

WHAT'S THE FUN IN FUNCTIONAL PROGRAMMING?

Presented by n0-t0



関数型プログラミングって何？

手続き型

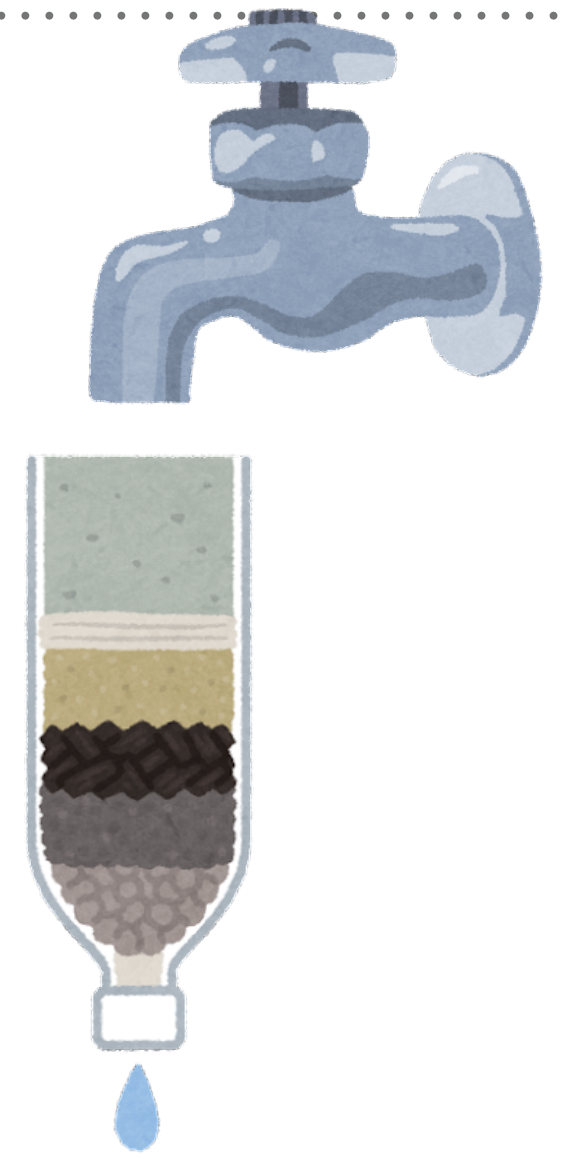


A する.
その後 B なら C する.
その後 D 回 E する.

時間の概念がある

「関数(メソッド)」は関数ではない
文(~する)が主体

関数型



A が来たら B を出す
関数を C とする.
D を C に通す.

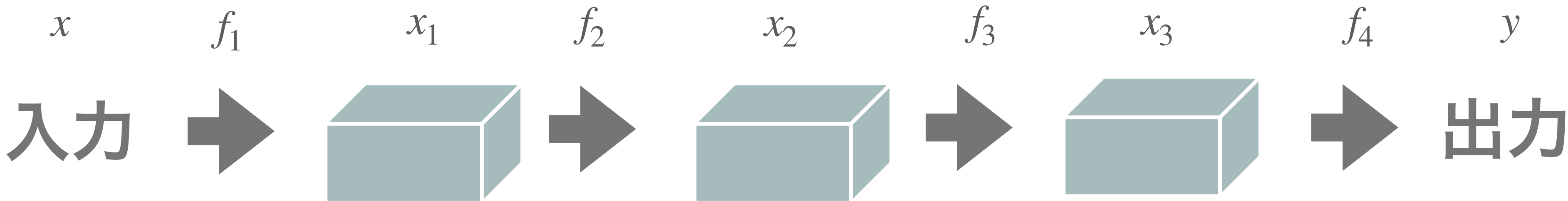
時間の概念はあまりない

いつも同じ入出力関係の関数を使う
式が主体

関数が主であるとは

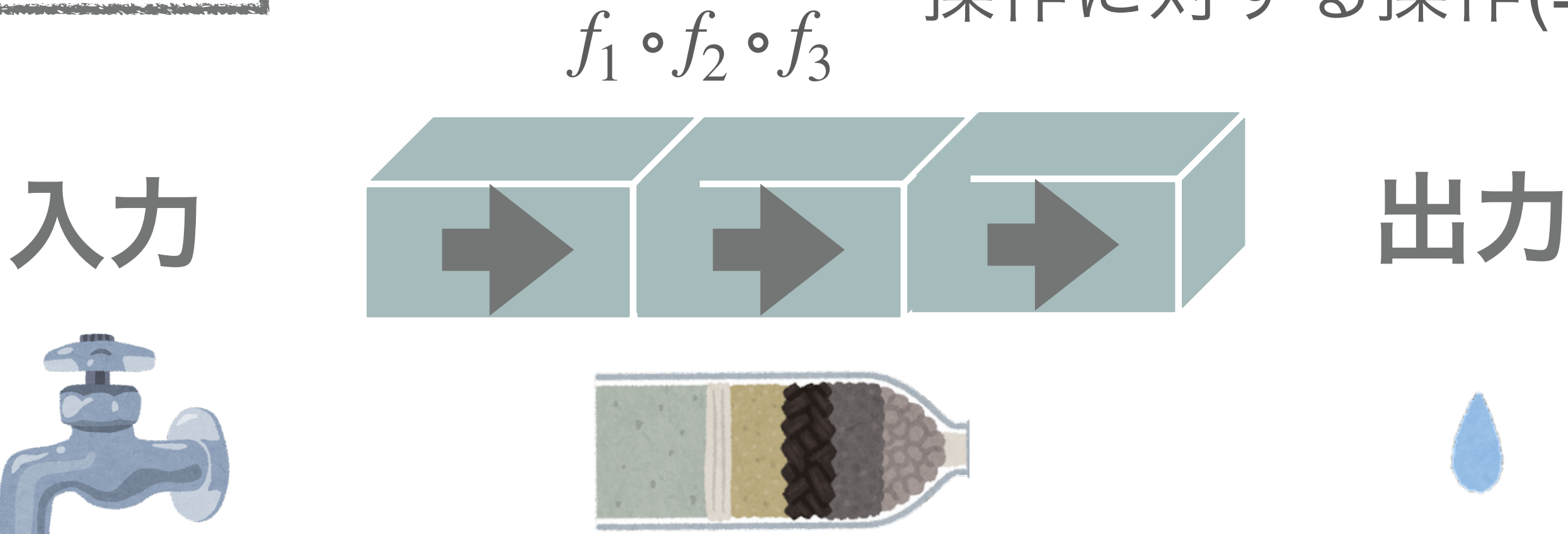
$$y = f_4(f_3(f_2(f_1(x))))$$

主語は実データ
操作によって順次変換する



$$y = (f_4 \circ f_3 \circ f_2 \circ f_1)(x)$$

主語は操作そのもの
操作に対する操作(=高階関数)を使用する



高階関数はどこにでも現れる

MAP

```
ls = [1, 2, 3, 4, 5]
doubled_sum = sum(map(lambda x: x * 2, ls))
print(doubled_sum)
# >> 30
```

$$\sum f(\mathbf{x}) \text{ where } f(\mathbf{x}) = \mathbf{x} * 2$$

FILTER

```
names = ['Alice', 'Bob', 'Charlie', 'David', 'Alexandria']
long_names1 = list(filter(lambda name: len(name) > 5, names))
print(long_names1)
# >> ['Charlie', 'Alexandria']

long_names2 = [name for name in names if len(name) > 5]
print(long_names2)
# >> ['Charlie', 'Alexandria']
```

$$N_{long} = \{n \in N \mid n > 5\}$$

関数型スタイルのメリット・デメリット

メリット

- 宣言的になりやすくなる(how?ではなくwhat?が書かれる)
- 副作用を閉じ込めて、プログラム全体を単純にできる
- 参照透過性により、局所的な推論が容易になる
- 各種の抽象化の恩恵を生かすことができる

デメリット

- 当然、抽象化にはオーバーヘッドがある
- 前提知識が必要な抽象化が多用される

Topic

A. Recursion-scheme で HMM Viterbi

B. Monad で小さな Parser ツールキット

A.0. 関数型でも動的計画法(DP)がしたい

一般的なDP

先に確保してループ

DP行列を確保して...

1			

埋めていく...

3	2		
1	2		

行列を畳み込む

6	6	5	5
5	5	4	4
4	4	3	6
3	2	3	5
1	2	3	4

求める値

p

純粹関数型スタイルでは


初期化値に新しい値を入れ直すのは✖

再帰的に構造を作る

1

初期化

3	
1	2



動的に構造を構築する

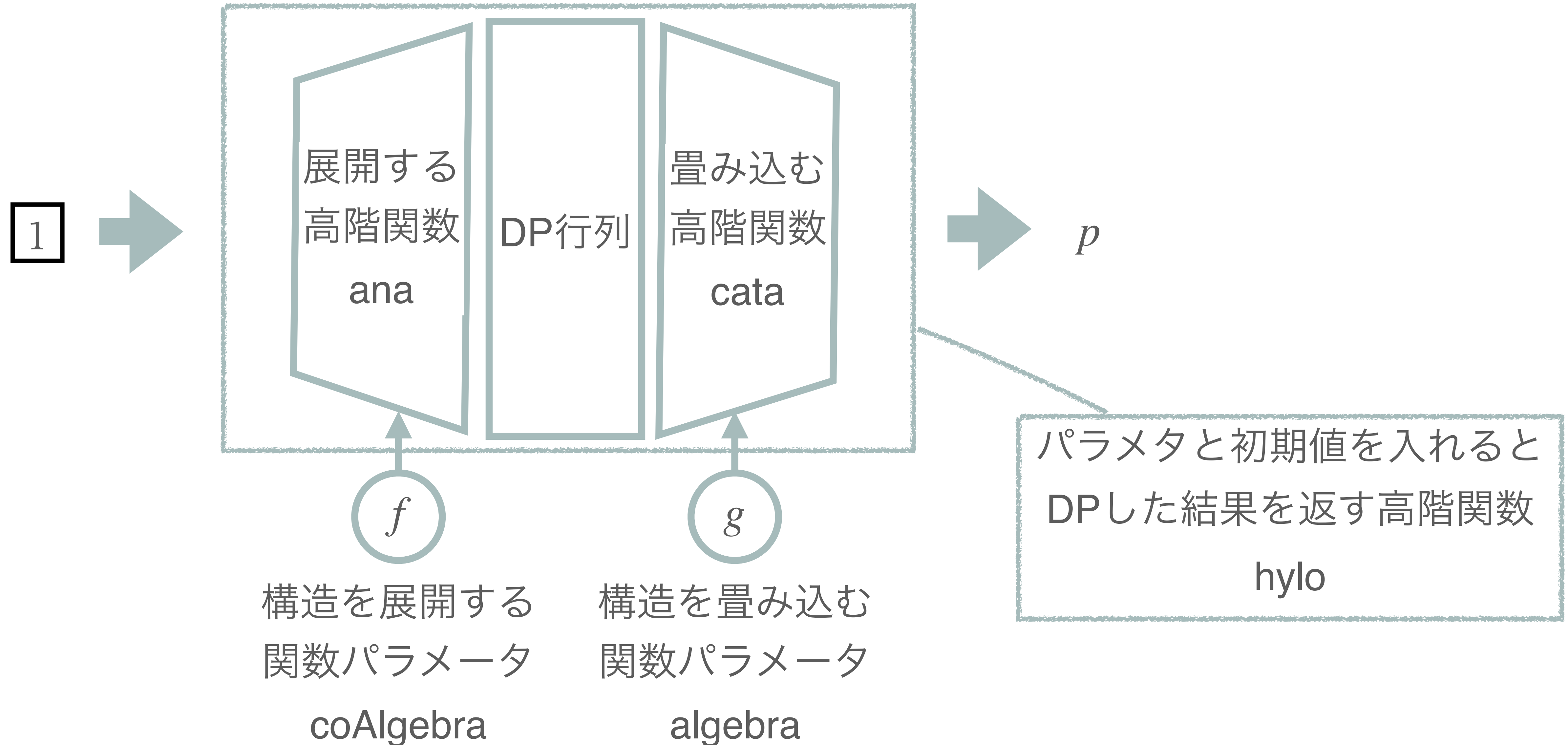
6	6	5	5
5	5	4	4
4	4	3	6
3	2	3	5
1	2	3	4

求める値

p

構造を畳み込む

A.1. 高階関数によるDPのパターン化



A.2. RECURSION-SCHEME : 再帰のパターン

```
type Fn[A, B] = Callable[[A], B]
```

```
class Functor[A](Protocol):  
    def fmap[B](self, f: Fn[A, B]) -> "Functor[B]":  
        ...
```

mapを持つ型Functorを用意

```
@dataclass(frozen=True)
```

```
class Fix[X]:  
    unfix: Functor["Fix[Functor[X]]"]
```

再帰型の不動点

```
def cata[A](algebra: Fn[Functor[A], A]) -> Fn[Fix[Functor], A]:  
    return lambda fix: algebra(fix.unfix.fmap(cata(algebra)))
```

ana/cata/hyloを再起的に定義

```
def ana[B](coAlgebra: Fn[B, Functor[B]]) -> Fn[B, Fix[Functor]]:  
    return lambda b: Fix(coAlgebra(b).fmap(ana(coAlgebra)))
```

```
def hylo[A, B](algebra: Fn[Functor[B], B], coAlgebra: Fn[A, Functor[A]]) -> Fn[A, B]:  
    return lambda a: cata(algebra)(ana(coAlgebra)(a))
```

A.3. 実際にVITERBIアルゴリズムをやってみる1

- Recursion-scheme自体には再帰するデータ構造の情報が入っていない。
- 配列、行列、木、DAGなど、どの構造をどのように走査するかの情報も与える

```
type Control[A] = "Continue[A]" | "End[A]"

@dataclass(frozen=True)
class Continue[A]:
    next: A
    def fmap[B](self, f: Fn[A, B]) -> "Continue[B]":
        return Continue(f(self.next))

@dataclass(frozen=True)
class End[A]:
    final: A
    def fmap[B](self, f: Fn[A, B]) -> "End[B]":
        return End(self.final)
```

今回は配列を走査するので、
配列の途中 or 終わりをFunctorとして書く

A.4. 実際にVITERBIアルゴリズムをやってみる2

```
@dataclass(frozen=True)
class Trace:
    outputs: list[Output]
    history: list[dict[State, tuple[State, float]]]
    viterbi: dict[State, float]
    optimal_path: list[State] | None

def hmmCoAlgebra(trace: Trace) -> Control[Trace]:
    match trace:
        case Trace([], history, viterbi, _):
            return End(Trace([], history, viterbi, None))
        case Trace(outputs, history, viterbi, _):
            # ...
            return Continue(Trace(outputs[1:], history + [v_pt_max_and_trace], viterbi_new, None))

def hmmAlgebra(ctrl: Control[Trace]) -> Trace:
    match ctrl:
        case Continue(a):
            # ...
        case End(a):
            # ...
            return Trace([], a.history, a.viterbi, [pi])

print(hylo(hmmAlgebra, hmmCoAlgebra)(initialState).optimal_path)
```

HMM用のパラメータ関数を書く

- 展開時は制御情報をつける
- 畳み込み時は外す

hyloにhmmAlgebra, hmmCoAlgebraを
与えて呼び出す(この行で全てが行われる)

Topic

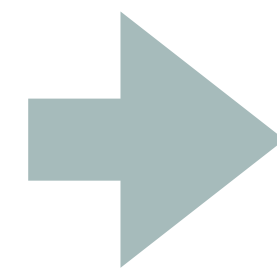
A. Recursion-scheme で HMM Viterbi

B. Monad で小さな Parser ツールキット

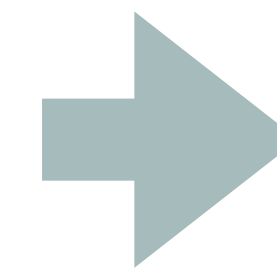
B.O. PARSEを関数で作る

構造化されていない入力文字列

>id_0001¥nATCGC...



Parser



構造化データ(A型)

```
{  
  Header: id_0001,  
  Sequence: ATCGC...  
}
```

parse: String => 成功(A, String) or 失敗

を持つ型をParserとして、

```
class Parser[A](ABC):  
  @abstractmethod  
  def parse(self, s: str) -> Either[str, (A, str)]:  
    pass
```

簡単なParserを合成して複雑なParserにしていく

B.1. FLATMAPによる文脈付き関数の合成1

parse: String => 成功(A, String) or 失敗はString => Stringのように簡単に合成できない。

bind(=flatMap)は文脈付きの関数を合成する操作



B.2. FLATMAPによる文脈付き関数の合成2

```
class Parser[A](ABC):
  def bind[B](self, f: Callable[[A], "Parser[B]"]) -> "Parser[B]":
    return DefaultParserCombinator._bind(self, f)

class DefaultParserCombinator:
  @staticmethod
  def _bind[X, Y](p: Parser[X], f: Callable[[X], Parser[Y]]) -> Parser[Y]:
    class BindParser(Parser[Y]):
      def parse(self, s: str) -> Either[str, (Y, str)]:
        return p.parse(s).bind(lambda x: f(x[0]).parse(x[1]))
    return BindParser()

print(
  DefaultParserCombinator.char("h")
    .bind(lambda x: DefaultParserCombinator.char("e").fmap(lambda y: x + y))
    .bind(lambda y: DefaultParserCombinator.char("l").fmap(lambda z: y + z))
    .parse("hello")
)
```

bind(lambda x: でParser同士を合成できる

B.3. PARSER MONAD1

合成のたびにbindするのは大変なので、
GeneratorとDecoratorで自動で合成してくれるようにする

```
def new_parser(a):  
    v1 = yield parser1(a)  
    v2 = yield parser2(v1)  
    return v1 + v2
```

のように合成できるようになる
(所謂do記法・内包表記)

合成したparseを持つ新しいParserを返すデコレータを書く

```
def parser_monad(comprehension):  
    @functools.wraps(comprehension)  
    def computation[X](args, **kwargs) -> Parser[X]:  
        class CombinedParser(Parser[X]):  
            def parse(self, s: str) -> Either[str, (X, str)]:  
                iter = comprehension(*args, **kwargs)  
                res_s = s  
                try:  
                    result_parser: Parser = next(iter)  
                    while True:  
                        result: Either[str, (X, str)] = result_parser.parse(res_s)  
                        print(result)  
                        match result.value:  
                            case Left(_):  
                                return result  
                            case Right(v):  
                                parsed_value, res_s = v  
                                result_parser = iter.send(parsed_value)  
                    except StopIteration as e:  
                        return Either.right((e.value, res_s))  
                return CombinedParser()  
        return computation
```

B.4. PARSER MONAD2

```
@parser_monad
def commaAndValue[X](p: Parser[X]) -> Parser[X]:
  _ = yield DefaultParserCombinator.token(DefaultParserCombinator.char(","))
  v = yield DefaultParserCombinator.token(p)
  return v
```

“, 値”のparser

```
@parser_monad
def commaSeparated[X](p: Parser[X]) -> Parser[list[X]]:
  v = yield p
  vs = yield DefaultParserCombinator.many(
    DefaultParserCombinator.commaAndValue(p)
  )
  return [v] + vs
```

“値, 値, ...”のparser

B.5. PARSER MONAD3

```
@parser_monad
def nonEmptyList[X](p: Parser[X]) -> Parser[list[X]]:
  _ = yield DefaultParserCombinator.token(DefaultParserCombinator.char("["))
  v = yield DefaultParserCombinator.commaSeparated(p)
  _ = yield DefaultParserCombinator.token(DefaultParserCombinator.char("]"))
  return v
```

空でないリスト “[値, 値, ...]” の parser

```
@parser_monad
def emptyList[X]() -> Parser[list[X]]:
  _ = yield DefaultParserCombinator.token(DefaultParserCombinator.char("["))
  _ = yield DefaultParserCombinator.token(DefaultParserCombinator.char("]"))
  return []
```

空リスト “[]” の parser

```
@parser_monad
def list_literal[X](p: Parser[X]) -> Parser[list[X]]:
  v = yield DefaultParserCombinator.either(
    DefaultParserCombinator.nonEmptyList(p), DefaultParserCombinator.emptyList()
  )
  return v
```

リストリテラルの parser

Csv, Fasta等も同様にできます...

Topic

A. Recursion-scheme で HMM Viterbi

B. Monad で小さな Parser ツールキット

Fin

いかがでしたでしょうか？

少しでも関数型プログラミングに興味を持ってもらえたら嬉しいです