Swiftの関数と代数学（指数）

part2 (#9 Algebraic Data Types: Exponents)

今回のテーマについて

- 前回の続きです
  - #4 Algebraic Data Types（前回：struct・enum と代数学）
  - **#9 Algebraic Data Types: Exponents（指数）**
  - #19 Algebraic Data Types: Generics and Recursion
- Swift の struct -> 直積型、enum -> 直和型（前回）
- Swift の func -> 指数（今回）

part1 の復習 - struct（直積型）

Swift の struct は直積型

```
struct Pair<A, B> {
  let first: A
  let second: B
}

Pair<Bool, Bool>(first: true, second: true)
Pair<Bool, Bool>(first: true, second: false)
Pair<Bool, Bool>(first: false, second: true)
Pair<Bool, Bool>(first: false, second: false)

// Bool は true or false の 2 通りの値を持つ
// 2 * 2 で全 4 パターンが想定される
```

part1 の復習 - enum（直和型）

Swift の associated value を持つ enum は直和型

```
enum Either<A, B> {
  case left(A)
  case right(B)
}

Either<Bool, Bool>.left(true)
Either<Bool, Bool>.left(false)
Either<Bool, Bool>.right(true)
Either<Bool, Bool>.right(false)

// 2 + 2 で全 4 パターンが想定される
```

part1 の復習 - Void は特殊

- Void は値を一つだけ持つ（空のタプル）
  - 代数的には **1** として扱う

```
// 2(Bool) * 1(Void) = 2 パターン
Pair<Bool, Void>(first: true, second: ())
Pair<Bool, Void>(first: false, second: ())

// 2(Bool) + 1(Void) = 3 パターン
Either<Bool, Void>.left(true)
Either<Bool, Void>.left(false)
Either<Bool, Void>.right(())
```

part1 の復習 - Never も特殊

- Never は case を持たない enum（値を持たない型）
  - 代数的には **0** として扱う

```
// 2(Bool) * 0(Never) = 0 パターン
Pair<Bool, Never>(first: true, second: ???) // コンパイルできない

// 2(Bool) + 0(Never) = 1 パターン
Either<Bool, Never>.first(true)
Either<Bool, Never>.first(false)
Either<Bool, Never>.second(???) // コンパイルできない
```

part1 の復習 - Optional について

Optional は assocaited value を持つ enum の片方の case が Void であることとほぼ同じ意味

```
Either<Void, A> {
  case left(())
  case right(A)
}

enum Optional<A> {
  case none
  case some(A)
}

// Bool? -> 1(Void or none) + 2(Bool) = 3
// Data? -> 1(Void or none) + 1(Data) = 2
```

part1 の復習 - それで何が嬉しい？

- 代数学的な直感を使えば、無駄な型を簡潔にすることが
  できたりする（↓の例は正確ではない。議論は Point-Free 参照）

```
URLSession.shared
.dataTask(with: url,
          completionHandler: (data: Data?,
                              response: URLResponse?,
                              error: Error?) -> Void)

// Swift のタプルは単なる積なので、パターンとして、
// 2(Data?) * 2(URLResponse?) * 2(Error?) = 8 が考えられるが無駄が多い
// 本当に必要なものは、Data と URLResponse か Error のみが返ってくるという情報だけ
// 代数的に表すと Data * URLResponse + Error
// 型にすると Either<Pair<Data, URLResponse>, Error>
// Swift で言う Reuslt が近い （≒ Result<(Data, Response), Error>）
```

代数学における指数（Exponents）

指数計算はどのように捉えることができるのか？

```
enum Three {
  case one, two, three
}

// Bool^Three
// = Bool^(1 + 1 + 1) (enum は単なる和と見なせるため)
// = Bool^1 * Bool^1 * Bool^1
```

それぞれの項は Three の値によってタグ付けされていると考えられる
つまり、Three の各値に Bool を割り当てることと等しいと考えられる

Three の各値に Bool を割り当てるとは？

`(Three) -> Bool`、つまり関数と捉えることができる

```
func f1(_ x: Three) -> Bool {
  switch x {
    case .one: return true
    case .two: return true
    case .three return true
  }
}

// ...（以下 2(Bool)^3(Three) = 8 パターン分続く
```

代数学における指数と関数を結びつけることができた

一旦指数と関数の関係についてまとめる

```
// Bool^Three
// = Bool^(1 + 1 + 1)
// = Bool^1 * Bool^1 * Bool^1 ≒ (Three) -> Bool

// 一般化すると
// A^B ≒ (B) -> A
```

`A^B ≒ (B) -> A` という関係が得られる

副作用を含むものについては考えない

無限に挙げられるため、以下のような副作用を持つものについては
考えないものとする

```
func foo1(_ x: Three) -> Bool {
  print("hello")
  return true
}
func foo2(_ x: Three) -> Bool {
  print("world")
  return false
}
func foo3(_ x: Three) -> Bool {
  URLSession.shared.dataTask(with: URL(string: "https://www.pointfree.co")!).resume()
  return true
}
```

指数と関数の関連性から考えられるパターン

- パターン1: 複数の指数計算
- パターン2: 1乗の指数計算
- パターン3: 0乗の指数計算

パターン1: 複数の指数計算

```
// (a^b)^c = a^(b * c)

// ↓ まず ^ を <- に置き換える

// (a <- b) <- c = a <- (b * c)

// ↓ 反転させる

// c -> (b -> a) = (b * c) -> a

// ↓ Swift の型システムに置き換える

// (C) -> (B) -> A = (B, C) -> A
```

(C) -> (B) -> A = (B, C) -> A

- ある関数を返す関数があれば、引数を二つとる関数と等価だということを表している
- Point-Free はこの左辺と右辺を行き来する関数として、`curry`, `uncurry` というものを紹介している
- `curry`: カリー化
  - カリー化を普及させたハスケル・カリーにちなんで名付けられている
  - 発見したのはモーゼス・シェーンフィンケル

curry, uncurry

```
func curry<A, B, C>(_ f: @escaping (A, B) -> C) -> (A) -> (B) -> (C) {
    return { a in { b in f(a, b) } }
}

func uncurry<A, B, C>(_ f: @escaping (A) -> (B) -> C) -> (A, B) -> C {
    return { a, b in f(a)(b) }
}

String.init(data:encoding:) // (Data, String.Encoding) -> String?
curry(String.init(data:encoding:)) // (Data) -> (String.Encoding) -> String?
uncurry(curry(String.init(data:encoding:))) // (Data, String.Encoding) -> String?
```

パターン2: 1乗の指数計算

```
// a^1 = a

// ↓ まず ^ を <- に置き換える

// a <- = a

// ↓ 反転させる

// 1 -> a = a

// ↓ Swift の型システムに置き換える

// (Void) -> A = A
// これに対して Point-Free は zurry/unzurry というものを紹介している
```

(Void) -> A = A

```swift
func to<A>(_ f: (Void) -> A) -> A {
    return f(())
}

func from<A>(_ a: A) -> (Void) -> A {
    return { _ in a }
}

func zurry<A>(_ f: () -> A) -> A {
    return f()
}

func unzurry<A>(_ a: A) -> () -> A {
    return { a }
}
```

⚠ When calling this

⚠ When calling this

パターン3: 0乗の指数計算

```
// a^0 = 1

// ↓ まず ^ を <- に置き換える

// a <- 0 = 1

// ↓ 反転させる

// 0 -> a = 1

// ↓ Swift の型システムに置き換える

// (Never) -> A = Void
```

(Never) -> A = Void

- この形は非常に奇妙
- Never には何もないはず
- しかし Never から 他の型への関数には Void、つまり一つの値があるということを示している
- これについて考えるために、それぞれを行ったり来たりする関数を今までと同じように考えてみる

(Never) -> A を Void にする関数

この場合は非常に簡単である

```
func to<A>(_ f: (Never) -> A) -> Void {
  return ()
}
```

Void を (Never) -> A にする関数

しかし、こちらは難しい

```
func from<A>(_ x: Void) -> (Never) -> A {
  // ?????
}
```

少しずつこの関数を完成させてみる

Void を (Never) -> A にする関数

まず、`(Never) -> A` を返すことがわかっているので、スタブで考えてみる

```
func from<A>(_ x: Void) -> (Never) -> A {
  return { never in
    // ?????
  }
}
```

- never には値がない
- そんな never を使って何かできるか？

Void を (Never) -> A にする関数

- never の値が存在しないとしても、never の値で動作する関数を定義できないということではない
- Never は enum であるため、↓のように定義することができる！

```
func from<A>(_ x: Void) -> (Never) -> A {
  return { never in
    switch never { // Will never be executed の警告は発生する
      // 何もないがコンパイルできる！！
    }
  }
}
```

なぜ定義できるか？

以下のような一般的な enum の処理を考えてみる
（Either は left と right の case を持つ enum）

```
func isInt(_ x: Either<Int ,String>) -> Bool {
  switch x {
  case .left: return true
  case .right: return false
  }
}
// ここから先では何かを return する必要はない

// なぜなら Swift は enum の全ての case が処理されたことを知っているから
```

何もしない関数 absurd

いくつかの言語ではこのような何もしない関数に absurd という
名前が付けられている

```
func absurd<A>(_ never: Never) -> A {
  switch never {}
}
```

- absurd の意味は「ばかげている、常識に反した、不条理など」
  他の言語においても使うことはほぼないらしい
- absurd を使うことができる例として `Result.fold` を見ていく

Result.fold

```
extension Result {
  func fold<A>(ifSuccess: (Value) -> A, ifFailure: (Error) -> A) -> A {
    switch self {
    case let .success(value):
      return ifSuccess(value)
    case let .failure(error):
      return ifFailure(error)
    }
  }
}
```

Value から A, Error から A にという形で、二つの型を一つの Aという型に「折りたたむ（Fold）」ことができる関数

fold の使い方

例えば Result に整数値が含まれていたり、エラーメッセージとして
文字列が含まれているとする

```
let result: Result<Int, String> = .success(2)
```

例えば、その Result を文字列に折りたたんで表示したい場合
↓のように使うことができる

```
result.fold(
  ifSuccess: { _ in "Ok" },
  ifFailure: {_ in "Something went wrong" }
)
// "Ok"
```

Result の Error に Never を入れる

Combine でもその考えは利用されているが、Result の Error に
Never を入れると失敗することがない Result を得ることができる

```
let infallibleResult: Result<Int, Never> = .success(2)

// オートコンプリートで ↓ が出てくる

infallibleResult.fold(
  ifSuccess: <# (Int) -> A #>,
  ifFailure: <# (Never) -> A #>
)
```

`(Never) -> A` ❓🤔🤔🤔

(Never) -> A = absurd

```
infallibleResult.fold(
  ifSuccess: <# (Int) -> A #>,
  ifFailure: <# (Never) -> A #>
)

          ↓↓↓↓↓↓↓

infallibleResult.fold(
  ifSuccess: { _ in "Ok" },
  ifFailure: absurd
)
```

何の意味もないと思っていた absurd が使用できることを発見🙌

指数と関数の関連性がわかって何が嬉しい？

- 以前は struct や enum を代数的に捉えることによって、
  型をどのように構築すれば良いかの役に立つことがわかった
- 今回は、指数・関数の関連性についても理解することによって
  関数の場合でも同じようなことができるようになった
- それを実感するために、もう少しだけ例を見ていきます

一つ目の例

```
// a^(b + c) == a^b * a^c
// a <- (b + c) == (a <- b) * (a <- c)
// (b + c) -> a == (b -> a) * (c -> a)

// (Either<B, C>) -> A == ((B) -> A, (C) -> A)

// Either は ↓ （ただの enum）
Either<A, B> {
  case left(A)
  case right(B)
}
```

- こちらも先ほどまでと同じく左辺、右辺を行ったり来たりする関数が定義できる

(Either<B, C>) -> A == ((B) -> A, (C) -> A)

```swift
// 閲覧者用の練習問題として用意されていたため、正しいかはわからないです 🤷
func to<A, B, C>(_ f: @escaping (Either<B, C>) -> A) -> ((B) -> A, (C) -> A) {
    return (
        { b in f(.left(b)) },
        { c in f(.right(c)) }
    )
}

func from<A, B, C>(_ f: ((B) -> A, (C) -> A)) -> (Either<B, C>) -> A {
    return { either in
        switch either {
        case .left(let b): return f.0(b)
        case .right(let c): return f.1(c)
        }
    }
}
```

この例からわかること

```
(Either<B, C>) -> A == ((B) -> A, (C) -> A)
```

- Either を取る関数は、実際には左の値を操作する関数と
  右の値を操作する関数の二つの関数であることを示している
- 関数からわかることは**関数の入力に enum の case を追加しても、小さな単位に分解できるため、複雑さはそこまで増加しない**こと
- 仮に入力を増やした場合でも **入力である Either は enum であるためコンパイラに修正することを強制される**（自然な形で修正可能）

二つ目の例

```
// (a * b)^c = a^c * b^c
// (a * b) <- c = (a <- c) * (b <- c)
// c -> (a * b) = (c -> a) * (c -> b)

// (C) -> (A, B) = ((C) -> A, (C) -> B)
```

(C) -> (A, B) = ((C) -> A, (C) -> B)

```swift
// こちらも同じく合っているかはわからないです 🤷
func to<A, B, C>(_ f: @escaping (C) -> (A, B)) -> ((C) -> A, (C) -> B) {
    return (
        { c in return f(c).0 },
        { c in return f(c).1 }
    )
}

func from<A, B, C>(_ f: ((C) -> A, (C) -> B)) -> (C) -> (A, B) {
    return { c in
        let a = f.0(c)
        let b = f.1(c)
        return (a, b)
    }
}
```

この例からわかること

```
(C) -> (A, B) = ((C) -> A, (C) -> B)
```

- タプルへの関数はタプルの各辺にマッピングされた関数のタプルと同じ
- **フィールドを追加して関数の戻り値の型を拡張しても、関数の複雑さがそれほど増すわけではない**
- また、そうなるように**コンパイラが強制している**（自然な形で修正可能）

以上の指数と関数の関連性からわかること

- 指数的に左辺 = 右辺が成り立っていれば、複雑な関数を分解することができたりする
- 関数の入力や出力を増やしたとしても、コンパイラが自然な形での修正に導いてくれる

仮に指数計算を誤っていたとしたらどうなるか？（これについては、自分の中であまり納得感を持つことができていません🤦）

一つ目の誤った指数計算

```
// a^(b * c) != a^b * a^c
// a <- (b * c) != (a <- b) * (a <- c)
// (b * c) -> a != (b -> a) * (c -> a)

// (B, C) -> A != ((B) -> A, (C) -> A)
```

(B, C) -> A != ((B) -> A, (C) -> A)

- タプルからの関数は**これ以上単純化されることはないということを示している**
  - つまり、フィールドを追加して関数の入力を拡張すると、より複雑な関数になるということ

二つ目の誤った例

```
// (a + b)^c != a^c + b^c
// (a + b) <- c != (a <- c) + (b <- c)
// c -> (a + b) != (c -> a) + (c -> b)

// (C) -> Either<A, B> != Either<(C) -> A, (C) -> B>
```

(C)->Either<A, B> != Either<(C) -> A, (C) -> B>

- Either にしても、それ以上単純化されることはない
- case を追加して関数の出力を増やすことは、考慮すべき case が増えてしまうことを示している
- それにも関わらず、それらのケースを考慮することを強制するものは何もない（正しい場合はコンパイラが強制してくれた）
- これは Result 型を返す関数や throw を返す関数が他の関数よりも当然複雑になる理由も示している
- 特に `(A) -> Result<B, E>` という形になっているため、代数的に捉えると $(B + E)^A$ となり、複雑であることがわかる

まとめ

- Swift の関数と代数学における指数には関連がある
- 指数を利用すれば、複雑な関数をコンパイラの手を借りながら分解することができる
- 指数計算として成立していないものは、それ以上関数を単純化できないということを示している（自分は納得感を持てていない）
- （余談）: TCA の combine, pullback をしっかりと理解したくて、Point-Free の Reducers and Stores を見始めています
  - combine, pullback がなぜ必要なのか、どのように実装されているかなどを知ることができてもっと早く読んでおけば良かったとなっています...😢