関数型プログラミング

プログラム言語論

目次

- 関数型 vs. 命令型プログラミング
- パターンマッチング
- 参照透明性
- 遅延評価
- 再帰
- 高階関数とカリー化

https://ja.pngtree.com/freepng/

目次

- **関数型 vs. 命令型プログラミング**
- パターンマッチング
- 参照透明性
- 遅延評価
- 再帰
- 高階関数とカリー化

歴史を簡単に

Lambda Calculus (Church, 1932-33)	計算の形式的モデル
Lisp (McCarthy, 1960)	Listを特徴とする記号計算
APL (Iverson, 1962)	配列を特徴とする代数的プログラミング
ISWIM (Landin, 1966)	let と where 句 等式推論; 純粋な関数型プログラミングの誕生
ML (Edinburgh, 1979)	源は定理証明用メタ言語
SASL, KRC, Miranda (Turner, 1976-85)	遅延評価
Haskell (Hudak, Wadler, et al., 1988)	関数型言語の大統一

状態を用いないプログラミング

命令型のスタイル:

```
n := x;
a := 1;
while n>0 do
begin a:= a*n;
   n := n-1;
end;
```

宣言(関数)型のスタイル:

```
fac n =
   if     n == 0
   then   1
   else   n * fac (n-1)
```

純関数型言語のプログラムには明示的な状態はない.
プログラムは、完全に、式の合成で構成されている.

純関数型プログラム言語

命令型プログラミング:

✓ Program = Algorithms + Data

関数型プログラミング:

- Program = Functions ∘ Functions

プログラムとは何?

- プログラムとは、入力データを出力データに変換する手順を書いたもの.

純関数型言語の特徴

1. 全てのプログラム・手続きは関数である
2. どこにも 変数や代入はない — あるのは入力パラメータだけ
3. どこにも 繰返し(loops)はない — あるのは再帰だけ
4. 関数の戻り値は、パラメータ(引数)値のみに依存している
5. 関数は *第一級市民(オブジェクト)である*

Haskell とは何か?

Haskell は汎用の純粋関数型プログラミング言語で、プログラミング言語設計の分野での最近の革新の多くが取入れられている。

Haskell が提供するのは、高階関数、非正格の意味論、静的多相型付け、利用者定義の代数的データ型、パターン照合、リストの内包表記,モジュールシステム、モナド I/O システムである。

また、さらに次のような豊かなプリミティブデータ型が用意されている。リスト、配列、任意倍長整数、固定倍長整数、浮動小数点数。

Haskell は遅延評価型関数型言語に関する長年の研究を凝縮したものであり、また、その頂点にたつものである。.

— The Haskell 98 report

https://www.sampou.org/haskell/report-revised-j/intro.html#sect1

お決まりの "Hello World"

```
hello() = print "Hello World"
```


目次

- 関数型 vs. 命令型プログラミング
- **パターンマッチング**
- 参照透明性
- 遅延評価
- 再帰
- 高階関数とカリー化

パターンマッチング

Haskell は、場合分けを用いた関数定義を記述する複数の方法を用意している:

パターン:

```
fac' 0  = 1
fac' n  = n * fac' (n-1)

-- または: fac' (n+1) = (n+1) * fac' n
```

ガード:

```
fac'' n   | n == 0  = 1
          | n >= 1  = n * fac'' (n-1)
```


リスト

リスト は、「要素」と「要素のリスト」の 対 である:

- ✓ `[ ]` — 空リスト
- ✓ `x:xs` — リストであって、ヘッドが `x`, リストの残りの部分が `xs` であるもの

下述の短縮形を用いると、より一層便利に使うことができる

- ✓ `[1,2,3]` — は `1:2:3:[ ]` の糖衣構文
- ✓ `[1..n]` — は `[1,2,3, ... n]` の略

糖衣構文(とういこうぶん、英: syntactic sugarあるいはsyntax sugar)は、プログラミング言語において、読み書きのしやすさのために導入される書き方であり、複雑でわかりにくい書き方と全く同じ意味になるものを、よりシンプルでわかりやすい書き方で書くことができるもののことである。 https://ja.wikipedia.org/wiki/糖衣構文

リストを用いて

リストはパターンで 解体 できる:

```
head (x:_)    = x

len [ ]       = 0
len (_:xs)    = 1 + len xs

prod [ ]      = 1
prod (x:xs)   = x * prod xs

fac''' n      = prod [1..n]
```


目次

- 関数型 vs. 命令型プログラミング
- パターンマッチング
- **参照透明性**
- 遅延評価
- 再帰
- 高階関数とカリー化

参照透明性

関数が 参照透明性 を持つとは 関数の値がパラメータ(引数)値のみに依存する場合 *である.*

- *f(x)+f(x)* は *2*f(x)* に等しいか? Cでは? Haskellでは?

参照透明性があると "equals can be replaced by equals" (等しいものは等しいものと置き換えられる).

純関数型言語においては、すべての関数に参照透明性があり、従って、それは何度呼び出されても いつも同じ結果を返す.

式の評価

式を(形式的に)評価するには、関数本体中の仮引数を実引数で置き換える

```
fac 4
→ if 4 == 0 then 1 else 4 * fac (4-1)
→ 4 * fac (4-1)
→ 4 * (if (4-1) == 0 then 1 else (4-1) * fac (4-1-1))
→ 4 * (if 3 == 0 then 1 else (4-1) * fac (4-1-1))
→ 4 * ((4-1) * fac (4-1-1))
→ 4 * ((4-1) * (if (4-1-1) == 0 then 1 else (4-1-1) * …))
→ …
→ 4 * ((4-1) * ((4-1-1) * ((4-1-1-1) * 1)))
→ …
→ 24
```

勿論, 実際の関数型言語においては、このような構文的な置き換えで実装がなされるわけではない ...

目次

- 関数型 vs. 命令型プログラミング
- パターンマッチング
- 参照透明性
- **遅延評価**
- 再帰
- 高階関数とカリー化

遅延評価

"Lazy (遅延)" または "normal-order (正規順序)" 評価方法では、式は、実際にその値が必要になった時のみ評価される. うまい実装方法 (Wadsworth, 1971) を用いれば複製した式は共有することができ、不要な再計算を避けることができる.

従って:

```
sqr n = n * n
```

```
sqr (2+5) → (2+5) * (2+5) → 7 * 7 → 49
```

遅延評価を用いると、間違った式や、(評価が)停止しない式を引数に持っていたとしても、関数の評価ができる:

```
ifTrue True x y  = x
ifTrue False x y = y
```

```
ifTrue True 1 (5/0) → 1
```

遅延リスト

遅延リスト は 無限長のデータ構造 であり、必要に応じて生成されるものである:

```
from n = n : from (n+1)
                    from 100 → [100,101,102,103,....
take 0 _            = [ ]
take _ [ ]          = [ ]
take (n+1) (x:xs)   = x : take n xs
```

```
take 2 (from 100)  → take 2 (100:from 101)
                   →100:(take 1 (from 101))
                   →100:(take 1 (101:from 102))
                   →100:101:(take 0 (from 102))
                   →100:101:[]   → [100,101]
```

NB: この遅延リスト (from n) は特別の構文を用いている: [n..]

遅延リストのプログラミング

シーケンスは、自然に、遅延リストとして実装されるものが多い
NB トップダウンかつ宣言的スタイル:

```
fibs = 1 : 1 :  fibsFollowing 1 1
     where fibsFollowing a b =
        (a+b) : fibsFollowing b (a+b)
```

```
take 10 fibs
→ [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

- *fibs の定義を(a+b) が一度しか現れないように書き換えれるだろうか?*

宣言的なプログラミングスタイル

```
primes = primesFrom 2
primesFrom n = p : primesFrom (p+1)
   where p = nextPrime n
nextPrime n
   | isPrime n       = n
   | otherwise       = nextPrime (n+1)
isPrime 2            = True
isPrime n            = notDivisible primes n
notDivisible (k:ps) n
   | (k*k) > n       = True
   | (mod n k) == 0  = False
   | otherwise       = notDivisible ps n
```

```
take 100 primes → [ 2, 3, 5, 7, 11, 13, ... 523, 541 ]
```

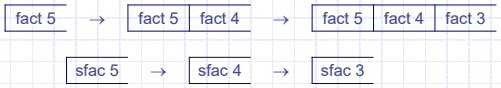

目次

- 関数型 vs. 命令型プログラミング
- パターンマッチング
- 参照透明性
- 遅延評価
- **再帰**
- 高階関数とカリー化

末尾再帰（Tail Recursion）

再帰関数は、繰り返しよりも効率性が低くなりうる. 殆どのハードウェアで、手続き呼び出しのコストが高い からである.

再帰が、最後の演算であり、しかも自分の 末尾再帰 呼び出しとなっている場合には、その再帰呼び出しは、最近のコンパイラでは、最適化により消去 (optimized away) している. 実行時スタックが一つあればよいからである:



末尾再帰...

再帰関数は、計算の途中経過を、明示的に、引数とすることにより、末尾再帰に変換することができる:

```
sfac s n = if    n == 0
           then  s
           else  sfac (s*n) (n-1)
```

```
sfac 1 4 → sfac (1*4) (4-1)
         → sfac 4 3
         → sfac (4*3) (3-1)
         → sfac 12 2
         → sfac (12*2) (2-1)
         → sfac 24 1
         → ...
         → 24
```

多重再帰

再帰をよく考えずに使うと、不必要な再計算をすることになる

```
fib 1       = 1
fib 2       = 1
fib (n+2)   = fib n + fib (n+1)  -- NB: 末尾再帰ではない!
```

効率向上は可能である: 途中経過を明示的に受け渡せばよい

```
fib' 1      = 1
fib' n      = a      where (a,_) = fibPair n
fibPair 1   = (1,0)
fibPair (n+2) = (a+b,a)
        where (a,b) = fibPair (n+1)
```

- では、Fibonacci 関数は末尾再帰にできるであろうか?

目次

- 関数型 vs. 命令型プログラミング
- パターンマッチング
- 参照透明性
- 遅延評価
- 再帰
- **高階関数とカリー化**

高階関数

高階関数は他の関数を *第一級オブジェクト* として扱い、新しい関数を合成する.

```
map f [ ]      = [ ]
map f (x:xs)   = f x : map f xs
```

```
map fac [1..5]
→ [1, 2, 6, 24, 120]
```

NB: `map fac` は関数でリストに適用できるものである:

```
mfac = map fac
```

```
mfac [1..3]
→ [1, 2, 6]
```


無名関数

無名関数は "λ抽象 (lambda abstractions)" として書くことができる.
関数 `(¥x -> x * x)` は `sqr` と同様の動きをする:

```
sqr x = x * x
```

無名関数をバックスラッシュで表す
バックスラッシュはλのつもり

```
sqr 10
→ 100
(¥x -> x * x) 10
→ 100
```

無名関数も第一級オブジェクトである:

```
map (¥x -> x * x) [1..10]
→ [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```


カリー化

カリー化された関数 [論理学者 H.B. Curry にちなむ] は 引数を一時に一個とる関数であり, その結果高階関数として取り扱える.

```
plus x y      = x + y     -- カリー化した加算
```

```
plus 1 2
→ 3
```

```
plus'(x,y)    = x + y     -- 普通の加算
```

```
plus'(1,2)
→ 3
```


カリー化された関数の考え方

無名関数をバックスラッシュで表す
バックスラッシュはλのつもり

`plus x y = x + y` は右と同じ: `plus x = ¥y -> x+y`

言い換えると, `plus` は結果として関数を返す1引数の関数 である.

`plus 5 6` は右と同じ: `(plus 5) 6`

言い換えると, `(plus 5)` を起動すると, 次の関数が得られ,

```
¥y -> 5 + y
```

それは引数 `6` を引き渡して、結果として `11` を得る

カリー化された関数を用いて

カリー化された関数は便利である. 引数を少しずつ結合することができるからである

```
inc  = plus 1              -- 第一引数を 1 に結合

inc 2
→ 3


fac  = sfac 1              -- 結合する第一引数は
     where sfac s n        -- カリー化された関数のもの
        | n == 0   = s
        | n >= 1  = sfac (s*n) (n-1)
```


カリー化

以下の (予め定義した) 関数は2引数関数を引数とし、それをカリー化された関数として返す:

```
curry f a b  = f (a, b)

plus(x,y)    = x + y               -- カリー化していない!
inc          = (curry plus) 1

sfac(s, n)   =  if n == 0          -- カリー化していない
                then s
                else sfac (s*n, n-1)

fac = (curry sfac) 1               -- 第一引数を結合
```