Tech Talk

関数プログラミング入門

masatora 2021/05/27

このテーマを扱う動機

1. Scalaを使っている会社でアルバイトを始めたので勉強したい
2. 面白かったので共有したい

ターゲット

命令型言語でのプログラミング経験があるが、関数プログラミングの経験はない人

ゴール

関数プログラミングがどんな感じかイメージできるようになってもらう

参考文献

- 関数プログラミング実践入門
- 関数プログラミングはなぜ重要か
- Scala研修テキスト by ドワンゴ
- Scala関数型デザイン&プログラミング
- Scala by Example
- カリー化と部分適用（JavaScriptとHaskell）

目次

- **関数プログラミングのデモ** - 関数プログラミングではどんな感じで問題を解くか簡単にデモします。
- **関数プログラミングとはなにか** - 関数プログラミングとはなにか説明します。
- **関数型言語とはなにか** - 関数型言語とはなにか説明します。
- **関数プログラミングの何が嬉しいか** - 関数プログラミングのメリットを説明します。
- **いろんな言語に見られる関数プログラミングの影響** - 最近のプログラミング言語は関数プログラミングの要素を取り入れている事が多いので紹介します。
- **関数プログラミングをやってみる** - 実際に手を動かしてもらえるような問題を用意しました。

関数プログラミングのデモ

命令型言語で書くクイックソート ~Cの場合~

```c
#include<stdio.h>

void swap (int *x, int *y) {
  int temp;    // 値を一時保存する変数
  temp = *x;
  *x = *y;
  *y = temp;
}

int partition (int array[], int left, int right) {
  int i, j, pivot;
  i = left;
  j = right + 1;
  pivot = left;   // 先頭要素をpivotとする

  do {
    do { i++; } while (array[i] < array[pivot]);
    do { j--; } while (array[pivot] < array[j]);
    // pivotより小さいものを左へ、大きいものを右へ
    if (i < j) { swap(&array[i], &array[j]); }
  } while (i < j);
  swap(&array[pivot], &array[j]);   //pivotを更新
  return j;
}
```

```c
void quick_sort (int array[], int left, int right) {
  int pivot;
  if (left < right) {
    pivot = partition(array, left, right);
    quick_sort(array, left, pivot-1);   // pivotを境に再帰的
    quick_sort(array, pivot+1, right);
  }
}
```

参考: https://qiita.com/omu58n/items/7794f097b2c569a469bc

関数型言語で書くクイックソート

Haskell

```haskell
sort [] = []
sort (x:xs) = sort [ a | a ← xs, a < x ] ++ x : sort [ a |
```

Scala

```scala
def quickSort(xs: Array[Int]): Array[Int] = {
    if (xs.length ⩽ 1) xs
    else {
        val pivot = xs(xs.length / 2)
        Array.concat(
            quickSort(xs filter (pivot >)),
            xs filter (pivot ==),
            quickSort(xs filter (pivot <)))
    }
}
```

関数プログラミングとはなにか

関数プログラミングとは

関数プログラミングとはプログラミングパラダイムの1つ

プログラミングのパラダイム

有名なプログラミングパラダイムとして、以下の３つがある。

1. 命令型プログラミングのパラダイム
2. オブジェクト指向プログラミングのパラダイム
3. 関数プログラミングのパラダイム

1. 命令型プログラミングのパラダイム

プログラムとは、計算機が行うべき命令の列であるとするパラダイム。

2. オブジェクト指向プログラミングのパラダイム

プログラムとは、オブジェクトとそのメッセージング
であるとするパラダイム。

3. 関数プログラミングのパラダイム

プログラムとは、関数であるとするパラダイム。値に関数を適用していくことで計算をすすめる。

関数とはなにか

関数プログラミングにおける関数とは、数学的な意味での関数。

すなわち、与えられた入力の値のみから出力となる値をただ１つ決める規則のこと。

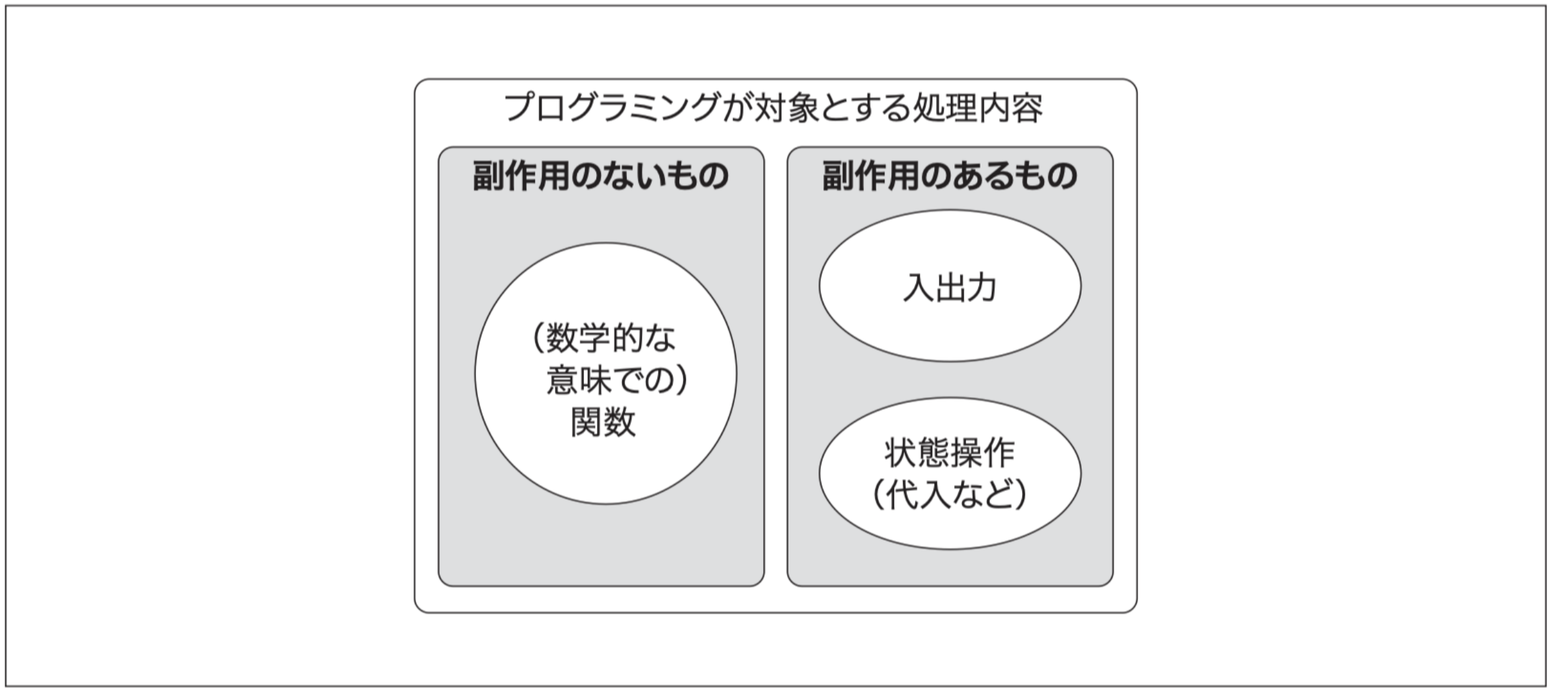
言い換えると、プログラミングが対象とする処理内容のうち、副作用のないもの。

$$f(x)$$

副作用とは

副作用とは、状態を参照し、あるいは状態に変化を与えることで、次回以降の結果にまで影響を与える効果のこと。

図0.1 プログラミングが対象とする処理



関数型言語とはなにか

関数型言語とはなにか

関数型言語とは、関数が第一級の対象である言語のこと。

第一級の対象であるとは

「第一級の対象である」とは、その言語において単なる値と同じように扱われるということ。言い換えると、以下の特徴を与えられているということ。

1. リテラルがある
2. 実行時に生成できる
3. 変数に入れて扱える
4. 手続きや関数に引数として与えることができる
5. 手続きや関数の結果として返すことができる

1. リテラルがある

関数型言語では、文字列や数値と同様に、関数にもリテラルが与えられている。たとえば、引数に1を足す関数は

```
(λ x . x + 1)
```

という風に表せる。この関数に2を適用すると

```
(λ x . x + 1) (2) = 2 + 1 = 3
```

2. 実行時に生成できる

関数型言語では、(関数合成、部分適用や高階関数等によって)関数を実行時に生成できる。

2. 実行時に生成できる ~部分適用とは~

関数の一部にだけ引数を与えて関数を作ること。例えば、２つの引数の合計を返す関数`add`を以下のように定義する。

```
add x y = x + y
```

以下のようにaddに部分適用を行なって新しい関数`add1`を作る。`add1`は引数を１つとって、引数に1を足して返す関数になる。

```
-- 部分適用
add1 = add 1

-- 使用例
> add1 2
3
```

2. 実行時に生成できる ~高階関数とは~

- 結果が関数になる関数
- 引数として関数を要求する関数

を高階関数という(詳細は後のページで説明)。

3. 変数に入れて扱える

関数型言語では、関数を変数に入れて扱うことができる。

4. 手続きや関数に引数として与えることができる

定義域が特定の関数の集合となるような関数を作ることができる。

例えば、関数を引数にとって、その関数に2を適用した結果になる関数を以下のように作れる。

```
(λ f . f(2))
```

この関数に先程の関数`(λ x . x + 1)`を適用すると

```
(λ f . f(2)) ((λ x . x + 1)) → 2 + 1 → 3
```

となる。↑は高階関数

5. 手続きや関数の結果として返すことができる

値域が特定の関数の集合となるような関数を作ることができる。

i.e. 関数の結果として関数を返すことができる。

例えば、『値を１つ取って、その値を足す』関数を返す関数を以下のようにして作れる。

```
(λ a . (λ b . b + a))
```

上記の関数に1を適用すると

```
(λ a . (λ b . b + a))(1) → (λ b . b + 1)
```

となり、『引数に1を足して返す』関数ができる。

命令型言語と関数型言語の違い

命令型言語と関数型言語の違いは主に2つ。

1. 手続きを書く vs 結果の性質を宣言する
2. 代入 vs 束縛

1. 手続きを書く vs 結果の性質を宣言する

命令型言語では、ある目的を達成するため、

- 値をどこに格納するのか(代入)
- どこから値を取り出してくるのか(参照)
- 次にどの手続に飛ぶのか(手続きの呼び出し)

といったような挙動を記述する。

関数型言語では、結果の性質を宣言することによって目的を達成する。

2. 代入 vs 束縛

関数型言語では、変数への代入は許されていないか、もしくは非常に限定されている。

代入の代わりに、一度変数の値を決めたら変更不可能になる束縛を用いる。

例えるなら、C言語ですべての変数に`const`をつけるようなもの

関数型言語の例

- Clojure
- Coq
- Erlang
- F#
- Haskell
- Idris
- OCaml
- Scala
- SML

関数型言語の分類

以下の6つの視点で分類する。

1. 動的型付けと静的型付け
2. 型付けの強弱
3. 純粋
4. 型推論
5. 依存型
6. 評価戦略

表0.1 おもな関数型言語と命令型言語の機能

言語	型付け	純粋	型推論	依存型	評価戦略
Agda	強い、静的	○	○	○	?
Clean	強い、静的	○	○	×	遅延(?)
Coq	強い、静的	○	○	○	遅延(?)
F#	強い、静的	×	△	×	積極
Haskell	強い、静的	○	○	△	遅延
Idris	強い、静的	○	○	○	積極
OCaml	強い、静的	×	○	△	積極
Scala	強い、静的	×	△	×	積極
SML	強い、静的	×	○	×	積極
Clojure	強い、動的	×	-	-	積極
Erlang	強い、動的	×	-	-	積極

1. 動的型付けと静的型付け

関数型言語には、静的型付け言語と動的型付け言語とが存在する。

- 静的型付け言語とは、型検査をコンパイル時に行う。
- 動的型付け言語は、実行時に型検査を行う。

2. 型付けの強弱

- 型検査(型に整合性があるかを検査する機能)に成功すれば安全性が保証される型付けを強い型付け
- 型検査に成功しても安全性が保証されない型付けを弱い型付けという。

3. 純粋

純粋とは、同じ式はいつ評価しても同じ結果になる参照透過性という性質を持っていること。

4. 型推論

型推論とは、陽に与えられた型の情報から、陽に与えられていない部分の型も推論してくれる機能。

5. 依存型

依存型とは、他の型に依存した型や、値に依存した型を作れる機能。

6. 評価戦略

評価戦略とは、どのような順番で式を評価するかという規則のこと。評価戦略には大きく2種類ある。

1. 積極評価: 引数を渡される前に評価する
2. 遅延評価: 必要になるまで評価されない

関数型言語の歴史

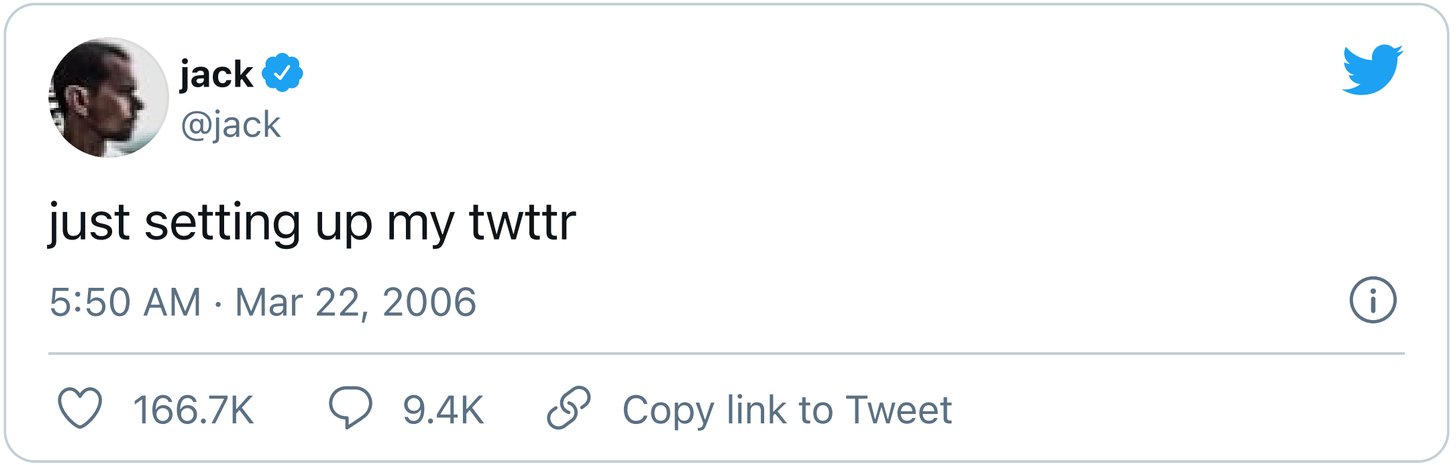
関数型言語のこれから

- これからの関数型言語は、言語の制約の記述力を高め、その制約を言語やライブラリが把握・利用できる方向に進むだろう
- 航空・宇宙・医療・原子力等、システムのバグが人命等に関わる分野で今後採用される可能性がある

関数型言語が採用されている分野/プロダクト

- Twitter(scala)
- LinkedIn(scala)
- Foursquare(scala)
- Tabbles(F#)
- manaba(Haskell)

等



関数プログラミングの何が嬉しいか

関数プログラミングのメリット

関数プログラミングのメリットは大きく以下の5つである。

1. コード量が少なくなる
2. 最適化がしやすい
3. 並行/並列化がしやすい
4. バグり/バグらせにくい
5. ドキュメントが少なくなる

1. コード量が少なくなる

抽象的な表現力が高く、かつ宣言的に記述できるため、記述が簡潔になる。

2. 最適化がしやすい

通常よく行われる最適化手法(ループ、データフロー、SSAなどの最適化・アセンブラレベルのコード生成最適化)に加えて、数学的な性質を利用した最適化を行いやすい。

3. 並行/並列化がしやすい

- いくつかの関数型言語ではSTMというDBのトランザクションおよびリトライ機構に似た機構を備えており、排他制御を簡単に扱える。
- (純粋な言語は)参照透過性を満たしていることが自明であるため、簡単に並列化できる。

4. バグり/バグらせにくい

高度な制約条件を型として表現でき、かつその制約条件が守られていることをコンパイル時にチェックできるため、『正しくない』使い方がされづらい。

5. ドキュメントが少なくなる

高度な制約条件が守られているかどうか言語がチェックしてくれるため、その制約条件についてのドキュメンテーションの必要がなくなる。

関数型言語のメリット

関数型言語のメリットは大きく以下の5つである。

1. 宣言的であることのメリット
2. 制約の充足をチェックしてくれるメリット
3. 型と型検査があることのメリット
4. 型推論のメリット
5. 束縛によるメリット

1. 宣言的であることのメリット

「結果が満たすべき性質」を書くだけで良いため、記述が簡潔になる。

2. 制約の充足をチェックしてくれるメリット

言語機能として課せられたか、もしくはプログラマが与えた制約が守られていることを言語がチェックしてくれる。それによって、プログラマが注意を払う必要のある事柄が減る。

3. 型と型検査があることのメリット

関数の引数として、その関数の定義域に収まらない値が与えられる(合成できない関数同士を合成してしまう場合等)ことを事前に防ぐことができる。

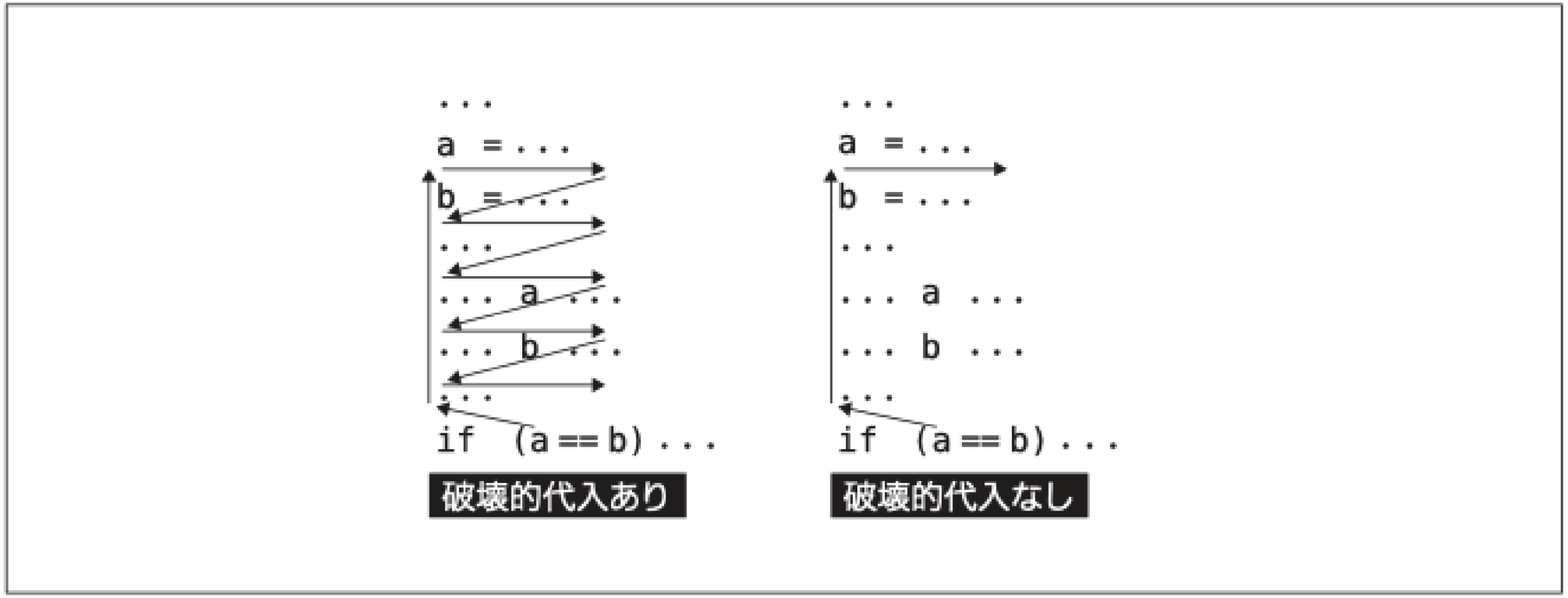
4. 型推論のメリット

逐一型を書かなくて良いため楽。

5. 束縛によるメリット

破壊的代入がないため、コードリーディングが楽。

図0.3 コードリーディング時の視線移動



いろんな言語に見られる関数プログラミングの影響

関数プログラミングの影響 ~ラムダ式~

いろんな言語でラムダ式による無名関数の定義が可能。

関数プログラミングの影響 ~部分適用~

RubyやC++, Python, Go等では部分適用に相当する機能を表現することができる。

Ruby

```ruby
add = ->(a, b) { a + b }.curry # Proc#curry を使う
inc = add.(1) # インクリメント
puts inc.(1) # 2を印字
puts inc.(2) # 3を印字
```

Go

```go
var add = func(x int, y int) int { return x + y }
var inc = func(y int) int { return add(1, y) } // インクリメント
```

関数プログラミングの影響 ~モナド~

Java, Swift等ではモナドが採用されている。

SwiftのOptionalモナド

```
let number: Int? = Optional.some(42)
let noNumber: Int? = Optional.none
print(number) // Optional(42)
print(noNumber) // nil
print(noNumber == nil) // true
```

SwiftのOptional Chaining

```
let imagePaths = ["star": "/glyphs/star.png",
                  "portrait": "/images/content/portrait.jpg",
                  "spacer": "/images/shared/spacer.gif"]
if imagePaths["star"]?.hasSuffix(".png") == true {
    print("The star image is in PNG format")
}

// The star image is in PNG format
```

参考: https://developer.apple.com/documentation/swift/optional

モナドとは

『モナド』とは、文脈を伴う計算同士を組み合わせ可能にする仕組みのこと

モナドの例

- Maybeモナド - 失敗の可能性という文脈を扱う
- Readerモナド - 参照できる環境を共有したい時に使う(configから環境変数を読み込むとか)
- Writerモナド - 主要な計算の横で、別の値も一直線に合成したいときに使う
- Stateモナド - 状態の引き継ぎという文脈を扱う
- IOモナド - 副作用を伴う

等々

ここで発生する(多分)疑問

1. 文脈とはなにか
2. 文脈を持つ計算同士の組み合わせが難しいのはなぜか
3. 文脈を持つ計算をうまく扱いたいのはなぜか

Maybeモナド(1)

計算は失敗することがある。例えば、以下の非負整数を2乗する関数`square`は入力が負数の場合失敗する。

```haskell
square :: Integer → Maybe Integer
square n
| 0 ⩽ n = Just (n * n) -- 入力が非負整数の場合、入力を2乗して返す
| otherwise = Nothing --それ以外の場合、失敗
```

次に、整数の1/2乗を返す関数`squareRoot`を考える。`squareRoot`は入力が負数の場合・結果が整数にならない場合に失敗する。

```haskell
squareRoot :: Integer → Maybe Integer
squareRoot n
| 0 ⩽ n = squareRoot' 1
| otherwise = Nothing where -- 入力が負数の場合失敗
squareRoot' x
  | n > x * x = squareRoot' (x + 1)
  | n < x * x = Nothing -- 結果が整数にならない場合失敗
  | otherwise = Just x
```

上記の関数`square`と`squareRoot`を順番に適用すれば、元の数が得られる。しかし、これらの関数は失敗することがあるので、単純に関数合成を行うことができない。

Maybeモナド(2)

`square`と`squareRoot`のような失敗する可能性のある関数を合成するには、以下のように場合分けする必要がある。

```haskell
squareAndSquareRoot1 :: Integer → Maybe Integer
squareAndSquareRoot1 n = case square n of
                           Nothing → Nothing -- squareが失敗したら→失敗
                           Just nn → squareRoot nn -- squareが成功したら→squareRootを適用
```

上記は引数が１つしかなく、関数も２つだけなのでそこまで問題ない。

しかし、仮に『2つの整数にそれぞれ関数`square`を適用し、結果を掛け算したものを関数`squareRoot`に与える』という処理を書く場合、以下のように冗長な記述となる。

```haskell
squareAndSquareRoot2 :: Integer → Integer → Maybe Integer
squareAndSquareRoot2 m n = case square m of
                            Nothing → Nothing -- square mが失敗したら→失敗
                            Just mm → case square n of -- square mが成功
                                        Nothing → Nothing -- square nが失敗したら→失敗
                                        Just nn → squareRoot (mm * nn) -- square mもsquare nも成功
```

Maybeモナド(3)

なぜ冗長な記述になってしまうのか？

→計算の持つ『失敗するかもしれない』という性質をうまく組み合わせることができていないから

→i.e. 『失敗するかもしれない』という文脈をうまく扱えていない

Maybeモナド(4)

そこでMaybeモナド

Maybeモナドを使えば、先程の冗長な関数は以下のように記述できる。

```
squareAndSquareRoot2 :: Integer → Integer → Maybe Integer
squareAndSquareRoot2 m n = do
  mm ← square m
  nn ← square n
  squareRoot (mm * nn)
```

こんな感じで、文脈を伴う計算同士をいい感じに組み合わせることができるようにするのがモナド

関数プログラミングの影響 ~その他~

その他にも関数プログラミングの影響を受けた機能は色々あります。

- RustやSwiftのデータ型定義とパターンマッチ
- C++のコンパイル時計算
- Swift・Rubyの演算子定義
- Python, C#のリスト内包表記
- 型システムの強化(型推論等)

等々

関数プログラミングをやってみる

最低限の知識解説

- 実行環境構築方法
- 最低限の文法
- 問題(1)
- 問題(2)

実行環境作成方法

- Web
  - https://paiza.io/ja/projects/new
- ローカル

```
$ brew install sbt
$ sbt console
```

最低限の文法(1)

List

ScalaではListは以下のように定義できます。

```
val lst = List(1, 2, 3, 4, 5)
// lst: List[Int] = List(1, 2, 3, 4, 5)
```

::（コンスと読みます）は既にあるListの先頭に要素をくっつけるメソッドです。

```
val a1 = 1 :: Nil
// a1: List[Int] = List(1)

val a2 = 2 :: a1
// a2: List[Int] = List(2, 1)

val a3 = 3 :: a2
// a3: List[Int] = List(3, 2, 1)

val a4 = 4 :: a3
// a4: List[Int] = List(4, 3, 2, 1)

val a5 = 5 :: a3
// a5: List[Int] = List(5, 3, 2, 1)
```

最低限の文法(2)

foldLeft

foldLeftはScalaにとって非常に基本的なメソッドです。(Haskellにおけるfoldl)

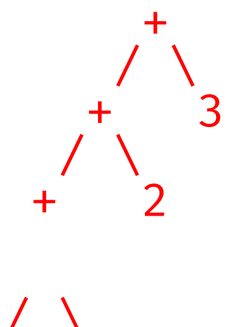
```
def foldLeft[B](z: B)(f: (B, A) ⇒ B): B
```

イメージを湧きやすくするため以下のサンプルコードを使ってfoldLeftの動作を説明します。

```
List(1, 2, 3).foldLeft(0)((x, y) ⇒ x + y)
```

上記の処理は

- 初期値: 0
- List(1, 2, 3)を左からたどって`(x, y) ⇒ x + y`を適用していく



問題(1)

reverse

`foldLeft`を用いて、Listの要素を反転させる次のシグニチャを持ったメソッドreverseを実装してください。

```
object Main extends App{
    def reverse[T](list: List[T]): List[T] = ???

    print(reverse(List(1, 2, 3, 4, 5))) // List(5, 4, 3, 2, 1)
}
```

問題(1)の答え

```
object Main extends App{
    def reverse[T](list: List[T]): List[T] = list.foldLeft(Nil: List[T])((a, b) ⇒ b :: a)

    print(reverse(List(1, 2, 3, 4, 5))) // List(5, 4, 3, 2, 1)
}
```

問題(2)

map

以下の???を埋めてmapメソッドをfoldLeftとreverseを使って実装してください。

```
object Main extends App{
    def reverse[T](list: List[T]): List[T] = list.foldLeft(Nil: List[T])((a, b) ⇒ b :: a)

    def map[T, U](list: List[T])(f: T ⇒ U): List[U] = ???
}
```

map メソッドは次のようにして使います。

```
println(map(List(1, 2, 3))(x ⇒ x + 1)) // List(2, 3, 4)
println(map(List(1, 2, 3))(x ⇒ x * 2)) // List(2, 4, 6)
println(map(List[Int]())(x ⇒ x * x)) // Nil
println(map(List(1, 2, 3))(x ⇒ 0)) // List(0, 0, 0)
```

問題(2)の答え

```
object Main extends App{
    def reverse[T](list: List[T]): List[T] = list.foldLeft(Nil: List[T])((a, b) ⇒ b :: a)

    def map[T, U](list: List[T])(f: T ⇒ U): List[U] = {
      list.foldLeft(Nil:List[U]){(x, y) ⇒ f(y) :: x}.reverse
    }

    println(map(List(1, 2, 3))(x ⇒ x + 1)) // List(2, 3, 4)
    println(map(List(1, 2, 3))(x ⇒ x * 2)) // List(2, 4, 6)
    println(map(List[Int]())(x ⇒ x * x)) // Nil
    println(map(List(1, 2, 3))(x ⇒ 0)) // List(0, 0, 0)
}
```