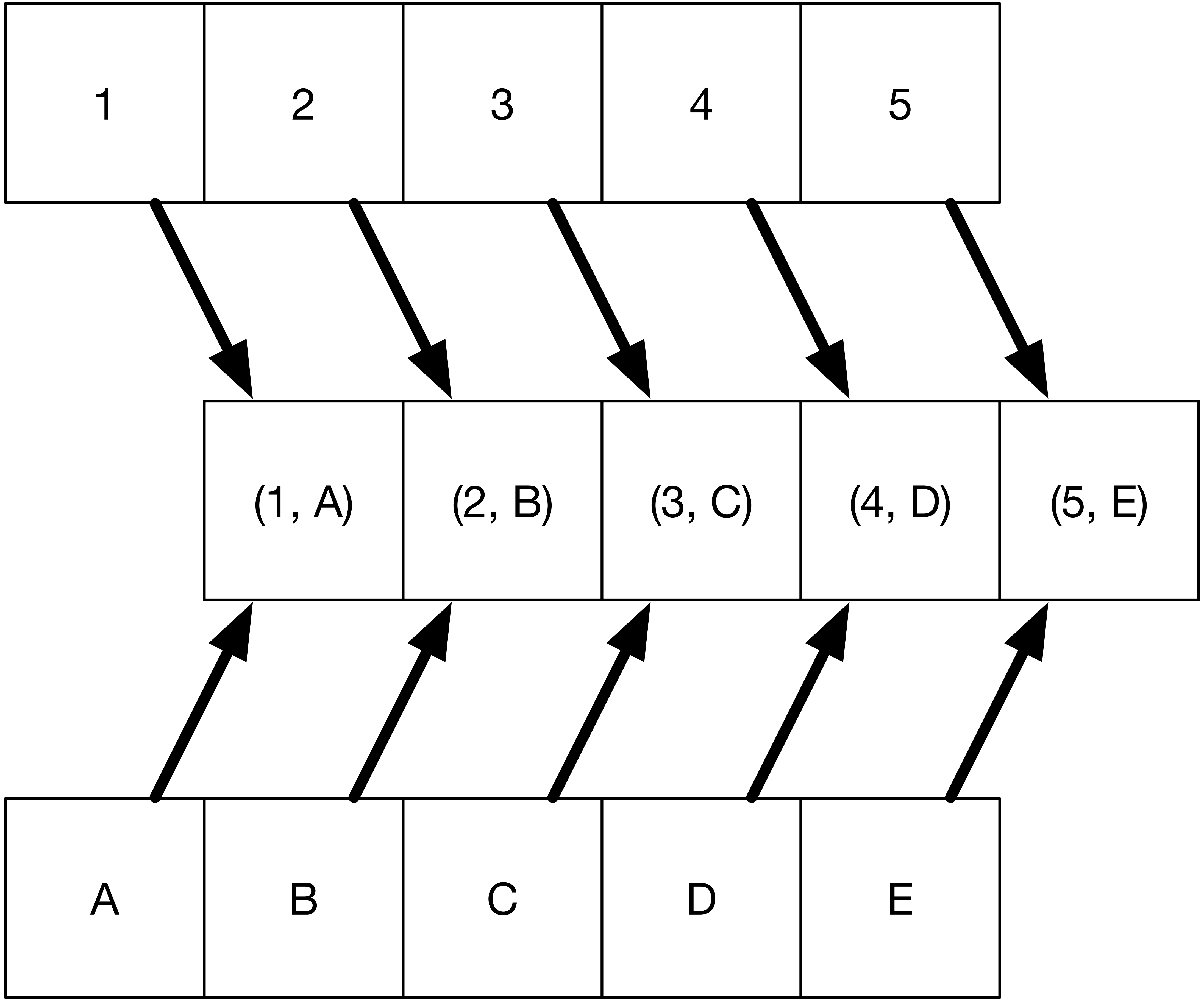
```swift
let ranges = [
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
    Range(start: 1, end: 2),
]
```

Closed Ranges?

```
func ...<Pos : ForwardIndexType>(minimum: Pos, maximum: Pos) -> Range<Pos> {
    return Range(start: minimum, end: maximum.successor())
}

1...2 // ==> 1..<3
```

Zippers


1
2
3
4
5
(1, A)
(2, B)
(3, C)
(4, D)
(5, E)
A
B
C
D
E

```
let xs = [1, 2, 3, 4, 5]
let ys = ["A", "B", "C", "D", "E"]

for pair in zip(xs, ys) {
    println(pair)
}
==>
(1, A)
(2, B)
(3, C)
(4, D)
(5, E)
```

```
func myEnumerate<Seq : SequenceType>(base: Seq)
    -> SequenceOf<(Int, Seq.Generator.Element)> {

        var n = 0
        let nats = GeneratorOf { n++ }
        return SequenceOf(zip(nats, base))
}
```

```
func myEnumerate<Seq : SequenceType>(base: Seq)
    -> SequenceOf<(Int, Seq.Generator.Element)>
```

vs.

```
func myEnumerate<Seq : SequenceType>(base: Seq)
    -> Zip2<GeneratorOf<Int>, Seq>
```

Generator -> Sequence -> Collection -> Sliceable

Array To Zipper

robnapier.net/cocoaconf

Just One More Thing…

Laziness

Mapping every element

```
let xs = [3, 1, 4, 1, 5, 9, 2, 6, 5]
let doublexs = xs.map { $0 * 2 }
let doubleSecond = doublexs[1]
```

Mapping too soon

```
for x in xs.map(f) {
    println(x)
}
```

Mapping too soon (fix)

```
for x in xs {
    println(f(x))
}
```

Map/filter chains

```
let smalldoublexs = xs
    .map { $0 * 2 }
    .filter { $0 < 10 }
```

Map/filter chains (fix)

```
let smalldoublexs = lazy(xs)
    .map { $0 * 2 }
    .filter { $0 < 10 }.array
```

No unasked multiplies

```
let xs = [3, 1, 4, 1, 5, 9, 2, 6, 5]
let doublexs = lazy(xs)
    .map { $0 * 2 }
let doubleSecond = doublexs[1]
```

The many faces of `lazy`

```
/// Augment `s` with lazy methods such as `map`, `filter`, etc.
func lazy<S : CollectionType where S.Index : ForwardIndexType>(s: S)
    -> LazyForwardCollection<S>

/// Augment `s` with lazy methods such as `map`, `filter`, etc.
func lazy<S : SequenceType>(s: S)
    -> LazySequence<S>

/// Augment `s` with lazy methods such as `map`, `filter`, etc.
func lazy<S : CollectionType where S.Index : RandomAccessIndexType>(s: S)
    -> LazyRandomAccessCollection<S>

/// Augment `s` with lazy methods such as `map`, `filter`, etc.
func lazy<S : CollectionType where S.Index : BidirectionalIndexType>(s: S)
    -> LazyBidirectionalCollection<S>
```

```
extension LazySequence {
    func filter(includeElement: (S.Generator.Element) -> Bool)
        -> LazySequence<FilterSequenceView<S>>
    func map<U>(transform: (S.Generator.Element) -> U)
        -> LazySequence<MapSequenceView<S, U>>
}
```

```
extension LazyRandomAccessCollection {
    func filter(includeElement: (S.Generator.Element) -> Bool)
        -> LazySequence<FilterSequenceView<S>>
    func map<U>(transform: (S.Generator.Element) -> U)
        -> LazyRandomAccessCollection<MapCollectionView<S, U>>
}
```

```
let everyother = enumerate(xs)
    .filter { (i, v) in i % 2 == 0 }
    .map    { (_, v) in v } // ERROR
```

```
let everyother = lazy(enumerate(xs))
    .filter { (i, v) in i % 2 == 0 }
    .map    { (_, v) in v }.array
```

- Laziness is opt-in
- Good for map/filter chains
- Nice for getting methods onto generic sequences
- Allows map/filter on infinite sequences