orphan: One of the issues that came up in our design discussions around `Result` was that enum cases don't really follow the conventions of anything else in our system. Our current convention for enum cases is to use `CapitalizedCamelCase`. This convention arose from the Cocoa `NSEnumNameCaseName` convention for constants, but the convention feels foreign even in the context of Objective-C. Non-enum type constants in Cocoa are often namespaced into classes, using class methods such as `[UIColor redColor]` (and would likely have been class properties if those were supported by ObjC). It's also worth noting that our "builtin" enum-like keywords such as `true`, `false`, and `nil` are lowercased, more like properties.

Swift also has enum cases with associated values, which don't have an immediate analog in Cocoa to draw inspiration from, but if anything feel "initializer-like". Aside from naming style, working with enum values also requires a different set of tools from other types, pattern matching with `switch` or `if case` instead of working with more readily-composable expressions. The compound effect of these style mismatches is that enums in the wild tend to grow a bunch of boilerplate helper members in order to make them fit better with other types. For example, `Optional`, aside from the massive amounts of language sugar it's given, vends initializers corresponding to `Some` and `None` cases:

```
extension Optional {
  init(_ value: Wrapped) {
    self = .Some(value)
  }

  init() {
    self = .None
  }
}
```

`Result` was proposed to have not only initializers corresponding to its `Success` and `Error` cases, but accessor properties as well:

```
extension Result {
  init(success: Wrapped) {
    self = .Success(success)
  }
  init(error: Error) {
    self = .Error(error)
  }

  var success: Wrapped? {
    switch self {
    case .Success(let success): return success
    case .Error: return nil
    }
  }
  var error: Error? {
    switch self {
    case .Success: return nil
    case .Error(let error): return error
    }
  }
}
```

This pattern of boilerplate also occurs in third-party frameworks that make heavy use of enums. Some examples from Github:

- https://github.com/antitypical/Manifold/blob/ae94eb96085c2c8195d457e06df485b1cca455cb/Manifold/Name.swift
- https://github.com/antitypical/TesseractCore/blob/73099ae5fa772b90cefa49395f237290d8363f76/TesseractCore/Symbol.swift
- https://github.com/antitypical/TesseractCore/blob/73099ae5fa772b90cefa49395f237290d8363f76/TesseractCore/Value.swift

That people inside and outside of our team consider this boilerplate necessary for enums is a strong sign we should improve our core language design. I'd like to start discussion by proposing the following:

- Because cases with associated values are initializer-like, declaring and using them ought to feel like using initializers on other types. A `case` declaration should be able to declare an initializer, which follows the same keyword naming rules as other initializers, for example:

  ```
  enum Result<Wrapped> {
    case init(success: Wrapped)
    case init(error: Error)
  }
  ```

  Constructing a value of the case can then be done with the usual initializer syntax:

  ```
  let success = Result(success: 1)
  let error = Result(error: SillyError.JazzHands)
  ```

  And case initializers can be pattern-matched using initializer-like matching syntax:

  ```
  switch result {
  case Result(success: let success):
    ...
  case Result(error: let error):
    ...
  ```

```
}
```

- Enums with associated values implicitly receive `internal` properties corresponding to the argument labels of those associated values. The properties are optional-typed unless a value with the same name and type appears in every `case`. For example, this enum:

  ```
  public enum Example {
    case init(foo: Int, alwaysPresent: String)
    case init(bar: Int, alwaysPresent: String)
  }
  ```

  receives the following implicit members:

  ```
  /*implicit*/
  internal extension Example {
    var foo: Int? { get }
    var bar: Int? { get }
    var alwaysPresent: String { get } // Not optional
  }
  ```

- Because cases without associated values are property-like, they ought to follow the `lowercaseCamelCase` naming convention of other properties. For example:

  ```
  enum ComparisonResult {
    case descending, same, ascending
  }

  enum Bool {
    case true, false
  }

  enum Optional<Wrapped> {
    case nil
    case init(_ some: Wrapped)
  }
  ```

Since this proposal affects how we name things, it has ABI stability implications (albeit ones we could hack our way around with enough symbol aliasing), so I think we should consider this now. It also meshes with other naming convention discussions that have been happening.

I'll discuss the points above in more detail:

Case Initializers

Our standard recommended style for cases with associated values should be to declare them as initializers with keyword arguments, much as we do other kinds of initializer:

```
enum Result<Wrapped> {
  case init(success: Wrapped)
  case init(error: Error)
}

enum List<Element> {
  case empty
  indirect case init(element: Element, rest: List<Element>)
}
```

It should be possible to declare unlabeled case initializers too, for types like Optional with a natural "primary" case:

```
enum Optional<Wrapped> {
  case nil
  case init(_ some: Wrapped)
}
```

Patterns should also be able to match against case initializers:

```
switch result {
case Result(success: let s):
  ...
case Result(error: let e):
  ...
}
```

Overloading

I think it would also be reasonable to allow overloading of case initializers, as long as the associated value types cannot overlap. (If the keyword labels are overloaded and the associated value types overlap, there would be no way to distinguish the cases.) Overloading is not essential, though, and it would be simpler to disallow it.

Named cases with associated values

One question would be, if we allow `case init` declarations, whether we should also remove the existing ability to declare named cases with associated values:

```
enum Foo {
  // OK
  case init(foo: Int)
  // Should this become an error?
  case foo(Int)
}
```

Doing so would help unambiguously push the new style, but would drive a syntactic wedge between associated-value and no-associated-value cases. If we keep named cases with associated values, I think we should consider altering the declaration syntax to require keyword labels (or explicit _ to suppress labels), for better consistency with other function-like decls:

```
enum Foo {
  // Should be a syntax error, 'label:' expected
  case foo(Int)

  // OK
  case foo(_: Int)

  // OK
  case foo(label: Int)
}
```

Shorthand for init-style cases

Unlike enum cases and static methods, initializers currently don't have any contextual shorthand when the type of an initialization can be inferred from context. This could be seen as an expressivity regression in some cases. With named cases, one can write:

```
foo(.Left(x))
```

but with case initializers, they have to write:

```
foo(Either(left: x))
```

Some would argue this is clearer. It's a bit more painful in `switch` patterns, though, where the type would need to be repeated redundantly:

```
switch x {
case Either(left: let left):
  ...
case Either(right: let right):
  ...
}
```

One possibility would be to allow `.init`, like we do other static methods:

```
switch x {
case .init(left: let left):
  ...
case .init(right: let right):
  ...
}
```

Or maybe allow labeled tuple patterns to match, leaving the name off altogether:

```
switch x {
case (left: let left):
  ...
case (right: let right):
  ...
}
```

Implicit Case Properties

The only native operation enums currently support is `switch`-ing. This is nice and type-safe, but `switch` is heavyweight and not very expressive. We now have a large set of language features and library operators for working with `Optional`, so it is expressive and convenient in many cases to be able to project associated values from enums as `Optional` values. As noted above, third-party developers using enums often write out the boilerplate to do this. We should automate it. For every `case init` with labeled associated values, we can generate an `internal` property to access that associated value. The value will be `Optional`, unless every `case` has the same associated value, in which case it can be nonoptional. To repeat the above example, this enum:

```
public enum Example {
  case init(foo: Int, alwaysPresent: String)
  case init(bar: Int, alwaysPresent: String)
```

```
}
```

receives the following implicit members:

```
/*implicit*/
internal extension Example {
  var foo: Int? { get }
  var bar: Int? { get }
  var alwaysPresent: String { get } // Not optional
}
```

Similar to the elementwise initializer for `struct` types, these property accessors should be `internal`, since they rely on potentially fragile layout characteristics of the enum. (Like the struct elementwise initializer, we ought to have a way to easily export these properties as `public` when desired too, but that can be designed separately.)

These implicit properties should be read-only, until we design a model for enum mutation-by-part.

An associated value property should be suppressed if:

- there's an explicit declaration in the type with the same name:

```
enum Foo {
  case init(foo: Int)

  var foo: String { return "foo" } // suppresses implicit "foo" property
}
```

- there are associated values with the same label but conflicting types:

```
enum Foo {
  case init(foo: Int, bar: Int)
  case init(foo: String, bas: Int)

  // No 'foo' property, because of conflicting associated values
}
```

- if the associated value has no label:

```
enum Foo {
  case init(_: Int)

  // No property for the associated value
}
```

  An associated value could be unlabeled but still provide an internal argument name to name its property:

```
enum Foo {
  case init(_ x: Int)
  case init(_ y: String)

  // var x: Int?
  // var y: String?
}
```

Naming Conventions for Enum Cases

To normalize enums and bring them into the "grand unified theory" of type interfaces shared by other Swift types, I think we should encourage the following conventions:

- Cases with associated values should be declared as `case init` initializers with labeled associated values.
- Simple cases without associated values should be named like properties, using `lowercaseCamelCase`. We should also import Cocoa `NS_ENUM` and `NS_OPTIONS` constants using `lowercaseCamelCase`.

This is a big change from the status quo, including the Cocoa tradition for C enum constants, but I think it's the right thing to do. Cocoa uses the `NSEnumNameCaseName` convention largely because enum constants are not namespaced in Objective-C. When Cocoa associates constants with class types, it uses its normal method naming conventions, as in `UIColor.redColor`. In Swift's standard library, type constants for structs follow the same convention, for example `Int.max` and `Int.min`. The literal keywords `true`, `false`, and `nil` are arguably enum-case-like and also lowercased. Simple enum cases are essentially static constant properties of their type, so they should follow the same conventions.