Scalaで自作するプログラミング言語処理系

2023年11月13日

無線部開発班 JG1VPP

nextzlog.dev

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。自作する言語の名前は fava とする。最低限の機能に限定した簡素な言語だが、改良次第では高機能な言語に拡張できる。

1.1　サンプル

理念的には**純粋関数型言語**に分類できる。状態の概念がなく、式の値は定数だが、理論的には任意の計算を実行できる。関数は関数の引数や返り値にできる。ただし、関数の名前も局所変数も定義できず、参照可能な変数は引数に限られる。

```
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章で述べる通り、任意の計算を実行できる。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
```

自然数を関数で定義する例を示す。この例題は、関数型言語が任意の計算を実行できる性質を示す際に、よく登場する。

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

任意の順序組も定義できる。端的に言えば構造体に相当し、複雑なグラフ構造も関数で表現できる性質を示す例である。

```
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
```

1.2　開発環境

fava の実装は公開済みで、以下の操作でビルドできる。実行環境として Java が、開発環境として Gradle が必要である。

```
$ git clone https://github.com/autodyne/fava
$ gradle build -p fava
```

以下の操作で起動する。fava 以外にも、第2章で実装する様々な計算モデルを実験する機能も含まれ、起動時に選べる。fava は対話的に実行できる。fava の内部で実行される命令列を表示する機能も用意した。内部的な動作の理解に役立つ。

```
$ java -jar build/libs/fava.jar --fava
fava$ "HELLO, WORLD!"
HELLO, WORLD!
fava$ compile(1 + 2)
Push(1) Push(2) IAdd
fava$ exit
$ java -jar build/libs/fava.jar --lisp
lisp$ (+ 1 2 3)
6
lisp$ (exit)
```

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。さて、2 次元のセルオートマトンの実装例を、以下に示す。引数は、遷移規則と、縦横に並んだセルの最初の状態である。

```
class Grid[S](rule: Rule[S], data: Array[Array[S]]) {
  def update = {
    val next = rule(data.map(_.toSeq).toSeq)
    next.zip(data).foreach(_.copyToArray(_))
  }
}
```

全てのセルが同時に状態遷移する**同期型セルオートマトン**を以下に実装する。引数は、遷移規則と、近傍の距離である。

```
class Rule[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. 2.1 に例を示す。

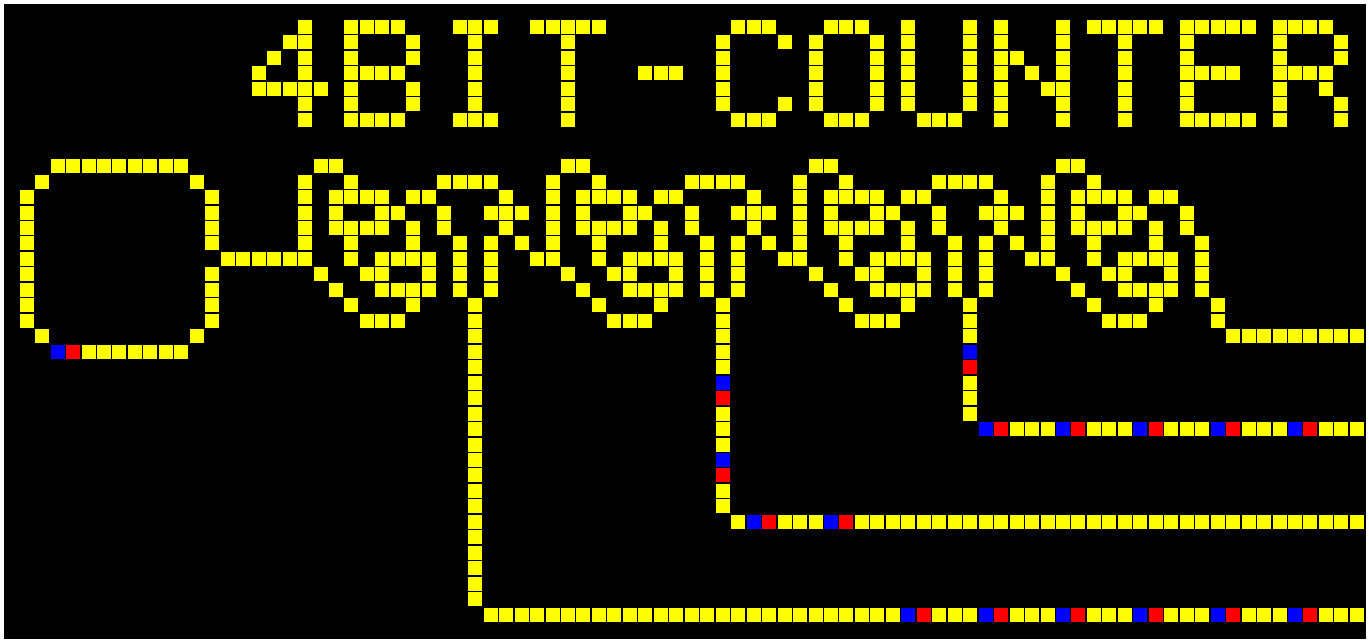


Fig. 2.1: Wireworld logic circuits.

Fig. 2.1 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る度に、右側の 4 本の配線を流れる信号が切り替わる。

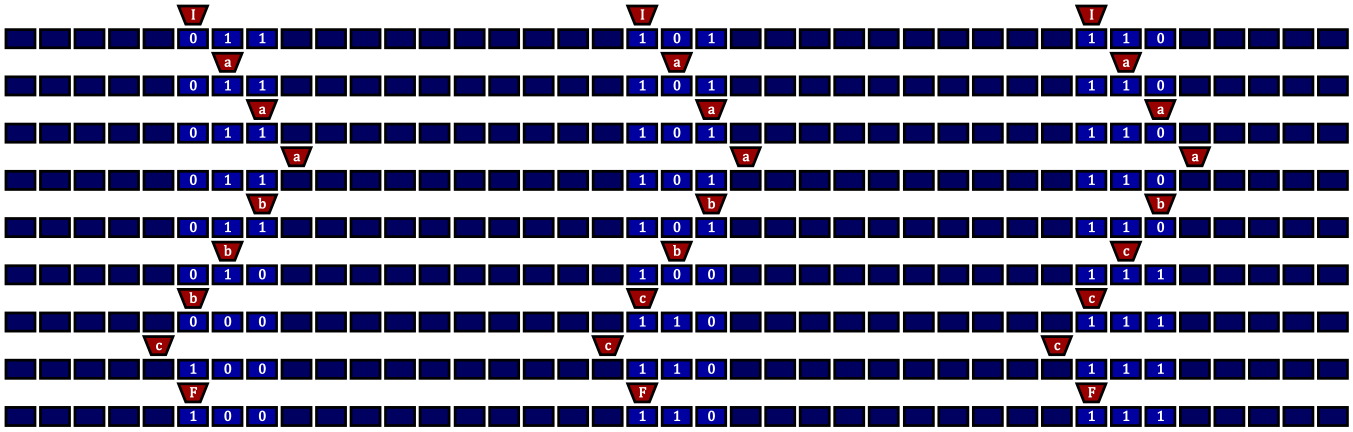
```
object WireWorldRule extends Rule[Char](ROI => (ROI(1)(1), ROI.flatten.count(_ == 'H')) match {
  case ('W', 1) => 'H'
  case ('W', 2) => 'H'
  case ('W', _) => 'W'
  case ('H', _) => 'T'
  case ('T', _) => 'W'
  case ('B', _) => 'B'
})
```

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 (2.2) 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。

$$(q_{n+1}, y_n, \lambda_n) = \delta(q_n, x_n), \quad \text{where} \quad \text{where} \begin{cases} q_n \in Q, \\ x_n, y_n \in \Sigma, \\ \lambda_n \in \{L, R\}. \end{cases} \tag{2.2}$$

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. 2.2 に示す。



(a) 100=011+001. (b) 110=101+001. (c) 111=110+001.

Fig. 2.2: 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、異なるテープを使用した例を `UTM` 型に実装する。状態 0 から 1 にかけて、遷移規則を検索する。

```
class UTM[V](data1: Seq[V], data2: Seq[V], b1: V, b2: V, mL: V, mR: V, var s1: V, var s2: Int = 0) {
  val tape1 = data1.zipWithIndex.map(_.swap).to(collection.mutable.SortedMap)
  val tape2 = data2.zipWithIndex.map(_.swap).to(collection.mutable.SortedMap)
  var hd1, hd2 = 0
  def r1 = tape1.getOrElse(hd1, b1)
  def r2 = tape2.getOrElse(hd2, b2)
  def apply(sop: V) = Iterator.continually(s2 match {
    case 0 if r2 == s1 => (s1 = s1, s2 = 1, tape1(hd1) = r1, hd1 += 0, hd2 += 1)
    case 0 if r2 != s1 => (s1 = s1, s2 = 0, tape1(hd1) = r1, hd1 += 0, hd2 += 5)
    case 1 if r2 == r1 => (s1 = s1, s2 = 2, tape1(hd1) = r1, hd1 += 0, hd2 += 1)
    case 1 if r2 != r1 => (s1 = s1, s2 = 0, tape1(hd1) = r1, hd1 += 0, hd2 += 4)
    case 2 if r2 != b2 => (s1 = r2, s2 = 3, tape1(hd1) = r1, hd1 += 0, hd2 += 1)
    case 3 if r2 != b2 => (s1 = s1, s2 = 4, tape1(hd1) = r2, hd1 += 0, hd2 += 1)
    case 4 if r2 == b1 => (s1 = s1, s2 = 5, tape1(hd1) = r1, hd1 += 0, hd2 += 1)
    case 4 if r2 == mL => (s1 = s1, s2 = 5, tape1(hd1) = r1, hd1 -= 1, hd2 += 1)
    case 4 if r2 == mR => (s1 = s1, s2 = 5, tape1(hd1) = r1, hd1 += 1, hd2 += 1)
    case 5 if r2 == b2 => (s1 = s1, s2 = 0, tape1(hd1) = r1, hd1 += 0, hd2 += 1)
    case 5 if r2 != b2 => (s1 = s1, s2 = 5, tape1(hd1) = r1, hd1 += 0, hd2 -= 1)
  }).takeWhile(t => s1 != sop || s2 != 0).map(t => tape1.values.mkString)
}
```

状態 2 から 4 にかけて、状態遷移と書き戻しと移動を行う。状態 5 でテープの左端に戻り、状態 0 に戻る。使用例を示す。遷移規則は式 (2.2) 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. 2.2 の計算が実行される。

```
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)
```

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 (2.3) 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。

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

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 (2.4) の計算を例に考える。

$$(1 + 2) * (10 - 20). \tag{2.4}$$

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 (2.5) 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 - *.
```
(2.5)

手順を Fig. 2.3 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。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。

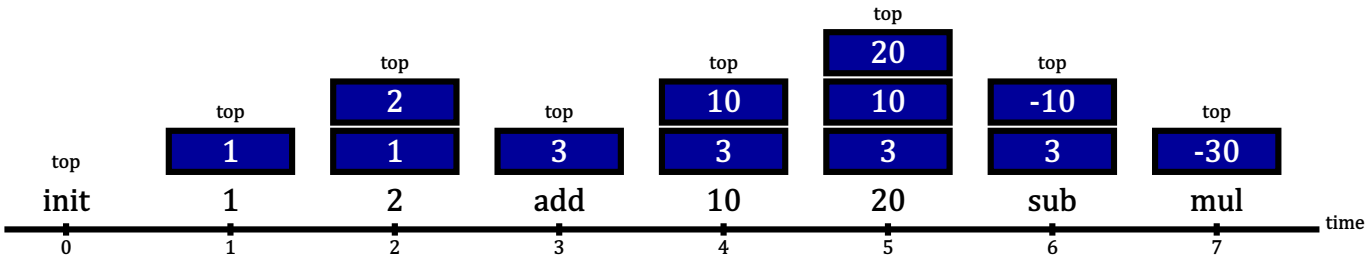


Fig. 2.3: `1 2 + 10 20 - *`.

逆ポーランド記法とスタック機械による四則演算の実装例を以下に示す。数式は、整数と演算子を空白で区切って渡す。

```
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
}
```

Fig. 2.3 の手順で計算を行う。整数を読み取るとスタックに積み、演算子を読み取ると演算を行う。以下に使用例を示す。

```
println(ArithStackMachine("1 2 + 10 20 - *")) // -30
println(ArithStackMachine("3 4 * 10 20 * +")) // 212
```

この実装は、第 4 章で再び使用する。第 4 章では、中置記法の数式を逆ポーランド記法に変換するコンパイラを実装する。

第3章　型付きラムダ計算

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。

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

3.1　万能性

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。

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

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。

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

再帰計算は、任意の関数 f の**不動点**を求める関数 p が存在し、即ち式 (3.6) を満たす場合に、無名関数の形で表現できる。

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

例えば、関数 g を任意の再帰計算とし、式 (3.7) に示す関数 h を定義すると、関数 g は関数 h と変数 x を引数に受け取る。

$$h \coloneqq \lambda x.pgx \equiv \lambda x.(pg)x \equiv \lambda x.(g(pg))x \equiv \lambda x.ghx,\ \ \text{where}\ \ g \coloneqq \lambda fy.E. \tag{3.7}$$

関数 h を通じて、関数 g が再帰的に参照される。任意の関数の不動点を与える関数 p の、最も著名な例を式 (3.8) に示す。

$$\mathbb{Y} \coloneqq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)). \tag{3.8}$$

関数 f に対し、式 (3.8) の関数 $\mathbb{Y}$ が式 (3.6) を満たす様子は、式 (3.9) で証明できる。ただし、無限再帰に注意を要する。例えば、式 $\mathbb{Y}fx$ を評価すると、無限に式 (3.9) が展開される。対策として、第5章で解説する非正格評価が必要になる。

$$\mathbb{Y}f \underset{\beta}{\rightarrow} (\lambda x.f(xx))(\lambda x.f(xx)) \underset{\beta}{\rightarrow} f((\lambda x.f(xx))(\lambda x.f(xx))) \equiv f(\mathbb{Y}f). \tag{3.9}$$

関数 $\mathbb{Y}$ と等価な関数 $\mathbb{Z}$ を利用する方法もある。式 (3.10) に示す関数 $\mathbb{Z}$ は、関数 $\mathbb{Y}$ に**イータ変換**の逆を施した関数である。

$$\mathbb{Z} \coloneqq \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)). \tag{3.10}$$

式 (3.11) を評価すると、右辺の関数が出現し、実際に引数 y を渡すまで式 (3.11) の展開が保留され、無限再帰を防げる。

$$\mathbb{Z}f \underset{\beta}{\rightarrow} (\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)) \underset{\beta}{\rightarrow} f(\lambda y.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))y) \underset{\beta}{\rightarrow} f(\lambda y.\mathbb{Z}fy). \tag{3.11}$$

以上で、算術や論理計算や再帰計算を含む、理論的な裏付けが揃った。第7章で言語処理系が完成したら、実験しよう。

3.2 型推論

型付きラムダ計算は、命題論理の規則に従って、式の型を推論し、型の矛盾や無限再帰を形式的に検出する体系である。簡単に命題論理を復習する。命題 P が命題 Q を、命題 Q が命題 R を含意する場合に、式 (3.12) の**三段論法**が成立する。

$$\frac{P \to Q \quad Q \to R}{P \to R}. \tag{3.12}$$

命題論理では、命題の妥当性は、演繹の累積で証明される。この過程は、命題の集合 Γ を仮定して、式 (3.13) で表せる。

$$\Gamma \vdash P \to R, \text{ where } \begin{cases} P \to Q \in \Gamma, \\ Q \to R \in \Gamma. \end{cases} \tag{3.13}$$

型付きラムダ計算では、集合 Γ を**型環境**と呼ぶ。具体的には、変数や部分式に設定した型の情報を格納した配列である。曖昧な型は変数で表す。例えば、自由変数 x の型は未知なので、変数 σ が環境 Γ に格納され、式 x の型は σ と推論される。

$$\frac{\Gamma(x) \coloneqq \sigma}{\Gamma \vdash x : \sigma}. \tag{3.14}$$

関数 f の型を推論しよう。過程を式 (3.15) に示す。関数は、含意の記号 $\to$ を利用して、定義域と値域の組で表現できる。

$$\frac{x : \sigma \vdash E : \tau \quad f \coloneqq \lambda x.E}{\Gamma \vdash f : \sigma \to \tau} \tag{3.15}$$

式 (3.15) を含意 $\to$ の**導入規則**と呼ぶ。最後に、関数 f の適用 fx の型を推論する。式 (3.16) を含意 $\to$ の**除去規則**と呼ぶ。

$$\frac{\Gamma \vdash f : \sigma \to \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash fx : \tau} \tag{3.16}$$

型推論の過程では、型変数が満たす制約条件の組が生成され、その全てを満たす型が解となる。式 (3.17) の例で考える。

$$(\lambda x.xy)(zy). \tag{3.17}$$

推論の過程を式 (3.18) に示す。同じ変数には、同じ型変数を設定する。推論の過程で、型変数の制約条件が生成される。

$$\frac{\dfrac{\dfrac{x : \alpha \quad y : \beta}{xy : \mu \mid \alpha = \beta \to \mu}}{\lambda x.xy : \alpha \to \mu \mid \alpha = \beta \to \mu} \quad \dfrac{z : \gamma \quad y : \beta}{zy : \nu \mid \gamma = \beta \to \nu}}{(\lambda x.xy)(zy) : \sigma \mid \alpha \to \mu = \nu \to \sigma \mid \alpha = \beta \to \mu \mid \gamma = \beta \to \nu} \tag{3.18}$$

代数学の要領で制約条件を消去し、解を得る作業を**単一化**と呼ぶ。特に、関数の型を分解する。式 (3.19) に過程を示す。

$$\frac{\dfrac{\dfrac{\alpha \to \mu = \nu \to \sigma}{\mu = \sigma} \quad \alpha = \beta \to \mu}{\alpha = \beta \to \sigma} \quad \dfrac{\dfrac{\alpha \to \mu = \nu \to \sigma}{\alpha = \nu} \quad \gamma = \beta \to \nu}{\gamma = \beta \to \alpha}}{\gamma = \beta \to \beta \to \sigma} \tag{3.19}$$

式 (3.19) の例では、全ての制約条件を消去できた。それでも、型変数 β, σ は任意の型になり得る。これを**多相型**と呼ぶ。なお、再帰関数の型推論では、式 (3.20) に示す**同値再帰型**が出現する。無闇に式 (3.20) を展開すると、無限再帰に陥る。

$$\sigma = \sigma \to \tau. \tag{3.20}$$

式 (3.9) を例に考える。式 (3.21) に示す推論により、型 ϕ, ψ は再帰型と判明する。その時点で推論を終える必要がある。

$$\frac{\dfrac{\dfrac{\dfrac{x : \phi}{xx : \mu \mid \phi = \phi \to \mu} \quad f : \eta}{f(xx) : \rho \mid \eta = \mu \to \rho \mid \phi = \phi \to \mu}}{\lambda x.f(xx) : \phi \to \rho \mid \eta = \mu \to \rho \mid \phi = \phi \to \mu} \quad \dfrac{\dfrac{\dfrac{x : \psi}{xx : \nu \mid \psi = \psi \to \nu} \quad f : \eta}{f(xx) : \tau \mid \eta = \nu \to \tau \mid \psi = \psi \to \nu}}{\lambda x.f(xx) : \psi \to \tau \mid \eta = \nu \to \tau \mid \psi = \psi \to \nu}}{(\lambda x.f(xx))(\lambda x.f(xx)) : \sigma \mid \phi \to \rho = (\psi \to \tau) \to \sigma \mid \eta = \mu \to \rho \mid \phi = \phi \to \mu \mid \eta = \nu \to \tau \mid \psi = \psi \to \nu} \tag{3.21}$$

同値再帰型を表す特殊な型変数を実装すれば、再帰関数の型推論も可能だが、誤った式に意図せず型が付く場合もある。

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.? ^ (_.toSeq), 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)
}
```

4.4 簡単な言語処理系

第 4.3 節の構文解析器で LISP を実装しよう。LISP は、簡素な文法ながら拡張性が高く、実用的な動的型付け言語である。LISP の式を S 式と呼ぶ。変数の名前や数値を表す**アトム**と、構文木を形成する**リスト**で構成される。文法も簡潔である。

```
object LispPEGs extends PEGs {
  def sexp: PEG[S] = list / quot / real / name
  def list = "(" ~> (sexp.* ^ List) <~ ")"
  def real = "[0-9]+".r ^ (real => Real(BigDecimal(real)))
  def name = """[^'`,@\(\)\s]+""".r ^ (name => Name(name))
  def quot = "'" ~> sexp ^ (Seq(Name("quote"), _)) ^ List
}
```

S 式の実装を以下に示す。引数の exp は式の文字列である。eval は評価器で、後述する環境を参照して、式の値を返す。

```
abstract class S(val exp: String, eval: S => Env => S) {
  override def toString = exp
  def apply(env: Env): S = eval(this)(env)
  def apply(env: Env)(args: Seq[S]): S = apply(env).asInstanceOf[Form].app(args, env)
}
```

次にアトムを実装する。変数を表す Name 型と、実数値の Real 型を以下に示す。他にも論理型や文字列型を実装しよう。

```
case class Name(name: String) extends S(name, v => env => env.apply(v))
case class Real(real: BigDecimal) extends S(real.toString, v => _ => v)
```

次にリストを実装する。LISP のリストは関数適用として評価される。最初の要素が関数で、残りが引数のリストとなる。

```
case class List(list: Seq[S]) extends S(list.mkString("(", " ", ")"), _ => list.head(_)(list.tail))
```

次に演算子を実装する。Form 型を継承して、関数の定義を表す Lambda 型と、**マクロ**の定義を表す Syntax 型を実装する。関数の場合は、まず引数を評価して、引数を記憶した環境を生成し、最後に関数の値を評価する。これを**正格評価**と呼ぶ。

```
class Form(exp: String, val app: (Seq[S], Env) => S) extends S(exp, v => _ => v)
class Lambda(p: List, v: S, e: Env) extends Form(s"(lambda $p $v)", (a, s) => v(Env(Some(e), p, s(a))))
class Syntax(p: List, v: S, e: Env) extends Form(s"(syntax $p $v)", (a, s) => v(Env(Some(e), p, a))(s))
```

対照的にマクロの場合は、引数を評価せず、式のままマクロの内部に展開し、そのマクロを通常の式と同様に評価する。例えば、C 言語の制御構文に相当するマクロも定義できる。次に、環境を実装する。環境は変数の名前と値を記憶する。

```
case class Env(out: Option[Env], params: List, args: Seq[S]) {
  val map = params.list.zip(args).to(collection.mutable.Map)
  def apply(name: S): S = {
    if(map.isDefinedAt(name)) map(name)
    else if(out.nonEmpty) out.get(name)
    else sys.error(s"$name undeclared")
  }
  def apply(args: Seq[S]): Seq[S] = args.map(_.apply(this))
}
```

最後に、お好みで Form 型を継承し、組込み関数や構文を充実させよう。関数とマクロを定義する構文の例を以下に示す。

```
object LambdaForm extends Form("lambda", (a, s) => new Lambda(a.head.asInstanceOf[List], a(1), s))
object SyntaxForm extends Form("syntax", (a, s) => new Syntax(a.head.asInstanceOf[List], a(1), s))
```

以上で、簡素ながら優れた拡張性と実用性を備える LISP が完成した。階乗を defun マクロで定義する例を以下に示す。

```
lisp$ (defun fact (x) (if (eq x 1) x (* x (fact (- x 1)))))
(lambda (x) (if (eq x 1) x (* x (fact (- x 1)))))
```

第5章　自作言語の仕様書

fava は静的型付け言語である。組込み型には、整数型と実数型と論理型と文字列型と関数型があり、型推論が行われる。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　命令セットを作る

第 6 章では、第 2.4 節で実装した計算機を拡張し、**分岐命令**を備えた**命令セット**を設計して、条件分岐や関数を実現する。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)=>0+x+y)
Def(7) Push(0) Load(0,0) IAdd Load(0,1) IAdd Ret
```

Def 命令は、それに続く命令列を実行する関数をスタックに積む。また、その命令列を読み飛ばし、意図せぬ実行を防ぐ。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) Load(0,0) Call(0) Ret Def(3) 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。

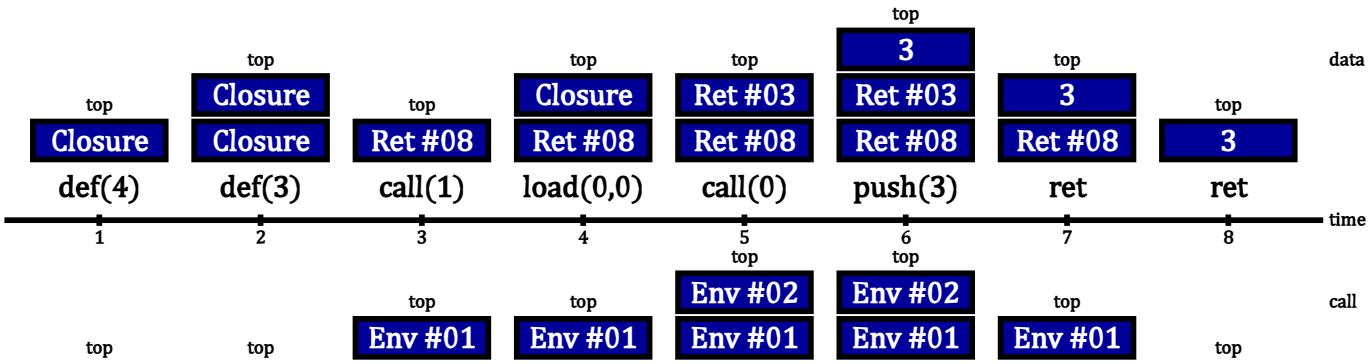


Fig. 6.1: function call mechanism for `((f)=>f())(()=>3)`.

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 6.6 節に述べる。

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

6.1 命令の基本設計

命令は、Code 型を継承する。FaVM 型は、実行環境の本体である。命令は、所定の操作を実行し、変数 pc を繰り上げる。

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

変数 pc は、**プログラムカウンタ**に相当する。これは、命令を実行する度に繰り上がり、次に実行する命令の位置を示す。

6.2 関数の基本設計

次に、関数の仕組みを実装する。関数は、関数が実行する最初の命令の位置と、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, out: Env)
```

遅延評価の仕組みも実装する。遅延評価は、関数の引数を包む関数と、その計算結果を格納する記憶領域で構成される。引数が計算済みの場合は、その値を使用し、何度も計算を繰り返す無駄を省く。計算を行う前の状態を**プロミス**と呼ぶ。

```
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(local: Seq[Any], out: Env = null) {
  def apply(depth: Int, index: Int): Any = depth match {
    case 0 => this.local(index)
    case d => out(d - 1, index)
  }
}
```

関数の引数は、Load 命令で取得する。現在の環境を起点に、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)))
```

以上で、関数や遅延評価の仕組みを整備した。実際に関数や遅延評価を実現する命令は、第 6.5 節や第 6.6 節で設計する。

6.3 実行環境の設計

以下の FaVM 型が実行環境である。Fig. 6.1 と同様に、2 個のスタックを備え、引数で渡された命令列を順番に実行する。

```
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]
}
```

6.4 演算命令の設計

第6.4節では、第2.4節で解説した逆ポーランド記法を参考に、四則演算や論理演算を含む各種の演算命令を充実させる。まず、Push命令を実装する。引数に指定された**即値**をスタックに積む命令である。特に、定数に相当する命令と言える。

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

次に、複数の引数をスタックから取り出して、計算結果をスタックに戻す演算命令は、以下に示すArity型を継承する。ただし、引数の型を指定する必要がある場合は、Typed型を継承する。引数のopは、この命令に対応する演算子である。

```
class Arity(n: Int, f: Function[Seq[Any], Any]) extends Code(vm => vm.data.push(f(vm.data.popN(n))))
class Typed(val n: Int, val op: String, val t: Type, f: Function[Seq[Any], Any]) extends Arity(n, f)
```

Typed型を継承した命令は、以下のOp型で管理する。これは、演算子と命令を紐付け、検索を容易にする仕組みである。

```
class Op(op: Typed*)(val table: Map[(String, Type), Typed] = op.map(op => (op.op, op.t) -> op).toMap)
```

例えば、加減算や乗除算の命令をOp型にまとめ、管理する。演算子を指定すれば、対応する命令がtableから得られる。なお、第7章で実装するコンパイラは、この機能を利用して、加減算や乗除算の構文木を、対応する演算命令に変換する。

```
object AddOp extends Op(IAdd, DAdd, SAdd, ISub, DSub, SSub)()
object MulOp extends Op(IMul, DMul, IDiv, DDiv, IMod, DMod)()
object RelOp extends Op(IGt, DGt, ILt, DLt, IGe, DGe, ILe, DLe)()
object EqlOp extends Op(IEq, DEq, SEq, BEq, INe, DNe, SNe, BNe)()
object LogOp extends Op(IAnd, BAnd, IOr, BOr)()
```

さて、演算命令には、単項演算と2項演算がある。単項演算は被演算子が1個の命令で、符号の反転や論理の否定を行う。以下に、数値の符号を反転させるNeg命令の例を示す。Typed型を継承し、被演算子の型に応じて、何通りか実装する。

```
case object INeg extends Typed(1, "-", It, -_.head.asInstanceOf[I])
case object DNeg extends Typed(1, "-", Dt, -_.head.asInstanceOf[D])
```

なお、第6.4節では、基本型に1文字ずつの別名を設定した。例えば、整数型はIで、これは演算命令の接頭辞でもある。

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

さて、2項演算は、被演算子が2個の命令で、殆どの演算命令が該当する。以下に、加算を表すAdd命令の実装例を示す。

```
case object IAdd extends Typed(2, "+", It, v => v.head.asInstanceOf[I] + v.last.asInstanceOf[I])
case object DAdd extends Typed(2, "+", Dt, v => v.head.asInstanceOf[D] + v.last.asInstanceOf[D])
case object SAdd extends Typed(2, "+", St, v => v.head.asInstanceOf[S] + v.last.asInstanceOf[S])
```

以下に、除算を表すDiv命令の例を示す。なお、減算や除算を実装する際は、被演算子を取り出す順番に注意を要する。

```
case object IDiv extends Typed(2, "/", It, v => v.head.asInstanceOf[I] / v.last.asInstanceOf[I])
case object DDiv extends Typed(2, "/", Dt, v => v.head.asInstanceOf[D] / v.last.asInstanceOf[D])
```

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 IGt extends Typed(2, ">", It, v => v.head.asInstanceOf[I] > v.last.asInstanceOf[I])
case object DGt extends Typed(2, ">", Dt, v => v.head.asInstanceOf[D] > v.last.asInstanceOf[D])
```

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。以上で、全ての算術演算と関係演算と論理演算の命令が揃った。殆どの命令は、誌面の都合で省略したが、適切に補おう。

```
case object IAnd extends Typed(2, "&", It, v => v.head.asInstanceOf[I] & v.last.asInstanceOf[I])
case object BAnd extends Typed(2, "&", Bt, v => v.head.asInstanceOf[B] & v.last.asInstanceOf[B])
```

6.5 分岐命令の設計

分岐命令は Jump 型を継承する。条件分岐や関数を呼び出す際に必要で、具体的に言えば、pc の値を操作する機能がある。

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

Skip 命令は、無条件分岐の命令で、指定された個数の命令を読み飛ばす。特に、条件分岐の処理から離脱する際に使う。
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、それに続く命令列を実行する関数を生成する。最初の命令の位置と環境を記録して、関数の直後に移動する。

```
case class Def(size: Int) extends Jump(vm => Some {
  vm.data.push(Closure(vm.pc + 1, 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、引数を回収して、関数の環境を構築する。関数を終えた後に戻る場所も記録して、関数の実行を開始する。

```
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)
  func.from
})
```

関数の正格評価は、Call 命令と Load 命令だけで実現できる。非正格評価の場合は、第 6.6 節に述べる命令が必要である。

6.6 遅延評価の設計

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。

```
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)
```

Set 命令は、引数に値を設定する。Fix 命令を実行すると、引数の値が確定する。以上の命令で、非正格評価を実現する。

第7章　コンパイラを作る

第7章では、第5章の仕様に従って、式を第6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7.1節から第7.2節で、様々な構文木を実装する。構文木は、命令列を生成する**コード生成器**の役割を兼ねる。

```
trait AST {
  def res(implicit env: Seq[DefST]): Type
  def gen(implicit env: Seq[DefST]): Seq[Code]
  def acc(unify: => Unit)(v: Type) = util.Try(unify).map(_ => v).get
}
```

resは、第7.3節で実装する型推論を実行して、式の型を決定する。genは、型推論の結果に従って、命令列を生成する。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索する。

7.1　定数と演算

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 res(implicit env: Seq[DefST]) = Atom(value.getClass)
  def gen(implicit env: Seq[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。ここで、特殊な文字の処理を行う。

```
case class StrST(string: String) extends AST {
  def res(implicit env: Seq[DefST]) = Atom(classOf[String])
  def gen(implicit env: Seq[DefST]) = LitST(StringContext.processEscapes(string)).gen
}
```

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 res(implicit env: Seq[DefST]) = acc(Form(v).unify(Form(expr.res)))(v)
  def gen(implicit env: Seq[DefST]) = expr.gen :+ UnOp.table(op, v.prune)
  val v = new Link
}
```

加減算の実装例を示す。2項演算では、まず左側の、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 AddST(op: String, e1: AST, e2: AST) extends AST {
  def res(implicit env: Seq[DefST]) = acc(Form(v, v).unify(Form(e1.res, e2.res)))(v)
  def gen(implicit env: Seq[DefST]) = e1.gen ++ e2.gen :+ AddOp.table(op, v.prune)
  val v = new Link
}
```

乗除算の実装例も示す。四則演算の演算子に対応する演算命令は、第6.4節で定義したAddOpやMulOpから取得できる。

```
case class MulST(op: String, e1: AST, e2: AST) extends AST {
  def res(implicit env: Seq[DefST]) = acc(Form(v, v).unify(Form(e1.res, e2.res)))(v)
  def gen(implicit env: Seq[DefST]) = e1.gen ++ e2.gen :+ MulOp.table(op, v.prune)
  val v = new Link
}
```

7.2 分岐と関数

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。条件分岐は、Skin 命令と Skip 命令の併用により実現する。条件式の真偽値により、次に実行する命令列が切り替わる。

```
case class IfST(c: AST, e: (AST, AST)) extends AST {
  def pos(pos: Seq[Code]) = (Skin(2 + pos.size) +: pos)
  def neg(neg: Seq[Code]) = (Skip(1 + neg.size) +: neg)
  def res(implicit env: Seq[DefST]) = acc(Form(Bt, v, v).unify(Form(c.res, e._1.res, e._2.res)))(v)
  def gen(implicit env: Seq[DefST]) = c.gen ++ pos(e._1.gen) ++ neg(e._2.gen)
  val v = new Link
}
```

次に、関数を表す DefST 型を実装する。引数は、関数の引数と内容である。また、外側の関数を参照する変数を定義する。関数を表す命令列は、関数閉包を生成する Def 命令を筆頭に、その関数が実行する命令列が続き、Ret 命令が配置される。

```
case class DefST(params: Seq[String], value: AST) extends AST {
  val args = params.map(_ -> new Link).toMap
  def get(name: String, depth: Int) = Load(depth, params.indexOf(name)) -> args(name)
  def res(implicit env: Seq[DefST]) = Form(params.map(args) :+ value.res(env :+ this) :_*)
  def gen(implicit env: Seq[DefST]) = tag(value.gen(env :+ this))
  def tag(codes: Seq[Code]) = Def(codes.size + 2) +: codes :+ Ret
}
```

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。該当する仮引数が存在した場合は、関数の入れ子の深さを数え、Load 命令を発行する。未定義の場合は、例外を投げる。

```
case class StIdST(val name: String) extends AST {
  def resolve(env: Seq[DefST], nest: Int = 0): (Load, Link) = {
    if(env.last.params.contains(name)) env.last.get(name, nest)
    else if(env.size >= 2) resolve(env.init, nest + 1)
    else sys.error(s"parameter $name is not declared")
  }
  def res(implicit env: Seq[DefST]) = resolve(env)._2.prune
  def gen(implicit env: Seq[DefST]) = Seq(resolve(env)._1)
}
```

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。引数とは、第 6.2 節に述べたプロミスである。計算済みの場合は、その値を使用する。各命令の詳細は第 6.6 節に述べた。

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

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 res(implicit env: Seq[DefST]) = body.res
  def gen(implicit env: Seq[DefST]) = DefST(Seq(), body).gen :+ 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 res(implicit env: Seq[DefST]) = acc(Form(args.map(_.res) :+ v :_*).prune.unify(f.res))(v)
  def gen(implicit env: Seq[DefST]) = f.gen ++ args.map(_.gen).flatten :+ Call(args.size)
  val v = new Link
}
```

7.3 型推論規則

第 3 章で議論した、型付きラムダ計算の型推論を実装する。手始めに、型変数や関数型の基底となる Type 型を実装する。unify には、制約条件の右辺を渡す。再帰処理を通じて、制約条件を消去する。prune は、型変数を型の値に変換する。

```
trait Type {
  def prune = this
  def unify(t: Type): Unit
}
```

次に、具体的な型を表す Atom 型を実装する。整数値や文字列など、型推論を始める時点で、型が明確な場合に使用する。

```
case class Atom(atom: Class[_]) extends Type {
  def unify(t: Type) = t.prune match {
    case t: Link => t.unify(this)
    case t: Type => require(this == t)
  }
}
```

次に、関数型を表す Form 型を実装する。引数の dom は、引数と関数の値の型を受け取る。式 (3.15) の推論規則に従う。

```
case class Form(dom: Type*) extends Type {
  def unify(t: Type) = t.prune match {
    case t: Form => t.align(this)
    case t: Type => t.unify(this)
  }
  def align(t: Form) = {
    require(this.dom.size == t.dom.size)
    dom.zip(t.dom).map(_.prune.unify(_))
  }
}
```

次に、型変数を表す Link 型を実装する。引数の to は、型の値を表す。unify を通じて確定し、prune は、その値を返す。なお、関数型と型変数の比較では、再帰構造が現れると無限再帰に陥るので、同値再帰型に対応した Loop 型で対策する。

```
class Link(var to: Option[Type] = None) extends Type {
  def unify(t: Type) = t.prune match {
    case t: Form => to = Some(Loop(t, this).prune)
    case t: Type => to = Option.when(this != t)(t)
  }
  override def prune = to.map(_.prune).getOrElse(this)
}
```

次に、同値再帰型を表す Loop 型を実装する。link に指定した型変数を展開すると、form の関数型が現れる様子を表す。厳密には、同値再帰型に限らず、型変数と関数型の対応関係を扱う型なので、prune で再帰構造を検出する実装とした。

```
case class Loop(form: Form, link: Link) extends Type {
  def unify(t: Type) = t match {
    case t: Form => link.unify(t)
    case t: Type => require(this == t)
  }
  override def prune = if(form.dom.contains(link)) this else form
}
```

最後に、Atom 型を継承し、第 5 章の基本型を定義する。関数型を除く、論理型と整数型と実数型と文字列型を定義する。

```
object Bt extends Atom(classOf[java.lang.Boolean])
object It extends Atom(classOf[java.lang.Integer])
object Dt extends Atom(classOf[java.lang.Double])
object St extends Atom(classOf[java.lang.String])
```

7.4 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 => LogST(op, _, _)))
  def and  = new Fold(eql, "&" ^ (op => LogST(op, _, _)))
  def eql  = new Fold(rel, """(!|=)=""".r ^ (op => EqlST(op, _, _)))
  def rel  = new Fold(add, """[<>]=?""".r ^ (op => RelST(op, _, _)))
  def add  = new Fold(mul, """[\+\-]""".r ^ (op => AddST(op, _, _)))
  def mul  = new Fold(unr, """[\*/%]""".r ^ (op => MulST(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 6 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。

7.5 ラムダ計算

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
```

無名関数でも、再帰計算を実現できる。式 (3.8) で議論した関数 $\mathbb{Y}$ を利用する。10 の階乗を計算する例を、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
```

正格評価の場合は、無限再帰に陥る。代替手段として式 (3.10) に掲載した関数 $\mathbb{Z}$ 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
```