Scalaで自作するプログラミング言語処理系

無線部開発班　2022年2月5日

https://pafelog.net

目次

第1章　言語処理系を作る

本書では、**ラムダ計算**を理論的背景に持つ独自のプログラミング言語の**インタプリタ**を自作して、**計算論**の基礎を学ぶ。実装は公開済みで気軽に閲覧できる。最低限の機能に留めた簡素な言語だが、改良次第では汎用的な言語に拡張できる。

```
$ git clone https://github.com/nextzlog/fava
```

改良には開発環境が必要だが、殆どの場合は**パッケージリポジトリ**経由で、即座に調達できる。ArchLinux の例を示す。

```
$ sudo pacman -S jdk-openjdk scala
```

頒布版には、**対話環境**を同梱した。また、第 2 章に述べる様々な計算モデルの実験機能も同梱した。起動時に機能を選ぶ。

```
$ java -jar build/libs/fava.jar
which language do you use?
[0]: fava
[1]: math
select: 0
fava$ "HELLO, WORLD!"
HELLO, WORLD!
```

自作言語の名前は fava とする。専用の命令を実装した仮想的な実行環境で動作する。算術演算や論理演算が実行できる。

```
fava$ 11 + 4 * 5 - 14
17
```

理念的には**純粋関数型言語**に分類できる。式に**副作用**がなく、式の値は定数で、式の構造から明確に求まる特徴を持つ。また、関数は関数の引数や返り値にできる。ただし、関数は名前も局所変数も定義できず、数学的な関数の概念に似る。

```
fava$ ((x)=>(y)=>3*x+7*y)(2)(3)
27
```

実用的な計算が困難な程の制約に思えるが、実際には第 3 章で解説する通り、優れた計算能力を持つ。階乗も計算できる。

```
fava$ ((f)=>((x)=>f(x(x)))((x)=>f(x(x))))((f)=>(n)=>(n==0)?1:n*f(n-1))(10)
3628800
```

理論的には、**帰納的に枚挙可能**な集合に対する、任意の計算を実行できる。自然数を関数で表現する芸当も可能である。

```
fava$ ((f,x)=>f(f(f(f(f(x)))))((x)=>x+1,0) // 0 + 1 + 1 + 1 + 1 + 1
5
```

また、任意の**順序組**を定義して操作できる。順序組は構造体と等価である。理論的には複雑な**グラフ構造**も表現できる。

```
fava$ ((pair)=>pair((car,cdr)=>car))(((car,cdr)=>(z)=>z(car,cdr))(12,34))
12
fava$ ((pair)=>pair((car,cdr)=>cdr))(((car,cdr)=>(z)=>z(car,cdr))(12,34))
34
```

対話環境の拡張機能として、実行環境で実行される命令列を表示する機能も用意した。実行環境の動作の理解に役立つ。

```
fava$ compile(1 + 2)
Push(1) Push(2) Add
```

第2章　計算モデルを作る

言語処理系とは、言語仕様に沿って書かれた計算手順を読み取り、任意の計算機を構築または模倣する**万能機械**である。計算機を抽象化した数学的な模型を**計算モデル**と呼ぶ。例えば、論理回路は第 2.1 節に述べる**有限状態機械**で表現できる。

2.1　有限状態機械

有限状態機械は、**状態**と**遷移規則**の有限集合で構成される。論理回路で言えば、**記憶素子**が保持する情報が状態である。有限状態機械に信号 x_n を与えると、Table 2.1 の遷移規則に従って、状態 q_n から状態 q_{n+1} に遷移して、信号 y_n を返す。

Table 2.1: state transition tables.

(a) SR flip-flop.

x_n	q_n	q_{n+1}	y_n
00	0	0	0
00	1	1	1
01	0	0	0
01	1	0	1
10	0	1	0
10	1	1	1
11	0	-	-
11	1	-	-

(b) JK flip-flop.

x_n	q_n	q_{n+1}	y_n
00	0	0	0
00	1	1	1
01	0	0	0
01	1	0	1
10	0	1	0
10	1	1	1
11	0	1	0
11	1	0	1

(c) 2bit counter.

x_n	q_n	q_{n+1}	y_n
0	00	00	00
1	00	01	00
0	01	01	01
1	01	10	01
0	10	10	10
1	10	11	10
0	11	11	11
1	11	00	11

有限状態機械が受け取る信号列を**文**と見做す場合もある。これを言語処理に応用する体系が、第 4 章の**言語理論**である。有限状態機械には、それに対応する**正規表現**が必ず存在する。この性質を利用して、正規表現の処理系を実装してみる。

```
class R[S](val test: Seq[S] => Option[Seq[S]])
```

正規表現は、正規表現を結合して、帰納的に定義できる。その最小単位が以下に示す `One` 型で、特定の 1 文字に適合する。

```
case class One[S](r: S) extends R[S](Some(_).filter(_.head == r).map(_.tail))
```

適合すると、残りの文字列を返す。ここに別の正規表現を適用すれば、正規表現の連結を意味する。これが `Cat` 型である。`Alt` 型は、正規表現の選択肢を表す。これは、遷移先の状態が複数ある状況を表す。これを遷移規則の**非決定性**と呼ぶ。

```
case class Cat[S](l: R[S], r: R[S]) extends R[S](seq => l.test(seq).map(r.test).flatten)
case class Alt[S](l: R[S], r: R[S]) extends R[S](seq => l.test(seq).orElse(r.test(seq)))
```

`Opt` 型は、指定された正規表現の省略可能な出現を表す。また、`Rep` 型は、指定された正規表現の 0 回以上の反復を表す。

```
case class Opt[S](r: R[S]) extends R[S](seq => r.test(seq).orElse(Some(seq)))
case class Rep[S](r: R[S]) extends R[S](seq => Cat(r, Opt(Rep(r))).test(seq))
```

正規表現 `Z(L+|G)O` に相当する、有限状態機械の実装例を示す。第 2.1 節の内容を応用すれば、言語処理系も実装できる。

```
val ZLO = Cat(One('Z'), Cat(Alt(Rep(One('L')), One('G')), One('O')))
println(if(ZLO.test("ZLLLLLLLLLLLLLLLO").isDefined) "OK" else "NO")
```

2.2 セルオートマトン

単体の有限状態機械は、再帰計算が苦手である。しかし、その集合体である**セルオートマトン**は、任意の計算ができる。構成単位を**セル**と呼ぶ。各セルは、近傍 k 個のセルの状態を参照し、式 (2.1) に示す遷移規則 δ に従って、状態遷移する。

$$\delta : Q^k \to Q. \tag{2.1}$$

空間的な自由を得た恩恵で、再帰構造を持つ計算に対応する。例えば、**フラクタル図形**を描画する遷移規則が存在する。Fig. 2.1 に示す**ラングトンの環**の例では、仮足を伸ばして式 (2.2) の遺伝子を注入し、分裂増殖する生物群集を模倣する。

$$07107107111110410410710710 71. \tag{2.2}$$

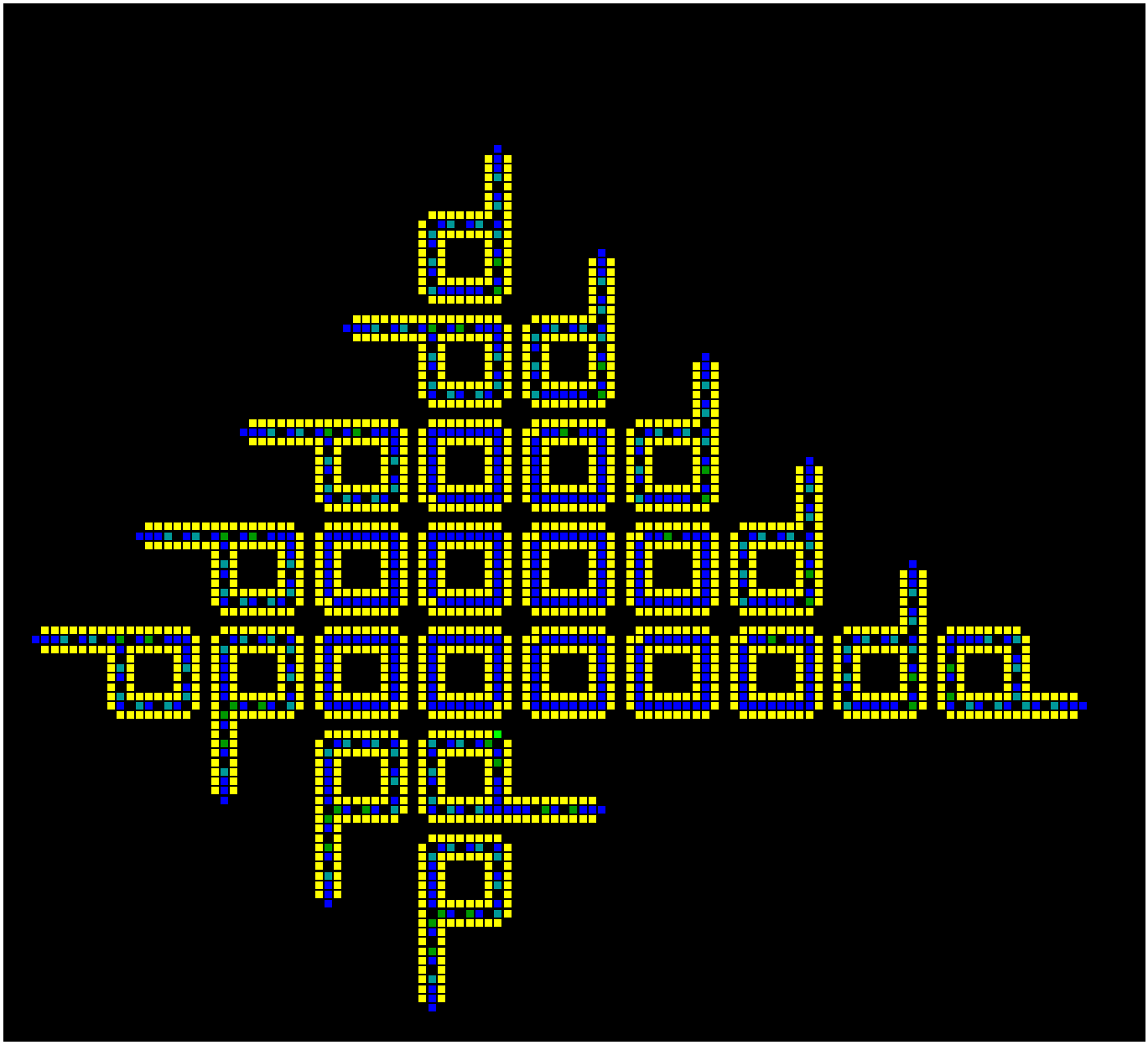
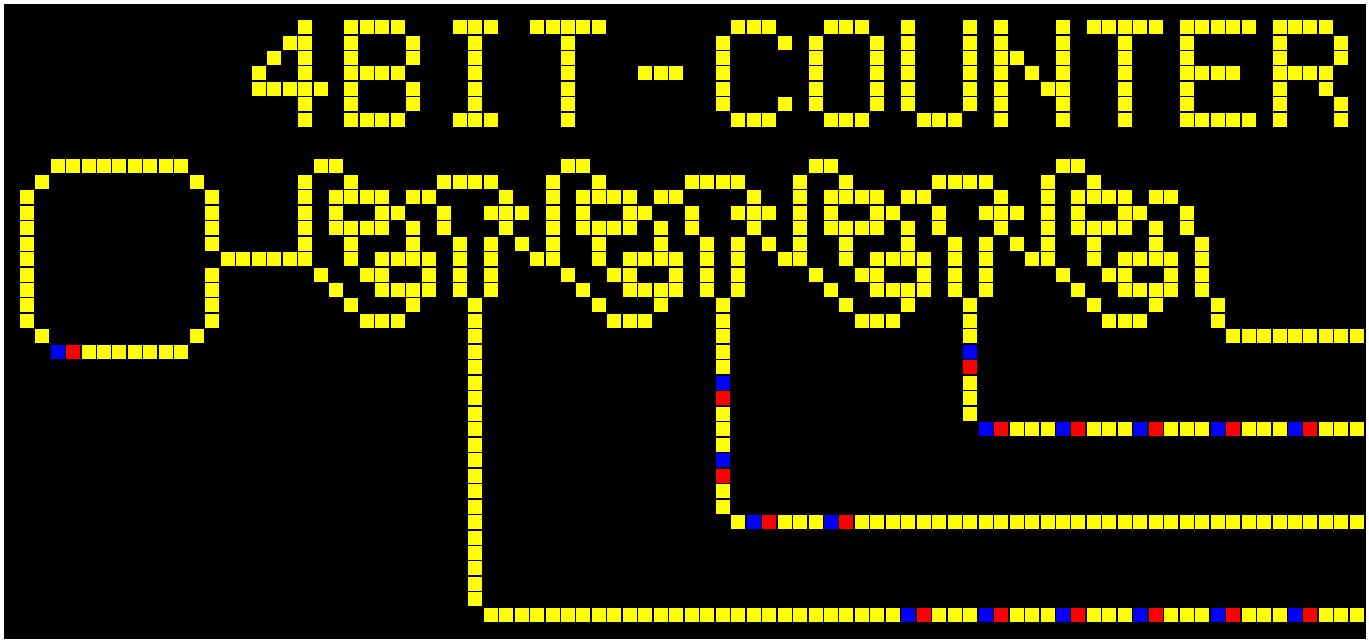


Fig. 2.1: Langton's loops in cellular automata.

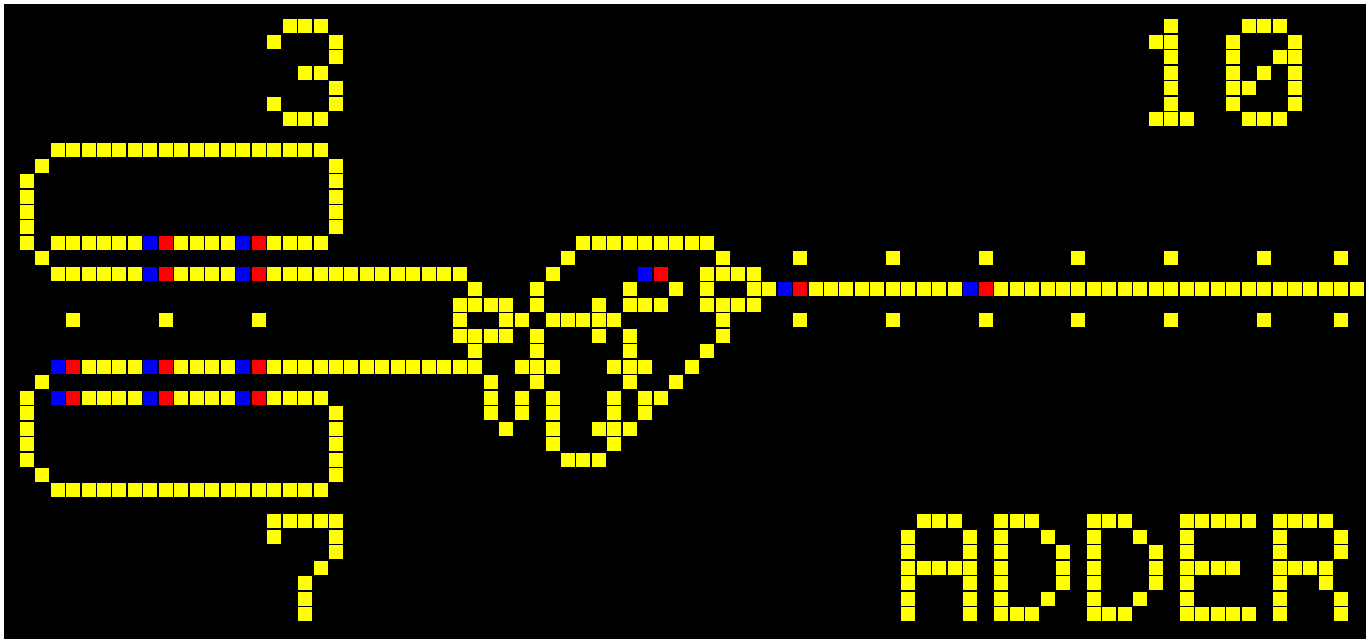
2 次元の実装例を示す。遷移規則は引数で指定する。全てのセルが同時に更新される**同期型セルオートマトン**を採用した。

```
class CA2[S](rule: Seq[Seq[S]] => S, d: Int = 1) {
  def ROI[V](i: Int)(s: Seq[V]) = Range.inclusive(i - d, i + d).map(Math.floorMod(_, s.size)).map(s)
  def apply(s: Seq[Seq[S]]) = s.indices.map(x => s(x).indices.map(y => rule(ROI(x)(s).map(ROI(y)))))
}
```

理論的には、任意の遷移規則を初期状態で受け取り、模倣する万能機械も構築できる。その例が**ワイヤワールド**である。黒の基板に黄色の配線を作ると、信号が配線を巡り、記憶素子を含む、様々な論理回路を模倣する。Fig. 2.2 に例を示す。



(a) 4bit counter.



(b) binary adder.

Fig. 2.2: Wireworld logic circuits.

Fig. 2.2(a) は**カウンタ**である。左側の**発振回路**から周期的に信号を送り込む度に、右側の配線 4 本の信号が切り替わる。Fig. 2.2(b) は**加算器**である。果ては、表示器を備えた計算機さえ実装可能な点が特徴だが、遷移規則は実に単純である。

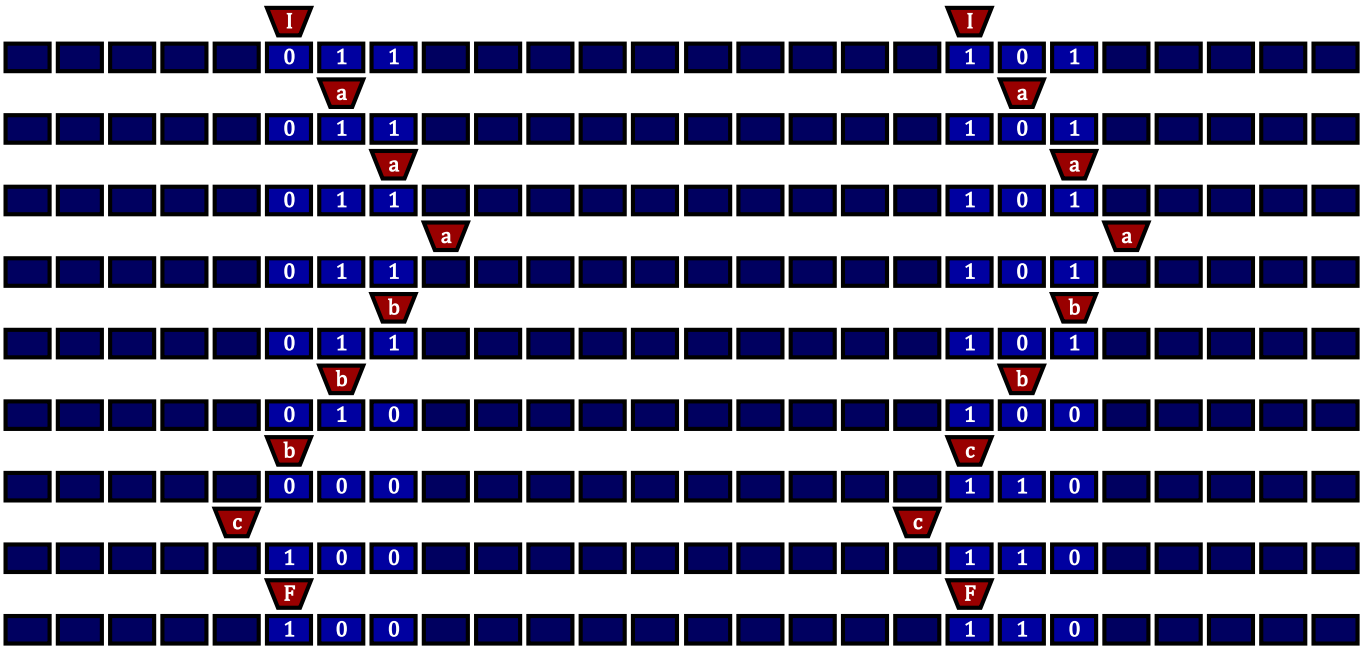
```
object WireWorldRule extends CA2[Char](ROI => ROI(1)(1) match {
  case 'W' if(ROI.flatten.count(_ == 'H') == 1) => 'H'
  case 'W' if(ROI.flatten.count(_ == 'H') == 2) => 'H'
  case 'W' => 'W'
  case 'H' => 'T'
  case 'T' => 'W'
  case 'B' => 'B'
})
```

2.3 チューリング機械

チューリング機械は、無限長の**テープ**と、その内容を読み書きする有限状態機械と、式 (2.3) の遷移関数 δ で構成される。状態 q_n で記号 x_n を読み取ると、記号 y_n に書き換える。状態 q_{n+1} に遷移して λ_n の方向に移動し、再び記号を読み取る。

$$(q_{n+1}, y_n, \lambda_n) = \delta(q_n, x_n), \quad \text{where} \begin{cases} q_n \in Q, \\ x_n, y_n \in \Sigma, \\ \lambda_n \in \{L, R\}. \end{cases} \tag{2.3}$$

この動作は、任意の逐次処理型の計算機と等価であり、並列処理型のセルオートマトンと並んで、計算機の頂点に立つ。特に、帰納的に枚挙可能な集合の計算が得意である。2 進数で与えられた自然数の後続を求める手順を、Fig. 2.3 に示す。



(a) 100=011+001.　　(b) 110=101+001.

Fig. 2.3: numerical increment operation on a Turing machine ($k = 1$).

任意の遷移関数を読み取り、その遷移関数を忠実に実行する、言語処理系と等価な**万能チューリング機械**も実装できる。遷移関数と計算手順で、別々のテープを使用した例を `UTM` 型に実装する。状態 F0 から F1 にかけ、遷移規則を検索する。

```
class UTM[V](data1: Seq[V], data2: Seq[V], bk1: V, bk2: V, mvL: V, mvR: V, var st1: V) {
  val tape1 = data1.zipWithIndex.map(_.swap).to(collection.mutable.SortedMap)
  val tape2 = data2.zipWithIndex.map(_.swap).to(collection.mutable.SortedMap)
  var (hd1, hd2) -> st2 = (0, 0) -> "F0"
  def rd1 = tape1.getOrElse(hd1, bk1)
  def rd2 = tape2.getOrElse(hd2, bk2)
  def apply(stop: V) = Iterator.continually(st2 match {
    case "F0" if rd2 == st1 => (st1 = st1, st2 = "F1", tape1(hd1) = rd1, hd1 += 0, hd2 += 1)
    case "F0" if rd2 != st1 => (st1 = st1, st2 = "F0", tape1(hd1) = rd1, hd1 += 0, hd2 += 5)
    case "F1" if rd2 == rd1 => (st1 = st1, st2 = "F2", tape1(hd1) = rd1, hd1 += 0, hd2 += 1)
    case "F1" if rd2 != rd1 => (st1 = st1, st2 = "F0", tape1(hd1) = rd1, hd1 += 0, hd2 += 4)
    case "F2" if rd2 != bk2 => (st1 = rd2, st2 = "F3", tape1(hd1) = rd1, hd1 += 0, hd2 += 1)
    case "F3" if rd2 != bk2 => (st1 = st1, st2 = "F4", tape1(hd1) = rd2, hd1 += 0, hd2 += 1)
    case "F4" if rd2 == bk1 => (st1 = st1, st2 = "F5", tape1(hd1) = rd1, hd1 += 0, hd2 += 1)
    case "F4" if rd2 == mvL => (st1 = st1, st2 = "F5", tape1(hd1) = rd1, hd1 -= 1, hd2 += 1)
    case "F4" if rd2 == mvR => (st1 = st1, st2 = "F5", tape1(hd1) = rd1, hd1 += 1, hd2 += 1)
    case "F5" if rd2 == bk2 => (st1 = st1, st2 = "F0", tape1(hd1) = rd1, hd1 += 0, hd2 += 1)
    case "F5" if rd2 != bk2 => (st1 = st1, st2 = "F5", tape1(hd1) = rd1, hd1 += 0, hd2 -= 1)
  }).takeWhile(t => st1 != stop || st2 != "F0").map(t => tape1.values.mkString)
}
```

状態 F2 から F4 で、状態遷移と書き戻しと移動を行う。状態 F5 でテープの左端に戻り、状態 F0 に戻る。使用例を示す。

```
case class CUTM(data1: String, data2: String) extends UTM(data1, data2, ' ', '*', 'L', 'R', 'I')
CUTM("0111111", "I0a0RI1a1Ra0a0Ra1a1Ra b Lb0c1Lb1b0Lb F1 c0c0Lc1c1Lc F R")('F').foreach(println)
```

遷移規則は式 (2.3) の通り、5 個組で読み込ませる。初期状態 I から状態 F まで動かすと、Fig. 2.3 の計算が実行される。

2.4 逆ポーランド記法

スタックを備え、再帰計算に対応した有限状態機械を**プッシュダウンオートマトン**と呼ぶ。式 (2.4) の遷移関数 δ に従う。Q は状態の、Σ と Γ は入力とスタックの記号の有限集合である。Γ^* は Γ の元を並べた任意長の記号列 y^* の集合である。

$$(q_{n+1}, y_n^*) = \delta(q_n, \sigma_n, x_n), \quad \text{where} \begin{cases} q_n \in Q, \\ x_n \in \Gamma, \\ y_n^* \in \Gamma^*, \\ \sigma_n \in \Sigma. \end{cases} \tag{2.4}$$

記号 σ_n を受け取ると、スタックの先頭の記号 x_n を取り除き、先頭に記号列 y_n^* を順番に積んで。状態 q_{n+1} に遷移する。再帰計算を活用した例として、第 2.1 節で実装した正規表現の拡張を考える。以下の関数 ZLO は、記号列 $\mathtt{Z}^n\mathtt{LO}^n$ を表す。

```
def ZLO: R[Char] = Cat(One('Z'), Cat(Alt(One('L'), new R(ZLO.test(_))), One('O')))
println(ZLO.test("ZZZZZZZZZZZZZZZZZZZZZZZZZLOOOOOOOOOOOOOOOOOOOOOOOOO").isDefined)
```

残念ながら、再帰計算は実行できても、受け取った記号列を読み返す機能がなく、計算能力はチューリング機械に劣る。ただし、記憶装置としてスタックを使う広義の**スタック機械**は、重要な計算モデルである。式 (2.5) の計算を例に考える。

$$(1 + 2) * (10 - 20). \tag{2.5}$$

演算子には優先順位があるため、式を左から読むだけでは、計算は困難である。数値を保持する記憶装置も必要である。前者は、式 (2.6) の**逆ポーランド記法**で解決する。演算子に優先順位はなく、出現する順番に、直前の数値に適用される。

```
1 2 + 10 20 - *.
```
(2.6)

手順を Fig. 2.4 に示す。逆ポーランド記法は、式の読み返しを伴う再帰計算や条件分岐を除き、任意の計算を実行できる。

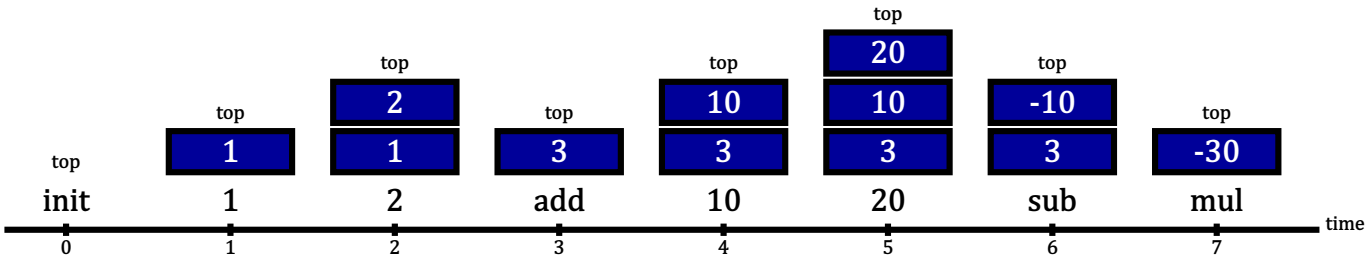


Fig. 2.4: 1 2 + 10 20 - *.

その再帰計算や条件分岐も、指定された長さだけ記号列を遡る**分岐命令**があれば実現できる。詳細は第 6 章に解説する。逆ポーランド記法の数式を計算する実装例を示す。数式は、空白で区切る必要がある。この実装は、第 4 章で再び使う。

```
object ArithStackMachine extends collection.mutable.Stack[Int]() {
  def apply(program: String): Int = program.split(" +").map {
    case "+" => push(((a: Int, b: Int) => b + a)(pop, pop))
    case "-" => push(((a: Int, b: Int) => b - a)(pop, pop))
    case "*" => push(((a: Int, b: Int) => b * a)(pop, pop))
    case "/" => push(((a: Int, b: Int) => b / a)(pop, pop))
    case num => this.push(num.toInt)
  }.lastOption.map(_ => pop).last
}
```

第3章　ラムダ計算の理論

実在する計算機を意識した第2章の計算モデルに対し、関数の**評価**と**適用**による計算手順の抽象化がラムダ計算である。第3章では、任意の式を**ラムダ式**と呼ぶ。変数も、整数も、関数もラムダ式である。関数は、式 (3.1) のように定義する。

$$f \coloneqq \lambda xy.2x + 3y + z + 1. \tag{3.1}$$

式 (3.1) を関数 f の**ラムダ抽象**と呼ぶ。変数 x と y を、λ により**束縛**された変数と呼ぶ。また、変数 z を**自由変数**と呼ぶ。式 (3.1) は式 (3.2) と等価である。関数 g は、変数 x を束縛し、変数 y を引数に取る関数を返す。これを**カリー化**と呼ぶ。

$$g \coloneqq \lambda x.\lambda y.2x + 3y + 1. \tag{3.2}$$

式 (3.3) は、変数 x と y を具体的な値で束縛する。これを関数適用と呼ぶ。また、式の実体を計算する操作を評価と呼ぶ。評価の途中で、束縛変数を定数に置換する操作を**ベータ簡約**と呼ぶ。式 (3.3) の値は、2度の簡約を経由して 27 と求まる。

$$\lambda x.\lambda y.(3x + 7y)\ \ 2\ \ 3 \underset{\beta}{\rightarrow} \lambda y.(6 + 7y)\ \ 3 \underset{\beta}{\rightarrow} 6 + 21 = 27. \tag{3.3}$$

任意の自然数と演算は、自然数を枚挙する関数 s と自然数 0 があれば、**ペアノの公理**で定義できる。式 (3.4) に例を示す。自然数は、2個の引数を取る関数で表す。変数 x に自然数を渡せば、加算になる。変数 s に自然数を渡せば、乗算になる。

$$n \coloneqq \lambda sx.(s^{\circ n}x)\ \ \lambda x.(x + 1)\ \ 0,\ \ \begin{cases} a + b \coloneqq \lambda ab.\lambda sx.as(bsx), \\ a \times b \coloneqq \lambda ab.\lambda sx.a(bs)x. \end{cases} \tag{3.4}$$

真偽値は、真と偽の順序組を引数に取り、どちらかを返す関数で表現できる。論理積と論理和の定義例を式 (3.5) に示す。真偽値の変数 y を偽で束縛すれば、変数 x との論理積になる。逆に、変数 x を真で束縛すれば、変数 y との論理和になる。

$$\mathrm{t} \coloneqq \lambda xy.x,\ \ \mathrm{f} \coloneqq \lambda xy.y,\ \ \begin{cases} a \wedge b \coloneqq \lambda ab.ab\mathrm{f}, \\ a \vee b \coloneqq \lambda ab.a\mathrm{t}b. \end{cases} \tag{3.5}$$

関数 f に対し、性質 $f(x) = x$ を満たす点 x を**不動点**と呼ぶ。また、式 (3.6) の性質を満たす関数 p を**不動点演算子**と呼ぶ。

$$\forall f,\ \ f(p(f)) \equiv p(f). \tag{3.6}$$

関数 p を利用すれば、再帰的な関数 $h(x)$ を式 (3.7) で定義できる。関数 h は、再帰計算の実体を表す関数 g を引数に取る。

$$h \coloneqq \lambda x.pgx,\ \ \text{where}\ \ g \coloneqq \lambda fy.E. \tag{3.7}$$

関数 h が再帰的であるには、関数 g の変数 f が関数 h を参照する必要がある。式 (3.8) の変形で、この要求は保証される。

$$h \equiv \lambda x.(pg)x \equiv \lambda x.(g(pg))x \equiv \lambda x.ghx. \tag{3.8}$$

式 (3.8) を**無名再帰**と呼ぶ。関数 p が実装できれば、任意の再帰計算を実行できる。最も著名な実装例を式 (3.9) に示す。

$$\mathbb{Y} \coloneqq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)). \tag{3.9}$$

関数 f に対し、式 (3.9) の関数 $\mathbb{Y}$ が式 (3.6) を満たす様子は、式 (3.10) で証明できる。ただし、無限再帰に注意を要する。関数 $\mathbb{Y}f$ を評価すると、同じ関数 $\mathbb{Y}f$ が右辺に出現する。無限再帰を防ぐには、関数 $\mathbb{Y}f$ の評価を遅延させる必要がある。

$$\mathbb{Y}f \underset{\beta}{\rightarrow} (\lambda x.f(xx))(\lambda x.f(xx)) \underset{\beta}{\rightarrow} f((\lambda x.f(xx))(\lambda x.f(xx))) \equiv f(\mathbb{Y}f). \tag{3.10}$$

または、式 (3.11) の関数 $\mathbb{Z}$ なら、関数 $\mathbb{Z}f$ の評価は停止する。関数 $\mathbb{Z}$ は、関数 $\mathbb{Y}$ に**イータ変換**の逆を施した関数である。

$$\mathrm{Z} \coloneqq \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)). \tag{3.11}$$

関数 $\mathbb{Z}f$ を評価すると、変数 y を引数に取る関数が出現する。右辺の関数 $\mathbb{Z}f$ の展開が保留され、無限再帰は回避される。

$$\mathrm{Z}f \underset{\beta}{\rightarrow} (\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)) \underset{\beta}{\rightarrow} f(\lambda y.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))y) \underset{\beta}{\rightarrow} f(\lambda y.\mathrm{Z}fy). \tag{3.12}$$

式 (3.9) の関数 $\mathbb{Z}$ を利用すれば、本書で自作する言語は、任意の計算を実行できる。第9章で実装を終えた後に実験する。

第4章　簡単なコンパイラ

第 2 章の計算モデルは、C 言語やラムダ計算など**高水準言語**の内容を実行するには原始的すぎる。そこで、翻訳を行う。翻訳を行う言語処理系を**コンパイラ**と呼ぶ。第 4 章では、簡単な逆ポーランド記法の翻訳を例に、その概念を解説する。

4.1　形式言語の階層性

形式言語とは、定義が明確で、何らかの計算手順で処理できる言語である。まず、形式言語 L は式 (4.1) で定義される。

$$L(G) \subset \Sigma^* = \{\langle \sigma_1, ..., \sigma_n, ... \rangle \mid \sigma_n \in \Sigma\}. \tag{4.1}$$

言語 L は**文**の集合である。文とは、記号 σ の列である。記号は有限集合 Σ で定義され、集合 Σ を**アルファベット**と呼ぶ。記号 σ の出現には、明確な規則がある。この規則を**生成規則**と呼び、生成規則の集合を**文法**と呼ぶ。式 (4.2) に例を示す。

$$P = \begin{cases} \texttt{S} \to \texttt{(S)}, \\ \texttt{S} \to \texttt{(f)}, \end{cases} : (N \cup \Sigma)^* \to (N \cup \Sigma)^*. \tag{4.2}$$

生成規則は、左辺の記号列を右辺の記号列に置換する規則である。式 (4.2) の例では、記号 S から式 (4.3) が導出される。

$$\texttt{(f)}, \texttt{((f))}, \texttt{(((f)))}, \texttt{((((f))))}, \texttt{(((((f)))))}, \texttt{((((((f))))))}, \texttt{(((((((f)))))))}, \texttt{((((((((f))))))))}, ... \tag{4.3}$$

生成規則の両辺に出現できる記号 $\nu \in N$ を**非終端記号**と呼ぶ。また、右辺に限って出現する記号 $\sigma \in \Sigma$ を**終端記号**と呼ぶ。任意の文は、**開始記号**と呼ばれる記号 S を起点に生成される。最終的に、言語 $L(G)$ の文法 G は式 (4.4) で定義される。

$$G = (N, \Sigma, P, S), \text{ where } S \in N. \tag{4.4}$$

文法 G に従う文を生成し、または文を開始記号 S に帰する手順が定義され、曖昧性がなければ、文法 G は**形式的**である。形式言語の中でも、生成規則が自由な言語を**帰納的可算言語**と呼び、式 (4.5) の制限を加えた言語を**文脈依存言語**と呼ぶ。

$$\alpha A \beta \to \alpha \gamma \beta, \text{ where } \begin{cases} A \in N, \\ \alpha, \beta \in (N \cup \Sigma)^*, \\ \gamma \in (N \cup \Sigma)^+. \end{cases} \tag{4.5}$$

形式言語の中でも、式 (4.6) の制限を持ち、前後の文脈に依存せずに、生成規則が適用できる言語を**文脈自由言語**と呼ぶ。第 2.4 節で述べたプッシュダウンオートマトンを利用して、文に対して生成規則を再帰的に適用することで処理できる。

$$A \to \alpha, \text{ where } \begin{cases} A \in N, \\ \alpha \in (N \cup \Sigma)^*. \end{cases} \tag{4.6}$$

形式言語の中でも、文法の制約が強く、有限状態機械で処理可能な言語を**正規言語**と呼ぶ。その記法が正規表現である。有限状態機械では、無限の記憶を持てず、特に再帰的な生成規則を扱えず、生成規則は式 (4.7) に示す形式に制限される。

$$\begin{cases} A \to a, \\ A \to aB, \end{cases} \text{ where } \begin{cases} a \in \Sigma, \\ A, B \in N. \end{cases} \tag{4.7}$$

形式言語の文は、適用した生成規則の木構造で表現できる。これを**構文木**と呼び、構文木を導く作業を**構文解析**と呼ぶ。特に LL 法では、終端記号の列を読み進め、見つけた終端記号に適う生成規則を、開始記号 S を起点に深さ優先探索する。

$$(S = \texttt{add}) \to (\texttt{mul + mul}) \to (\texttt{num * num + num}) \to (\texttt{1 * 2 + 3}). \tag{4.8}$$

LR 法では、終端記号の列を読み進め、置換可能な部分を非終端記号に置換する。最終的に開始記号 S に到達して終わる。

$$(\texttt{1 * 2 + 3}) \to (\texttt{num * num + num}) \to (\texttt{mul + mul}) \to (S = \texttt{add}). \tag{4.9}$$

通常、高水準言語は形式言語である。仮に自然言語を採用すると、翻訳する手順が曖昧になり、実装困難なためである。

4.2 解析表現文法の例

形式文法は、構文解析の際には曖昧になる。そこで、生成規則ではなく、構文解析の手順を形式的に定義した文法もある。**解析表現文法**はその例である。解析表現文法は、文脈依存言語の部分集合を扱う。簡単な四則演算を定義する例を示す。

加減算 `add ::= mul (('+' / '-') mul)*`
乗除算 `mul ::= num (('*' / '/') num)*`
整数値 `num ::= [0-9]+ / '(' add ')'`

左辺が非終端記号で、右辺が非終端記号と終端記号の列を表す。ただし、右辺に出現する記号には、以下の意味がある。

*	直前の記号が 0 回以上出現する。
+	直前の記号が 1 回以上出現する。
?	直前の記号が出現する場合がある。
/	直前または直後の記号が出現する。
()	括弧内の記号列をひと纏めにする。
''	引用内の字句がそのまま出現する。
[]	範囲内の記号が選択的に出現する。
&	直後の記号が出現すれば成功する。
!	直後の記号が出現すれば失敗する。

解析表現文法は LL 法の亜種である**再帰下降構文解析**を定義する記法である。これは、再帰的な関数で構文解析器を表す。様々な言語を定義可能だが、非終端記号を置換すると再び左に出現する**左再帰**の言語では、無限再帰に陥る欠点がある。

左再帰 `add ::= add ('+' / '-') mul / mul`

左再帰は、**左結合**の式を表す際に重要である。式 (4.10) に例を示す。左結合の式では、式の左側の演算子が優先される。

$$1-2-3-4-5=(((1-2)-3)-4)-5=-13. \tag{4.10}$$

右結合にすれば無限再帰を回避できるが、式の意味が変化してしまう。反復を表す特殊記号*など、代替手段で回避する。

4.3 構文解析器の実装

第 4.3 節では、第 2.1 節で解説した正規表現の処理系を改良して、解析表現文法に基づく**パーサコンビネータ**を実装する。以下の PEG 型を継承した**高階関数**を組み合わせ、再帰下降構文解析器を構築する。構文解析に成功すると、結果を返す。

```
class PEG[+M](f: String => Option[Out[M]]) {
  def skip = Reg("""\s*""".r) ~> this <~ Reg("""\s*""".r)
  def / [R >: M](q: => PEG[R]): PEG[R] = new Alt(this, q)
  def ~ [R](q: => PEG[R]): PEG[(M, R)] = new Cat(this, q)
  def <~[R](q: => PEG[R]) = this ~ q ^ (_._1)
  def ~>[R](q: => PEG[R]) = this ~ q ^ (_._2)
  def ^ [T](f: M => T) = new Map(this, f)
  def * = new Rep(this)
  def ? = new Opt(this)
  def apply(in: String) = f(in)
}
```

構文解析器は、その構文解析器で構築した構文木と、構文解析器が読み残した終端記号の列を返す。Out 型に実装する。

```
case class Out[+M](m: M, in: String) {
  def tuple[R](o: Out[R]) = Out(m -> o.m, o.in)
  def apply[R](p: PEG[R]) = p(in).map(tuple(_))
  def toSome = Out(Some(m), in)
}
```

最初に、最も単純な構文解析器を実装する。Str 型は、指定された終端記号列を左端に発見すると、その記号列を返す。Reg 型は、正規表現で指定された終端記号列を左端に発見すると、その記号列を返す。両者を総称して**字句解析器**と呼ぶ。

```
case class Str(p: String) extends PEG(s => Option.when(s.startsWith(p))(Out(p, s.substring(p.length))))
case class Reg(p: Regex) extends PEG(p.findPrefixMatchOf(_).map(m => Out(m.matched, m.after.toString)))
```

Alt 型は、1 個目の構文解析器が構文解析に成功すると、その結果を返し、失敗した場合は、2 個目の構文解析器を試す。Cat 型は、1 個目の構文解析器を試してから、読み残した終端記号列に 2 個目の構文解析器を試し、結果を結合して返す。

```
class Alt[L, R >: L](p: => PEG[L], q: => PEG[R]) extends PEG[R](s => p(s) orElse q(s))
class Cat[+L, +R](p: => PEG[L], q: => PEG[R]) extends PEG(p(_).map(_ apply q).flatten)
```

Map 型は、指定された構文解析器を試し、その結果に対して、関数を適用する。構文木に何らかの加工を施す際に使う。Opt 型は、省略可能な構文を表す。指定された構文解析器を試すが、構文解析に失敗した場合でも、成功したと見做す。

```
class Map[+S, +T](p: => PEG[S], f: S => T) extends PEG[T](p(_).map(t => Out(f(t.m), t.in)))
class Opt[+T](p: => PEG[T]) extends PEG(s => p(s).map(_.toSome).orElse(Some(Out(None, s))))
```

And 型と Not 型は、先読みを行う。指定された構文解析器を試すが、読み取り位置を進めず、構文解析の成否のみを返す。LL 法でも先読み可能だが、正規表現で切り出した字句の先読みに限定されるため、解析表現文法よりも表現能力が劣る。

```
class And[+T](p: => PEG[T]) extends PEG(s => if(p(s).isDefined) Some(Out(None, s)) else None)
class Not[+T](p: => PEG[T]) extends PEG(s => if(p(s).isDefined) None else Some(Out(None, s)))
```

Rep 型は、記号の反復を表す。読み取り位置を進めては構文解析を実行し、構文解析に失敗すると、結果を結合して返す。

```
class Rep[+T](p: => PEG[T]) extends PEG(s => {
  def ca(a: Out[T]): Out[Seq[T]] = Out(a.m +: re(a.in).m, re(a.in).in)
  def re(s: String): Out[Seq[T]] = p(s).map(ca).getOrElse(Out(Nil, s))
  Some(re(s))
})
```

次に、特殊な構文解析器を実装する。Fold 型は、左結合する中置記法の式を表す。無限再帰を回避する代替手段である。1 個目の構文解析器は、被演算子を表す。2 個目の構文解析器は演算子を表し、演算子の左右に並ぶ被演算子を結合する。

```
class Fold[T](p: => PEG[T], q: => PEG[(T, T) => T]) extends PEG({
  (p ~ (q ~ p).*)^(x => x._2.foldLeft(x._1)((l, r) => r._1(l, r._2)))
} apply(_))
```

最後に実装する Sep 型は、Rep 型の特殊な場合で、区切り文字で区切られた記号の反復を表す。区切り文字は廃棄される。

```
class Sep[T](p: => PEG[T], q: => PEG[_]) extends Fold[Seq[T]](p^(Seq(_)), q^(_ => _ ++ _))
```

最終的な構文解析器を、以下の PEGs 型を継承して実装すると、文字列や正規表現は、暗黙的に構文解析器に変換される。

```
class PEGs {
  implicit def implicitText(p: String): PEG[String] = new Str(p).skip
  implicit def implicitRegex(p: Regex): PEG[String] = new Reg(p).skip
}
```

使用例を示す。この実装は、数式を読み取る機能に加え、逆ポーランド記法の命令列を生成する**コード生成器**も兼ねる。

```
object ArithPEGs extends PEGs {
  def add: PEG[String] = new Fold(mul, ("+" / "-").^(op => (a, b) => s"$a $b $op"))
  def mul: PEG[String] = new Fold(num, ("*" / "/").^(op => (a, b) => s"$a $b $op"))
  def num: PEG[String] = "[0-9]+".r / ("(" ~> add <~ ")")
  def apply(e: String) = +ArithStackMachine(add(e).get.m)
}
```

第5章　自作言語の仕様書

本書で設計する fava は**動的型付け**を行う言語である。データ型は、整数型と実数型と論理型と文字列型と関数型がある。整数型は符号付き 32 bit 整数で、実数型は IEEE 754 (64 bit 2 進数) 浮動小数点数で、文字列は UTF-16 で表現される。

整数型　`int  ::= [0-9]+`
実数型　`real ::= [0-9]* ([0-9] '.' / '.' [0-9]) [0-9]*`
論理型　`bool ::= 'true' / 'false'`
文字列　`text ::= '"' [^"]* '"'`
関数型　`func ::= '(' (id (',' id)*)? ')' '=>' expr`

識別子は、その識別子が記述された箇所を包含し、識別子と同じ名前の引数を持つ、最も内側の関数の引数を参照する。関数の内外で変数を宣言または代入する機能はなく、従って、識別子が参照する実体は何らかの関数の引数に限られる。

識別子　`id ::= [@A-Z_a-z] [@0-9A-Z_a-z]*`

引数の値は変更不可能なため、識別子を含む式が参照する実体は、状態によらず自明である。これを**参照透明性**と呼ぶ。複数の文を逐次的に実行する**ブロック構文**はなく、関数宣言の構文も省略した。以下に解析表現文法による定義を示す。

ラムダ式　`expr ::= cond / or`
条件分岐　`cond ::= or  '?' expr ':' expr`
論理積　`or   ::= or  '|' and / and`
論理和　`and  ::= and '&' eql / eql`
等値比較　`eql  ::= eql ('==' / '!=') rel / rel`
順序比較　`rel  ::= rel ('<' / '>' / '<=' / '>=') add / add`
加減算　`add  ::= add ('+' / '-') mul / mul`
乗除算　`mul  ::= mul ('*' / '/' / '%') unr / unr`
単項演算　`unr  ::= ('+' / '-' / '!') unr / call`
関数適用　`call ::= call '(' expr (',' expr) ')' / fact`
式の要素　`fact ::= func / bool / text / real / int / id / '(' expr ')'`

この定義は左再帰を含む。中置記法の論理演算と比較演算と算術演算は、全て左結合である。単項演算は右結合である。関数は、他の関数を引数や返り値にできる。これを**高階関数**と呼ぶ。ただし、関数は宣言できず、従って**無名関数**である。

```
fava$ ((function)=>function())(()=>1+2)
3
```

高階関数の内部で定義された関数からは、外側の高階関数で定義された引数を参照できる。この関数を**関数閉包**と呼ぶ。関数の引数は、その引数を参照する関数閉包が存在し、参照される限り、関数適用が完了しても生存し、参照可能である。

```
fava$ ((x)=>((y)=>x*y))(2)(3)
6
```

引数の値は、関数を呼び出す時点では計算されず、値が必要になった時点で計算される。この動作を**非正格評価**と呼ぶ。**遅延評価**とも呼ばれる。詳細は第 3 章に述べるが、端的に言えば、再帰的な関数が無限再帰に陥るのを防ぐ効果がある。

```
fava$ ((f)=>((x)=>f(x(x)))((x)=>f(x(x))))((f)=>(n)=>(n==0)?1:n*f(n-1))(10)
3628800
```

第6章　分岐命令の仕組み

逆ポーランド記法の計算機で条件分岐を行うには、指定された長さだけ命令列を読み進め、または遡る分岐命令を使う。以下に例を示す。まず、Push 命令がスタックに値を積む。その値が偽なら、続く Skin 命令が、3 個の命令を読み飛ばす。

```
fava$ compile(true? 12: 34)
Push(true) Skin(3) Push(12) Skip(2) Push(34)
```

Skip 命令は**無条件分岐命令**で、条件式の値に依らず指定された個数の命令を読み飛ばす。この例の計算結果は 12 である。関数も同様の仕組みで実現できる。関数は、Def 命令に始まり、Ret 命令に終わる命令列に翻訳される。以下に例を示す。

```
fava$ compile((x,y)=>x+y)
Def(5,2) Load(0,0) Load(0,1) Add Ret
```

Def 命令は、指定された個数の引数を持つ関数を生成し、スタックに積む。また、関数の内容を読み飛ばす機能も備える。Ret 命令は、関数の引数を格納した**環境**を廃棄して、関数を呼び出した位置に復帰する。関数適用の命令列も以下に示す。

```
fava$ compile(((f)=>f())(()=>3))
Def(4,1) Load(0,0) Call(0) Ret Def(3,0) Push(3) Ret Call(1)
```

Call 命令は、引数と関数をスタックから取り出す。復帰位置を記録して、環境を構築してから、関数の冒頭に移動する。Fig. 6.1 に動作を示す。上下に 2 個のスタックがあるが、上のスタックで計算を行う。下のスタックは、環境を格納する。

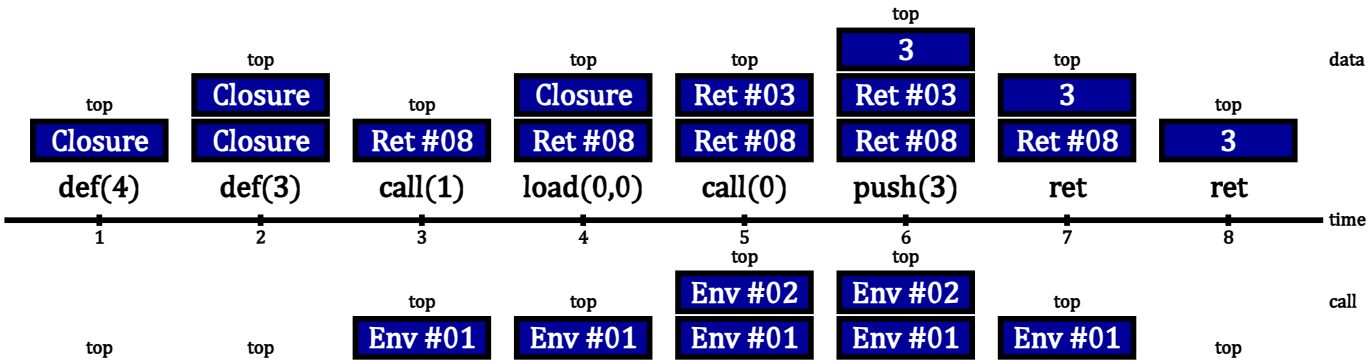


Fig. 6.1: function call mechanism for `((f)=>f())(()=>3)`.

以上は、引数の値を関数適用の前に求める**正格評価**の説明である。非正格評価では、引数の値を計算せずに関数を呼ぶ。条件分岐を含む関数では、引数の値が使用されず廃棄される場合がある。非正格評価であれば、この無駄を解消できる。

```
fava$ ((x,y,z)=>(x?y:z))(true,3+3,3*3)
6
```

引数を無名関数で包み、引数を参照する際にその関数を呼べば、非正格評価と同じ挙動になる。これを**名前呼び**と呼ぶ。

```
fava$ ((x,y)=>x()*x()+y())(()=>3+3,()=>3*3)
45
```

同じ引数を何度も参照する場合は、値を再利用すると効率的である。これを**必要呼び**と呼ぶ。詳細は第 8.3 節に述べる。

```
fava$ compile(((x)=>x)(5))
Def(11,1) Load(0,0) Nil Skin(6) Ref Call(0) Load(0,0) Fix Set Get Ret Def(3,0) Push(5) Ret Arg Call(1)
```

第7章　仮想計算機を作る

第7章では、第2.4節で実装した逆ポーランド記法の計算機を拡張し、第6章の分岐命令を備えた**仮想計算機**を実装する。FaVM型が仮想計算機の本体である。Fig. 6.1と同じ2個のスタックを備え、**プログラムカウンタ**が示す命令を実行する。

```
class FaVM(val codes: Seq[Code], var pc: Int = 0) {
  val call = new Stack[Env]
  val data = new Stack[Any]
  while(pc < codes.size) codes(pc)(this)
}
```

Stack型はスタックを実装する。指定された個数の値を取り出す機能や、指定された型で値を取り出す機能を実装する。

```
class Stack[E] extends collection.mutable.Stack[E] {
  def popN(n: Int) = Seq.fill(n)(pop).reverse
  def popAs[Type]: Type = pop.asInstanceOf[Type]
  def topAs[Type]: Type = top.asInstanceOf[Type]
  def env = (this :+ null).top.asInstanceOf[Env]
}
```

命令はCode型を継承する。仮想計算機の参照を受け取り、所定の操作を実行する。最後に命令を読み取る位置を進める。

```
class Code(op: FaVM => Unit) {
  def apply(vm: FaVM) = (op(vm), vm.pc += 1)
}
```

演算命令は、値をスタックから取り出して、計算結果をスタックに積む。**算術演算**と**論理演算**と**関係演算**に分類できる。演算命令はALU型を継承する。演算命令は**型検査**も行う。部分関数の使用は型検査が目的で、型次第では例外を起こす。

```
class ALU(n: Int, s: String, f: PartialFunction[Seq[Any], Any]) extends Code(vm => {
  def err(v: Seq[_]) = v.map(s => s"$s: ${s.getClass}").mkString("(", ", ", ")")
  vm.data.push(f.applyOrElse(vm.data.popN(n), (v: Seq[_]) => sys.error(err(v))))
})
```

分岐命令はJump型を継承する。引数に渡された関数が整数を返す場合は、命令を読み取る位置を整数の位置に変更する。

```
class Jump(op: FaVM => Option[Int]) extends Code(vm => op(vm).foreach(to => vm.pc = to - 1))
```

次に、関数の仕組みを実装する。関数は、関数の冒頭の位置と、引数の個数と、関数を生成した時点の環境を参照する。関数が参照する環境は、この関数を包み込む関数の引数を格納した環境である。関数閉包を実現するための布石である。

```
case class Closure(from: Int, narg: Int, out: Env)
```

遅延評価の仕組みも実装する。引数の式を包む関数と、計算結果を格納する。これを**プロミス**と呼ぶ。以下に実装する。

```
case class Promise(thunk: Closure, var cache: Any = null, var empty: Boolean = true)
```

最後に環境を実装する。環境は、関数適用の際に構築され、関数の引数を記憶する。遅延評価ではプロミスを管理する。関数閉包の機能を実現するため、環境は連鎖構造を持つ。環境は、関数の包含関係と連動し、**静的スコープ**を構成する。

```
case class Env(args: Seq[Any], out: Env = null) {
  def apply(nest: Int, index: Int): Any = if(nest > 0) out(nest - 1, index) else args(index)
}
```

第8章　命令セットを作る

第８章では、第７章に述べた仮想計算機に演算命令と分岐命令と遅延評価の命令を実装し、専用の**命令セット**を構築する。まず、最も基本的な Push 命令を実装する。引数に指定された**即値**をスタックに積む。定数式は Push 命令で実現できる。

```
case class Push(v: Any) extends Code(vm => vm.data.push(v))
```

機械語では、命令を**オペコード**と呼び、命令の引数を**オペランド**と呼ぶ。第 8.2 節の分岐命令にも、類似した引数がある。

8.1　演算命令の設計

演算命令には単項演算と２項演算がある。単項演算はスタックから値を１個だけ取り出し、計算結果をスタックに戻す。数値の符号を反転させる Neg 命令の実装例を示す。他には Pos 命令と論理演算の Not 命令もあるが、誌面では省略する。

```
case object Neg extends ALU(1, "-", {
  case Seq(v: I) => - v
  case Seq(v: D) => - v
})
```

演算命令は型検査も担当する。被演算子が部分関数に列挙するどの型とも異なる場合は、ALU 型の内部で例外が起こる。なお、実装の本筋とは無関係だが、型検査の条件分岐の式を綺麗に揃える目的で、基本型に１文字ずつ別名を設定した。

```
import java.lang.{String=>S}, scala.{Any=>A, Int=>I, Double=>D, Boolean=>B}
```

２項演算は２個の値を取り出し、計算結果をスタックに戻す。最初に、Add 命令を始め５種類の算術演算命令を実装する。減算や除算を実装する際は、被演算子の順序に注意を要する。演算子の左側の値が lhs に、右側の値が rhs に渡される。

```
case object Add extends ALU(2, "+", {
  case Seq(lhs: I, rhs: I) => lhs + rhs
  case Seq(lhs: I, rhs: D) => lhs + rhs
  case Seq(lhs: D, rhs: I) => lhs + rhs
  case Seq(lhs: D, rhs: D) => lhs + rhs
  case Seq(lhs: S, rhs: A) => s"$lhs$rhs"
  case Seq(lhs: A, rhs: S) => s"$lhs$rhs"
})
```

次に、関係演算命令を実装する。同値関係を調べる関係演算命令２種類と、順序関係を調べる関係演算命令４種類がある。

```
case object Gt extends ALU(2, ">", {
  case Seq(lhs: I, rhs: I) => lhs > rhs
  case Seq(lhs: I, rhs: D) => lhs > rhs
  case Seq(lhs: D, rhs: I) => lhs > rhs
  case Seq(lhs: D, rhs: D) => lhs > rhs
  case Seq(lhs: S, rhs: S) => lhs > rhs
})
```

最後に、２種類の論理演算命令を実装する。論理積と論理和である。整数値の場合は２進数の論理積と論理和を計算する。

```
case object Or extends ALU(2, "|", {
  case Seq(lhs: B, rhs: B) => lhs | rhs
  case Seq(lhs: I, rhs: I) => lhs | rhs
})
```

8.2 分岐命令の設計

分岐命令は第 6 章に準拠する。Skip 命令は、条件分岐で真の場合の処理を実行した後で、条件分岐を脱出する際に使う。Skin 命令は、値をスタックから取り出し、偽の場合は指定された個数の命令を読み飛ばす。条件分岐や遅延評価で使う。

```
case class Skip(plus: Int) extends Jump(vm => Some(vm.pc + plus))
case class Skin(plus: Int) extends Jump(vm => Option.when(!vm.data.popAs[B])(vm.pc + plus))
```

Def 命令は、指定された個数の引数を取る関数を生成する。関数の開始位置と環境を保存し、関数定義の直後に脱出する。

```
case class Def(size: Int, narg: Int) extends Jump(vm => Some {
  vm.data.push(Closure(vm.pc + 1, narg, vm.call.env))
  vm.pc + size
})
```

Ret 命令は、関数定義の最後に配置される。関数の環境を廃棄して、関数を呼ぶ直前の状態を復元し、以前の位置に戻る。

```
case object Ret extends Jump(vm => Some {
  vm.call.remove(0).asInstanceOf[Env]
  vm.data.remove(1).asInstanceOf[Int]
})
```

Call 命令は、指定された個数の引数で環境を構築する。関数から復帰する際の位置を控え、関数の開始位置に移動する。

```
case class Call(argc: Int) extends Jump(vm => Some {
  val args = vm.data.popN(argc)
  val func = vm.data.popAs[Closure]
  vm.call.push(Env(args, func.out))
  vm.data.push(vm.pc + 1)
  if(args.size == func.narg) func.from
  else sys.error(s"${func.narg} arguments required")
})
```

正格評価の場合は、第 8.2 節に実装した分岐命令と、第 8.3 節に実装する Load 命令があれば、任意の計算を実行できる。

8.3 遅延評価の設計

最初に、関数の引数を参照する Load 命令を実装する。現在の環境の連鎖構造を辿り、指定された番号の引数を取り出す。

```
case class Load(nest: Int, id: Int) extends Code(vm => vm.data.push(vm.call.env(nest, id)))
```

Arg 命令は、関数をスタックから取り出し、値が未定の引数とする。この命令は、関数適用の直前に実行する想定である。

```
case object Arg extends Code(vm => vm.data.push(Promise(vm.data.popAs[Closure])))
```

Get 命令は、引数をスタックから回収して、引数の値をスタックに積む。引数の値は、事前に計算済みである必要がある。引数の値が計算済みか確認するには、Nil 命令を使う。計算が必要な場合は、次に実装する Ref 命令を利用して計算する。

```
case object Get extends Code(vm => vm.data.push(vm.data.popAs[Promise].cache))
case object Nil extends Code(vm => vm.data.push(vm.data.topAs[Promise].empty))
case object Ref extends Code(vm => vm.data.push(vm.data.topAs[Promise].thunk))
```

Ref 命令は、引数の実体である関数を取り出す。Ref 命令を実行した直後に Call 命令を実行すれば、引数の値が求まる。Set 命令は、引数に値を設定する。Ref 命令と Call 命令で計算した結果を引数に設定すれば、非正格評価を実現できる。

```
case object Set extends Code(vm => vm.data.popAs[Promise].cache = vm.data.pop)
case object Fix extends Code(vm => vm.data.topAs[Promise].empty = false)
```

第9章　コンパイラを作る

第 9 章では、第 5 章の仕様に従って、式を第 8 章の命令列に翻訳する仕組みを作る。構文解析には第 4.3 節の実装を使う。最初に、第 9.1 節から第 9.2 節で、様々な構文木を実装する。構文木は、分割統治法によるコード生成器の役割を兼ねる。

```
trait AST {
  def code(implicit env: DefST): Seq[Code]
}
```

引数 env は、その構文木が表す式を包む最も内側の関数を表す。関数の引数を探す場合は、関数を外向きに辿って探す。

9.1　定数と演算の構文木

算術演算や関係演算や論理演算の式は、逆ポーランド記法の命令列に翻訳される。まず、定数を表す構文木を実装する。

```
case class LitST(value: Any) extends AST {
  def code(implicit env: DefST) = Seq(Push(value))
}
```

定数は単に Push 命令に翻訳される。なお、文字列の場合は、特別に StrST 型で扱う。特殊文字の処理を行う目的である。

```
case class StrST(string: String) extends AST {
  def code(implicit env: DefST) = LitST(StringContext.processEscapes(string)).code
}
```

次に、演算子の構文木を実装する。単項演算は UnST 型で表す。被演算子の命令列を生成し、直後に演算命令を追加する。

```
case class UnST(op: String, expr: AST) extends AST {
  def code(implicit env: DefST) = op match {
    case "+" => expr.code :+ Pos
    case "-" => expr.code :+ Neg
    case "!" => expr.code :+ Not
  }
}
```

2 項演算は BinST 型で表す。まず左側の被演算子の、次に右側の被演算子の命令列を生成し、直後に演算命令を追加する。

```
case class BinST(op: String, e1: AST, e2: AST) extends AST {
  def code(implicit env: DefST) = op match {
    case "+"  => e1.code ++ e2.code :+ Add
    case "-"  => e1.code ++ e2.code :+ Sub
    case "*"  => e1.code ++ e2.code :+ Mul
    case "/"  => e1.code ++ e2.code :+ Div
    case "%"  => e1.code ++ e2.code :+ Mod
    case "&"  => e1.code ++ e2.code :+ And
    case "|"  => e1.code ++ e2.code :+ Or
    case ">=" => e1.code ++ e2.code :+ Ge
    case "<=" => e1.code ++ e2.code :+ Le
    case ">"  => e1.code ++ e2.code :+ Gt
    case "<"  => e1.code ++ e2.code :+ Lt
    case "==" => e1.code ++ e2.code :+ Eq
    case "!=" => e1.code ++ e2.code :+ Ne
  }
}
```

条件分岐の式は IfST 型で表す。条件式と、真の場合に評価する式と、偽の場合に評価する式で、合計 3 個の引数を取る。

```
case class IfST(cond: AST, vals: (AST, AST)) extends AST {
  def code(implicit env: DefST) = {
    val code1 = vals._1.code
    val code2 = vals._2.code
    val jmp1 = Skin(2 + code1.size) +: code1
    val jmp2 = Skip(1 + code2.size) +: code2
    cond.code ++ jmp1 ++ jmp2
  }
}
```

条件分岐は、条件式の命令列の後に、Skin 命令と、真の場合の命令列と、Skip 命令と、偽の場合の命令列を出力する。

9.2 関数と引数の構文木

最初に、関数を表す DefST 型を実装する。引数と、関数の内容を引数に取る。また、外側の関数を参照する変数を持つ。関数の内容の命令列を生成してから、その長さと引数の個数に対応した Def 命令と、Ret 命令を冒頭と最後に追加する。

```
case class DefST(pars: Seq[String], body: AST, var out: DefST = null) extends AST {
  def code(implicit env: DefST) = {
    val codes = body.code((this.out = env, this)._2)
    (Def(codes.size + 2, pars.size) +: codes :+ Ret)
  }
}
```

関数は、定義された際の関数の包含関係を保持する。最も外側の関数は、便宜的に架空の関数を表す Root 型を参照する。

```
object Root extends DefST(Seq(), null)
```

次に、識別子の構文木を実装する。正格評価の場合は StIdST 型を使う。関数の包含構造を外向きに遡り、仮引数を探す。

```
case class StIdST(val name: String) extends AST {
  def search(env: DefST, nest: Int = 0): Load = {
    if(env.pars.contains(name)) return Load(nest, env.pars.indexOf(name))
    if(env.out != null) search(env.out, nest + 1)
    else sys.error(s"variable $name not defined")
  }
  def code(implicit env: DefST) = Seq(search(env))
}
```

非正格評価の場合は LzIdSt 型を使う。引数を取り出し、計算が必要なら計算し、引数の値を取り出す命令列を生成する。

```
case class LzIdST(val name: StIdST) extends AST {
  val (head, tail) = Seq(Nil, Skin(6), Ref, Call(0)) -> Seq(Fix, Set, Get)
  def code(implicit env: DefST) = (name.code ++ head ++ name.code ++ tail)
}
```

非正格評価の場合は、関数に渡す実引数を関数に包む必要がある。関数の命令列を生成し、直後に Arg 命令を配置する。

```
case class LzArgST(body: AST) extends AST {
  def code(implicit env: DefST) = DefST(Seq(), body).code :+ Arg
}
```

CallST 型は、関数適用を表す。まず、関数を参照する式の、次に引数の命令列を展開し、最後に Call 命令を配置する。

```
case class CallST(f: AST, args: Seq[AST]) extends AST {
  def code(implicit env: DefST) = f.code ++ args.map(_.code).flatten :+ Call(args.size)
}
```

9.3 再帰下降構文解析器

最後に、第 4.3 節で実装した解析表現文法の構文解析器を組み合わせ、再帰下降構文解析器を構築する。以下に実装する。第 5 章に掲載した文法の定義とほぼ同じ構造である。ただし、左結合の演算子は Fold 型を利用して、左再帰を回避した。

```
object FavaPEGs extends PEGs {
  def expr: PEG[AST] = (cond / or) <~ ("//" ~ ".*$".r).?
  def cond = (or <~ "?") ~ (expr ~ (":" ~> expr)) ^ IfST
  def or   = new Fold(and, "|" ^ (op => BinST(op, _, _)))
  def and  = new Fold(eql, "&" ^ (op => BinST(op, _, _)))
  def eql  = new Fold(rel, """(!|=)=""".r ^ (op => BinST(op, _, _)))
  def rel  = new Fold(add, """[<>]=?""".r ^ (op => BinST(op, _, _)))
  def add  = new Fold(mul, """[\+\-]""".r ^ (op => BinST(op, _, _)))
  def mul  = new Fold(unr, """[\*/%]""".r ^ (op => BinST(op, _, _)))
  def unr  = ("+" / "-" / "!").* ~ call ^ ((o,e) => o.foldRight(e)(UnST))
  def call = fact ~ args.* ^ ((f,a) => a.foldLeft(f)(CallST))
  def args = "(" ~> new Sep(expr ^ LzArgST, ",") <~")"
  def fact = func / bool / text / real / int / name / ("(" ~> expr <~ ")")
  def func = pars ~ ("=>" ~> expr) ^ ((p,e) => DefST(p, e))
  def pars = "(" ~> new Sep(name, ",") <~ ")" ^ (_.map(_.name.name))
  def bool = ("true" / "false") ^ (_.toBoolean) ^ LitST
  def text = ("\"" ~> """([^"\\]|\\[\\'"bfnrt])*""".r <~ "\"") ^ StrST
  def int  = """\d+""".r ^ (_.toInt) ^ LitST
  def real = """(\d+\.\d*|\d*\.\d+)""".r ^ (_.toDouble) ^ LitST
  def name = """[@A-Z_a-z][@0-9A-Z_a-z]*""".r ^ StIdST ^ LzIdST
}
```

使用例を以下に示す。構文解析を実行し、命令列に翻訳して第 7 章で実装した仮想計算機に渡すと、計算が実行される。

```
println(new FaVM(FavaPEGs.expr("((x,y)=>x+y)(2,3)").get.m.code(Root)).data.pop)
```

この構文解析器に、特殊な命令として compile を追加して、命令列を文字列で出力する機能を実装すれば、完成である。

9.4 言語処理系を動かす

完成した言語処理系は、第 3 章に述べたラムダ計算の実験環境として利用できる。まず、式 (3.4) の自然数の演算を試す。自然数は帰納的に枚挙可能で、自然数の後続の自然数を求める関数と 0 で表現できる。加算と乗算も、簡単に実装できる。

```
fava$ ((l,r)=>(f,x)=>l(f)(r(f)(x)))((f)=>(x)=>f(x),(f)=>(x)=>f(f(x)))((x)=>x+1,0) // 1 + 2
3
fava$ ((l,r)=>(f,x)=>l(r(f))(x))((f)=>(x)=>f(f(x)),(f)=>(x)=>f(f(x)))((x)=>x+1,0) // 2 * 2
4
```

次に、式 (3.5) の真偽値の演算を試す。真偽値は、真と偽の順序組で表現できる。論理積と論理和も、簡単に実装できる。

```
fava$ ((l,r)=>l(r,(x,y)=>y))((x,y)=>x,(x,y)=>y)(true,false) // true & false
false
fava$ ((l,r)=>l((x,y)=>x,r))((x,y)=>x,(x,y)=>y)(true,false) // true | false
true
```

無名関数による再帰計算も可能である。式 (3.9) に述べた不動点演算子を利用する。10 の階乗を求める例を以下に示す。

```
fava$ ((f)=>((x)=>f(x(x)))((x)=>f(x(x))))((f)=>(n)=>(n==0)?1:n*f(n-1))(10)
3628800
```

正格評価の言語では、無限再帰に陥る。代替手段として式 (3.11) の不動点演算子を利用すれば、再帰計算が可能になる。

```
fava$ ((f)=>((x)=>f((y)=>x(x)(y)))((x)=>f((y)=>x(x)(y))))((f)=>(n)=>(n==0)?1:n*f(n-1))(10)
3628800
```