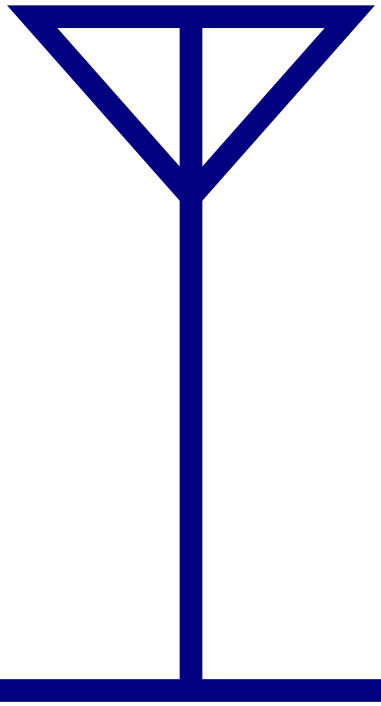
Journal of Hamradio Informatics No.2

Scalaで学ぶプログラミング言語の作り方

Computation Models & Compiler on Scala



無線部開発班　**平成 29 年 11 月 18 日改訂**

http://pafelog.net

目次

第1章　逆ポーランド電卓の実験

通常、数式は 1 + 2 などと**中置記法**で記す。演算子を後置して 1 2 +と記す流儀を**逆ポーランド記法**と呼ぶ。この記法は、言語処理系の代表的な計算模型のひとつである**スタック機械**の命令を記述する際に便利である。

1.1　スタック機械の実装

記憶領域として**スタック**を実装した計算機をスタック機械と呼ぶ。具体的な計算の仕組みを Fig. 1.1 に示す。

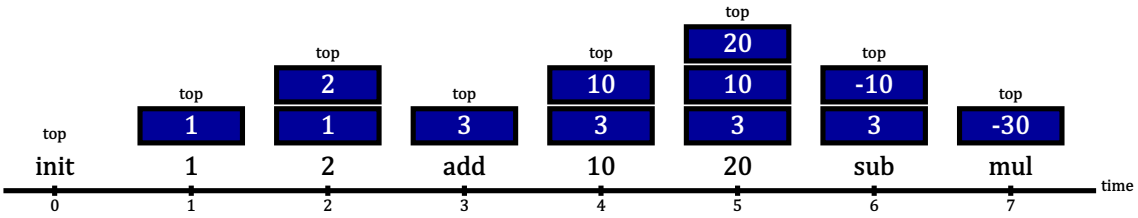


Fig. 1.1: arithmetic operation $(1+2) \times (10-20)$ in a stack machine.

まず命令 1 で 1 を、次に命令 2 で 2 をスタックに積み、命令 add で 1 + 2 を計算し、結果をスタックに戻す。続けて後続の命令を実行し、最終的な計算結果をスタックから取り出す。下記の StackMachine に実装する。

StackMachine.scala

```
object StackMachine {
  val stack = new collection.mutable.ArrayStack[Int]
  def apply(codes: Seq[String]) = codes.map(_ match {
    case "+" => stack(0) = stack(1) + stack.pop
    case "-" => stack(0) = stack(1) - stack.pop
    case "*" => stack(0) = stack(1) * stack.pop
    case "/" => stack(0) = stack(1) / stack.pop
    case num => stack.push(num.toInt)
  }).take(1).map(_ => stack.pop).last
}
```

命令は逆ポーランド記法で与える。数式を逆ポーランド記法に変換する**コンパイラ**は、第 1.2 節で実装する。

1.2　中置記法からの翻訳

数式を分解し、数値と演算子の列に変換する作業を**字句解析**と呼ぶ。下記の Tokenizer クラスに実装する。

Tokenizer.scala

```
class Tokenizer(expr: String) {
  val tokens = "[0-9]+|\\p{Punct}".r.findAllIn(expr).toList
  var cursor = Stream.from(0).iterator
  def next() = tokens.lift(cursor.next).getOrElse(null)
  def back() = cursor = Stream.from(cursor.next-1).iterator
}
```

数式の分割には、正規表現を利用する。字句解析で得られた演算子や被演算子を**字句**ないし**トークン**と呼ぶ。

test.scala

```
println((new Tokenizer("(1+20)*3-40/5")).tokens.map(t => s"'$t'"))
```

tokens メソッドを呼び出せば、字句の列を得られる。下記は、(1+20)*3-40/5 を字句解析した結果である。

```
List('(', '1', '+', '20', ')', '*', '3', '-', '40', '/', '5')
```

余談だが、有限状態機械が受理する語の集合を**正規言語**と呼び、正規言語を記述する手段が正規表現である。
次に実装する**構文解析器**は、字句の列を数式の文法に基づき解釈する。今回は**再帰下降構文解析**を採用する。

RecursiveDescentParser.scala

```
class RecursiveDescentParser(expr: String) {
  def parse = parseAdd(new Tokenizer(expr))
```

構文解析器では、演算子の優先順位を考慮して、演算子を探す。parseAdd メソッドは、加減算を担当する。

RecursiveDescentParser.scala

```
  def parseAdd(lex: Tokenizer): Seq[String] = {
    val buf = parseMul(lex).toBuffer
    while(true) lex.next() match {
      case "+" => buf ++= parseMul(lex) :+ "+"
      case "-" => buf ++= parseMul(lex) :+ "-"
      case _ => lex.back(); return buf.toList;
    }
    return buf.toList
  }
```

加減算の部分式を発見するたび、逆ポーランド記法に変換する。乗除算は、parseMul メソッドが担当する。

RecursiveDescentParser.scala

```
  def parseMul(lex: Tokenizer): Seq[String] = {
    val buf = parseNum(lex).toBuffer
    while(true) lex.next() match {
      case "*" => buf ++= parseNum(lex) :+ "*"
      case "/" => buf ++= parseNum(lex) :+ "/"
      case _ => lex.back(); return buf.toList;
    }
    return buf.toList
  }
```

最後に定義する parseNum メソッドは、加減算や乗除算より優先順位が高い、数値単体の部分式を担当する。

RecursiveDescentParser.scala

```
  def parseNum(lex: Tokenizer) = Seq(lex.next())
}
```

以上でコンパイラが完成した。式 $1+2\times3$ を命令列に変換して、StackMachine で計算する例を以下に示す。

test.scala

```
println(StackMachine(new RecursiveDescentParser("1+2*3").parse))
```

第 1 章の内容を応用すれば、**言語処理系**の自作も可能である。本格的な言語処理系は、第 3 章以降に述べる。

第2章　チューリング機械の実験

計算模型やプログラミング言語で計算可能な任意の問題には、それを計算する**チューリング機械**が存在する。チューリング機械とは、有限状態機械に Fig. 2.1 に示す無限長の**テープ**と**ヘッド**を追加した計算模型である。

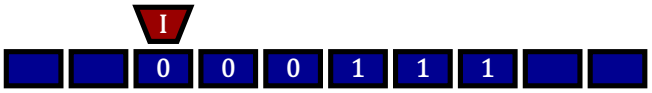


Fig. 2.1: an infinite tape and head of a Turing machine.

現実の計算機で言えば、有限状態機械は**プロセッサ**に、テープとヘッドは記憶装置と**メモリ番地**に相当する。TuringMachine クラスに実装する。コンストラクタには、有限状態機械の遷移表と、テープの初期値を渡す。

TuringMachine.scala

```
class TuringMachine(table: Map[(Char, Char), (Char, Char, Int)], init: Seq[Char]) {
  val tape = scala.collection.mutable.Map[Int, Char](Stream.from(0).zip(init):_*)
  var (head, state) = (0, 'I')
```

状態遷移表は、現状態とテープの値を引数とし、次状態とテープに書き戻す値とヘッドの移動量を指示する。

TuringMachine.scala

```
  while(state != 'F') table(state, tape.getOrElse(head, ' ')) match {
    case (st, w, move) => tape(head) = w; head += move; state = st;
  }
  val result = tape.keys.min.to(tape.keys.max).map(tape(_))
}
```

初期状態 I で起動し、受理状態 F に遷移すれば停止する。テープの値や**停止した事実**を以て計算結果とする。

2.1　自然数の後続の計算

下記は、テープ上の 2 進数を読み取って、その**後続**、即ち 1 を足した数を計算するチューリング機械である。

test.scala

```
val tm = new TuringMachine(Map(
  ('I', '0') -> ('a', '0', +1),
  ('I', '1') -> ('a', '1', +1),
  ('a', '0') -> ('a', '0', +1),
  ('a', '1') -> ('a', '1', +1),
  ('a', ' ') -> ('b', ' ', -1),
  ('b', '0') -> ('c', '1', -1),
  ('b', '1') -> ('b', '0', -1),
  ('b', ' ') -> ('F', '1', +0),
  ('c', '0') -> ('c', '0', -1),
  ('c', '1') -> ('c', '1', -1),
  ('c', ' ') -> ('F', ' ', +1)
), Seq('1', '0', '1', '1'))
```

第 1 章と同様に、ベクタ画像で視覚化を試みた。Fig. 2.2 は、2 進数 11 の次の整数 100 を求める様子である。

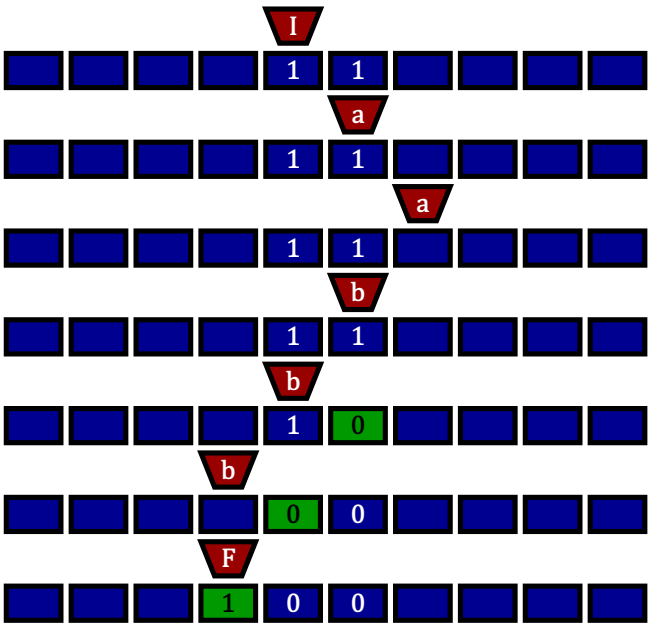


Fig. 2.2: increment from 11 to 100 in a Turing machine.

状態 a でビット列を右方向に走査しつつ終端を探し、終端に達したらビットを反転させて状態 b に遷移する。
状態 b はビット列を左方向に走査しつつ桁上げを行い、桁上げが終わると状態 c に遷移し、左端に到達する。

2.2 文脈自由言語の受理

下記は、テープ上の記号列を走査して、言語 $\{0^n1^n \mid n \geq 1\}$ に適合すると停止するチューリング機械である。
言語 $\{0^n1^n \mid n \geq 1\}$ は**文脈自由言語**の例であり、正規表現では表記不可能である。詳細は第 3 章で紹介する。

test.scala

```
val tm2 = new TuringMachine(Map(
  ('I', '0') -> ('a', '0', +1),
  ('I', '1') -> ('I', '1', +0),
  ('a', '0') -> ('a', '0', +1),
  ('a', '1') -> ('b', 'B', -1),
  ('a', ' ') -> ('a', ' ', +0),
  ('b', 'A') -> ('b', 'A', -1),
  ('b', 'B') -> ('b', 'B', -1),
  ('b', '0') -> ('c', 'A', +1),
  ('b', ' ') -> ('b', ' ', +0),
  ('c', 'A') -> ('c', 'A', +1),
  ('c', 'B') -> ('c', 'B', +1),
  ('c', '1') -> ('b', 'B', -1),
  ('c', ' ') -> ('d', ' ', -1),
  ('d', 'A') -> ('d', 'A', -1),
  ('d', 'B') -> ('d', 'B', -1),
  ('d', '0') -> ('d', '0', +0),
  ('d', ' ') -> ('F', ' ', +1)
), Seq('0', '0', '1', '1'))
```

状態 a で記号列を右方向に走査し、記号 1 を発見すると B に書き換え、状態 b に遷移して左方向に走査する。
以降は、テープ上を往復して、記号 0 は A に、記号 1 は B に書き換え、0 と 1 が同数と判明すれば停止する。

第3章　文脈自由文法と言語仕様

第 3 章からは、Scala の**パーサコンビネータ**を活用して**ラムダ計算**に基づくプログラミング言語を自作する。

3.1　バッカスナウア記法

英字や数字など**終端記号**の有限集合 Σ を**字母**と呼び、Σ に属する記号 σ を並べた**文**の集合 L を**言語**と呼ぶ。

$$L \subset \Sigma^* = \{\sigma_1\sigma_2...|\sigma_k \in \Sigma\}\,. \tag{3.1}$$

文法が記号の置換規則を表す言語を**形式言語**と呼び、中でも、正規言語と文脈自由言語は簡単な部類に入る。P を**生成規則**の集合、N を**非終端記号**の集合、S を**開始記号**とすると、形式文法 G は式 (3.2) で与えられる。

$$G = (N, \Sigma, P, S),\ \ P : N \to (N \cup \Sigma)^*,\ \ S \in N. \tag{3.2}$$

文脈自由言語では、生成規則は、非終端記号を非終端記号と終端記号の列に置換する。四則演算の例を示す。

```
数式  expr ::= add | mul | num
加算  add  ::= num ('+' | '-') num
乗算  mul  ::= num ('*' | '/') num
整数  num  ::= [0-9]+
```

記号の置換を表す上記の記法は**バッカスナウア記法**と呼ばれる。右辺に登場する各記号は下記の意味を持つ。

`A|B`	`A` もしくは `B`
`A B`	`A` の直後に `B`
`A+`	1 回以上の反復
`A*`	0 回以上の反復
`A?`	1 回以下の出現

前掲の四則演算の定義では、任意個の項を扱えず、演算の優先順位も未定義なので、下記のように修正する。

```
数式  expr ::= add
加算  add  ::= mul (('+' | '-') mul)*
乗算  mul  ::= num (('*' | '/') num)*
整数  num  ::= [0-9]+
```

文脈自由言語の解読は、置換した記号をスタックに記憶しつつ文字を読み進める**プッシュダウン機械**で行う。特に LL 法と呼ばれる実装では、開始記号 S を起点に、P から生成規則を選びながら、終端記号の列を得る。

$$(S = \texttt{expr}) \to (\texttt{add}) \to (\texttt{mul + mul}) \to (\texttt{num * num + num}) \to (\texttt{1 * 2 + 3}). \tag{3.3}$$

逆に LR 法と呼ばれる実装では、終端記号の列を起点に、P から生成規則を選びつつ、開始記号 S まで遡る。

$$(\texttt{1 * 2 + 3}) \to (\texttt{num * num + 3}) \to (\texttt{mul + 3}) \to (\texttt{mul + mul}) \to (\texttt{add}) \to (S). \tag{3.4}$$

LL 法は深さ優先型の探索アルゴリズムで実装できる。第 1.2 節で実装した再帰下降構文解析も LL 法である。LL 法は LR 法に比べて実装が容易だが、下記のような**左再帰**を含む文法では、無限再帰に陥る欠点がある。

加算 `add ::= add '+' mul | mul`

3.2 自作する言語の仕様

fava は強い動的型付けを行う言語である。データ型は整数型、実数型、論理型、文字列型、関数型がある。整数型は符号付き 32bit 整数値、実数型は IEEE 754 2 進倍精度小数である。文字列は UTF-16 で表現する。

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 `str  ::= '"' char* '"'`
関数型 `func ::= '(' (id (',' id)*)? ')' '=>' expr`

fava の関数は無名関数だが、*first-class function* でもある。例えば、関数の引数や返り値として指定できる。fava は明示的な型変換を行う機能を持たないが、整数と実数との四則演算は、暗黙的に実数に変換される。

```
fava$ 0.1 + 234
234.1
```

fava の識別子は、当該箇所を包含し、同名の仮引数を有する、最も内側の関数の仮引数として解決される。

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

fava のプログラムは単独のラムダ式である。繰り返し文や変数宣言、変数の再代入などの構文は排除する。fava では、式の意味と式の値は同義であり、部分式を同値な他の式に置換しても、式の意味は不変である。

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

fava では、式は**作用的順序**で評価され、演算子は *called by value* である。部分評価や部分適用は禁止する。fava の束縛変数は、それを引数に持つ関数が生存する限り保存される。このような関数を**関数閉包**と呼ぶ。

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

自作言語で無名関数の再帰を嗜むことは、言語処理系を自作する醍醐味である。詳細は第 4.2 節で述べるが、式 (4.14) の Z コンビネータを fava で実装し、無名関数を引数に与えれば、無名関数を再帰的に呼び出せる。

第4章　型なしラムダ計算の基礎

ラムダ計算はチューリング機械と等価な計算模型で、LISP や ML を始め、数多の言語の理論的基礎である。式 (4.1) は**ラムダ式**で、関数 `(x,y)=>2*x+3*y+1` を表す。記号 λ で関数を定義する作業を**ラムダ抽象**と呼ぶ。

$$\lambda xy.2x + 3y + 1. \tag{4.1}$$

複数の変数を持つ関数は、式 (4.2) に示す通り、1 変数の高階関数に変換できる。この変換を**カリー化**と呼ぶ。

$$\lambda xy.3x + 7y = \lambda x.\lambda y.3x + 7y. \tag{4.2}$$

`fava` も、明示的に高階関数に変換する必要があるものの、関数閉包を利用することでカリー化に対応する。

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

式 $\lambda x.E$ は、式 E に現れる変数 x を**束縛**し、x を実引数に紐づける。実引数を与える操作を**関数適用**と呼ぶ。式 (4.3) に示す通り、関数適用は左結合であり、まず引数 x を 2 で、次に引数 y を 3 で置換して、27 を得る。

$$\lambda x.\lambda y.3x + 7y \;\; 2 \;\; 3 = (((\lambda x.\lambda y.3x + 7y)\; 2)\; 3) = ((\lambda y.6 + 7y)\; 3) = 27. \tag{4.3}$$

形式的には、式 (4.4) の操作を**ベータ簡約**と呼び、その左辺を、引数 E_2 に対する関数 $\lambda x.E_1$ の適用と呼ぶ。

$$(\lambda x.E_1)E_2 \underset{\beta}{\rightarrow} E_1|_{x:=E_2}. \tag{4.4}$$

4.1　チャーチ符号化

ラムダ計算は、計算可能な任意の関数を表現する能力を持ち、論理演算をラムダ式で定義することもできる。

$$\begin{cases} T = \lambda xy.x, \\ F = \lambda xy.y. \end{cases} \tag{4.5}$$

式 (4.5) は、**チャーチ論理値**による真と偽の表現である。各種の演算子は、式 (4.6) に示す通りに定義できる。

$$\begin{cases} x \wedge y = \lambda xy.xyF, \\ x \vee y = \lambda xy.xTy. \end{cases} \tag{4.6}$$

式 (4.5)(4.6) で符号化された論理値が**ブール論理**の性質を満たす様子は、`fava` でも下記の通りに確認できる。

```
fava$ ((x)=>(y)=>x(y)((x)=>(y)=>y))((x)=>(y)=>x)((x)=>(y)=>y)(true)(false)
false
fava$ ((x)=>(y)=>x((x)=>(y)=>x)(y))((x)=>(y)=>x)((x)=>(y)=>y)(true)(false)
true
```

自然数 n も**ペアノの公理**に基づき、変数 x に対する n 回の関数適用で表現できる。これを**チャーチ数**と呼ぶ。

$$\begin{cases} 0 = \lambda fx.x, \\ n+1 = \lambda nfx.f(nfx). \end{cases} \tag{4.7}$$

式 (4.7) で表現した自然数に対し、加算や乗算を式 (4.8) で定義できる。これも正当性を fava で検証できる。

$$\begin{cases} a+b = \lambda ab.\lambda fx.af(bfx), \\ a \times b = \lambda ab.\lambda fx.a(bf)x. \end{cases} \tag{4.8}$$

変数 f に関数 (x)=>x+1 を、変数 x に定数 0 を与えると、整数型に変換されるため、検証の際に便利である。

```
fava$ ((a)=>(b)=>(f)=>(x)=>a(f)(b(f)(x)))((f)=>(x)=>f(x))((f)=>(x)=>f(f(x)))((x)=>x+1)(0)
3
fava$ ((a)=>(b)=>(f)=>(x)=>a(b(f))(x))((f)=>(x)=>f(f(x)))((f)=>(x)=>f(f(x)))((x)=>x+1)(0)
4
```

4.2 無名関数の再帰

ラムダ計算には再帰関数を定義する規則はないが、**不動点コンビネータ**を利用すれば再帰関数を表現できる。関数 $f(x)$ に対し $p = f(p)$ となる点 p を**不動点**と呼び、p を求める高階関数 g を不動点コンビネータと呼ぶ。

$$g(f) = f(g(f)). \tag{4.9}$$

関数 $h(x)$ を、関数 f が変数として出現する式 E と不動点コンビネータ g により、式 (4.10) の通り定義する。

$$h(x) = g\lambda f.\lambda x.E. \tag{4.10}$$

式 (4.9) を式 (4.10) に代入すると、式 (4.11) を得る。式 E の中の変数 f は、関数 $h(x)$ により束縛される。

$$h(x) = (\lambda f.\lambda x.E)(g\lambda f.\lambda x.E) = (\lambda f.\lambda x.E)(h(x)) = \lambda x.E|_{f:=h(x)}. \tag{4.11}$$

式 (4.11) は、関数 $h(x)$ が引数 f を通じて関数 $h(x)$ を参照する様子を表し、関数 $h(x)$ を再帰的に呼び出す。関数 g の候補は星の数ほど存在するが、特に有名な例として、Haskell Curry の Y コンビネータを紹介する。

$$\boldsymbol{y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)). \tag{4.12}$$

関数 h に対し、式 (4.12) の Y コンビネータが式 (4.9) の性質を満たすことは、式 (4.13) により明らかである。

$$\boldsymbol{y}h \underset{\beta}{\rightarrow} (\lambda x.h(xx))(\lambda x.h(xx)) \underset{\beta}{\rightarrow} h((\lambda x.h(xx))(\lambda x.h(xx))) = h(\boldsymbol{y}h). \tag{4.13}$$

fava で Y コンビネータを利用して 10 の階乗を求める例を以下に示す。しかし、残念ながら無限再帰に陥る。

```
fava$ ((f)=>((x)=>f(x(x)))((x)=>f(x(x))))((f)=>(n)=>(n==0)?1:n*f(n-1))(10)
program does not halt
```

$\boldsymbol{y}h$ を無限に $h(\boldsymbol{y}h)$ に展開するためである。**イータ変換**により、引数の評価を遅延させる対策が有効である。

$$\boldsymbol{z} = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)) \underset{\eta^{-1}}{\longleftarrow} \boldsymbol{y}. \tag{4.14}$$

式 (4.14) を Z コンビネータと呼び、*call by value* を原則とする言語で Y コンビネータの代わりに使われる。

```
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
```

なお、式 (4.15) で、式 E に変数 x が未束縛で出現しない場合、ラムダ抽象を外す操作をイータ変換と呼ぶ。

$$\lambda x.Ex \underset{\eta}{\rightarrow} E \quad \cdots\cdots \quad fx = gx \leftrightarrow \lambda x.fx = \lambda x.gx \overset{\eta^{-1}}{\underset{\eta}{\longleftrightarrow}} f = g. \tag{4.15}$$

式 (4.15) の操作を許容する場合、任意の引数 x に対し同値関係にある関数 f と g は**外延的に同値**と言える。

第5章　高機能な仮想機械の設計

第5章では、第1章で実装したスタック機械を発展させ、自作言語 fava の実行環境と命令体系を実装する。

fava.scala

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

下記の VirtualMachine は、変数 pc が示す番地の命令を順番に実行する。pc を**プログラムカウンタ**と呼ぶ。

fava.scala

```
object VirtualMachine extends Function1[Seq[Code], Any] {
  def apply(codes: Seq[Code]): Any = {
    val stack = new DualStack
    var pc = 0
    while(pc < codes.size) pc = codes(pc).exec(stack, pc)
    stack.data.pop
  }
}
```

基本的な仕組みは第1章で実装した StackMachine と同じだが、条件分岐や関数適用のための機構を備える。例えば、演算用のスタックに加え、関数の情報を管理する**コールスタック**を DualStack クラスに装備する。

fava.scala

```
class DualStack {
  val call = new Stack[Env]
  val data = new Stack[Any]
}
```

個別のスタックは、下記の Stack クラスで実装する。スタックから値を取り去る pop メソッド等を定義する。

fava.scala

```
class Stack[E] extends collection.mutable.ArrayStack[E] {
  def apply[T](f: (Any, Any)=>T): T = f(this(1), this(0))
  def swap(n: Int) = 1.to(n).map(_=>pop).foreach(push(_))
  def popN(n: Int) = 1.to(n).map(_=>pop).reverse
  def popAs[Type]: Type = pop.asInstanceOf[Type]
  def env = (this :+ null).top.asInstanceOf[Env]
}
```

命令は下記の Code トレイトを継承する。スタックと pc を受け取り、命令を実行し、次の命令の位置を返す。

fava.scala

```
trait Code {
  def exec(stack: DualStack, pc: Int): Int
}
```

条件分岐や関数適用の命令を除けば、その命令の次に実行すべき命令は、その命令の直後に並ぶ命令である。

5.1 演算命令とロード命令

fava の**演算命令**は、全て下記の Bin クラスを継承した 2 項演算である。単項演算は適当な被演算子を補う。

fava.scala

```
abstract class Bin(op: String) extends Code {
  def apply(a: Any, b: Any): Option[Any]
  def exec(stack: DualStack, pc: Int) = Try {
    val res = stack.data(apply _).get
    stack.data.popN(2)
    stack.data.push(res)
    pc + 1
  }.getOrElse(throw new TypeError(op, stack))
}
```

実装例として、演算子-に対応する Sub 命令を掲載する。なお、単項演算子の式-foo は式 0-foo として扱う。

fava.scala

```
case class Sub() extends Bin("-") {
  def apply(a: Any, b: Any) = Some(a->b) collect {
    case (val1: I, val2: I) => val1 - val2
    case (val1: I, val2: D) => val1 - val2
    case (val1: D, val2: I) => val1 - val2
    case (val1: D, val2: D) => val1 - val2
  }
}
```

上記の match 式は、被演算子の型を確認する。型が不正な場合、下記の TypeError を投げる仕組みである。

fava.scala

```
class TypeError(op: String, stack: DualStack) extends ScriptException(
  stack.data.popN(2).map(a => s"$a:${a.getClass.getSimpleName}").mkString(s" $op ")
)
```

下記の Gt クラスは、不等号>に相当する。同様に、他の**算術論理演算**や**比較演算**の命令も漏れなく実装する。

fava.scala

```
case class Gt() extends Bin(">") {
  def apply(a: Any, b: Any) = Some(a->b) collect {
    case (val1: I, val2: I) => val1 > val2
    case (val1: I, val2: D) => val1 > val2
    case (val1: D, val2: I) => val1 > val2
    case (val1: D, val2: D) => val1 > val2
    case (val1: S, val2: S) => val1 > val2
  }
}
```

最後に、整数や文字列などの即値をスタックに積む**ロード命令**を実装する。下記の Push クラスに実装する。

fava.scala

```
case class Push(value: Any) extends Code {
  def exec(stack: DualStack, pc: Int) = {
    stack.data.push(value)
    pc + 1
  }
}
```

5.2 条件分岐と関数の実現

条件分岐は Jmp と Jmpf の**分岐命令**で実現する。例えば、式 true?"A":"B"は、下記の命令列に翻訳される。

```
Push(true) Jmpf(3) Push("A") Jmp(2) Push("B")
```

2 番目の Jmpf 命令は、冒頭の条件式の値が false なら、変数 pc に引数の 3 を加えて、式の後半に移動する。

fava.scala

```
case class Jmp(carry: Int) extends Code {
  def exec(stack: DualStack, pc: Int) = pc + carry
}
```

4 番目の Jmp 命令は、条件が真の場合の式を評価した後、続けて偽の式を評価する事態を防ぐ仕組みである。

fava.scala

```
case class Jmpf(carry: Int) extends Code {
  def exec(stack: DualStack, pc: Int) = pc + test(stack.data)
  def test(stack: Stack[_]) = if(stack.popAs[B]) 1 else carry
}
```

以後、関数を定義して呼び出す仕組みを設計する。まず、コールスタックで引数を管理する**環境**を実装する。

fava.scala

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

関数型は Closure クラスで表す。from は関数の位置を、out は関数の外の変数を管理する環境を参照する。

fava.scala

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

fava の関数は、Def 命令で生成される。例えば、下記は、式 (x,y)=>x+y から関数を生成する命令列である。

```
Def(5) Load(0,0) Load(0,1) Add() Ret()
```

Def 命令は、その位置を起点に関数を生成した後、関数の部分を飛ばすため、変数 pc に引数の 5 を加える。

fava.scala

```
case class Def(size: Int) extends Code {
  def exec(stack: DualStack, pc: Int) = {
    stack.data.push(Closure(pc + 1, stack.call.env))
    pc + size
  }
}
```

Load 命令は、関数の引数を読み出す。id が引数の識別番号で、nest は関数の外の変数を参照する際に使う。

fava.scala

```
case class Load(nest: Int, id: Int) extends Code {
  def exec(stack: DualStack, pc: Int) = {
    stack.data.push(stack.call.env(nest, id))
    pc + 1
  }
}
```

関数定義の末尾の Ret 命令は、関数を呼び出す前の文脈をスタックから復元し、その場所に戻る命令である。

fava.scala

```
case class Ret() extends Code {
  def exec(stack: DualStack, pc: Int) = {
    stack.data.swap(2)
    stack.call.popAs[Env]
    stack.data.popAs[Int]
  }
}
```

次に、関数を呼び出す Call 命令を実装しよう。fava は式 ((x,y)=>x+y)(3,5) を下記の命令列に翻訳する。

```
Def(5) Load(0,0) Load(0,1) Add() Ret() Push(3) Push(5) Call(2)
```

上記の命令列を実行するスタック機械の動作を Fig. 5.1 に示す。上下 2 段の DualStack の挙動に注目しよう。

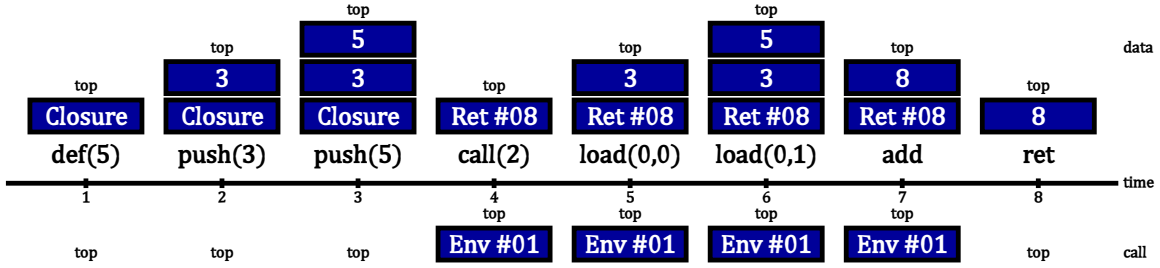


Fig. 5.1: ((x,y)=>x+y)(3,5).

Call は引数を回収して環境を生成し、直後の命令の位置を控えた後、関数の場所に移動する命令と言える。

fava.scala

```
case class Call(argc: Int) extends Code {
  def exec(stack: DualStack, pc: Int) = {
    val args = stack.data.popN(argc)
    val func = stack.data.popAs[Closure]
    stack.data.push(pc + 1)
    stack.call.push(new Env(args, func.out))
    func.from
  }
}
```

他の例も視覚化してみよう。Fig. 5.2 は式 ((f)=>f())(()=>3) を評価する際のスタック機械の挙動である。

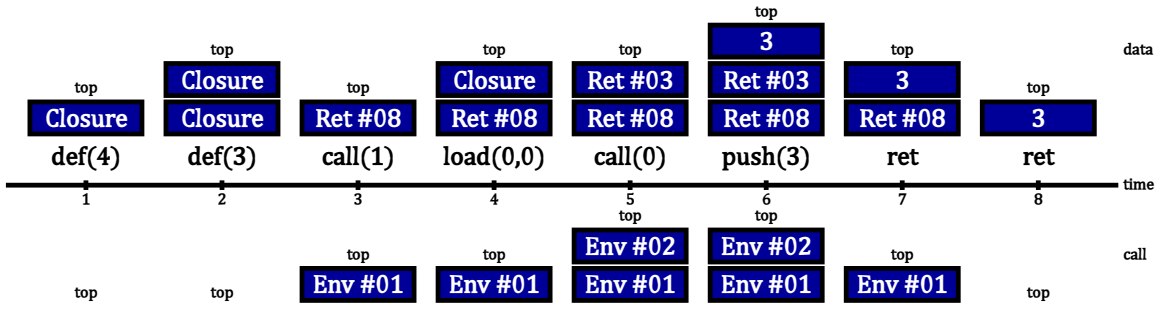


Fig. 5.2: ((f)=>f())(()=>3).

第 6 章では、スタック機械で実行する命令を出力するコンパイラを実装し、自作言語処理系の完成を目指す。

第6章　コンパイラの実装と拡張

第 1 章では手で実装した再帰下降構文解析だが、Scala のパーサコンビネータを使えば、手軽に実装できる。生成規則を表す関数を~や chainl1 などの高階関数で結合し、構文解析器を宣言的に実装する仕組みである。

fava.scala

```
object Parser extends scala.util.parsing.combinator.JavaTokenParsers {
  def parse(str: String): AST = parseAll(expr, str) match {
    case Success(ast, _) => ast
    case NoSuccess(msg, _) => throw new ScriptException(msg)
  }
  def expr: Parser[AST] = cond|or
  def cond = (or<~"?")~expr~(":"~>expr)^^{case c~y~n => If(c, y, n)}
  def or  = chainl1(and, "|"^^(op => (Bin(op, _: AST, _: AST))))
  def and = chainl1(eql, "&"^^(op => (Bin(op, _: AST, _: AST))))
  def eql = chainl1(rel, """(!|=)=""".r^^(op => (Bin(op, _: AST, _: AST))))
  def rel = chainl1(add, """[<>]=?""".r^^(op => (Bin(op, _: AST, _: AST))))
  def add = chainl1(mul, """[\+\-]""".r^^(op => (Bin(op, _: AST, _: AST))))
  def mul = chainl1(unr, """[\*/%]""".r^^(op => (Bin(op, _: AST, _: AST))))
  def unr = rep("+"|"-"|"!")~call^^{case o~e => o.foldRight(e)(Unary(_,_))}
  def call = fact~rep(args)^^{case f~a => a.foldLeft(f)(Call(_,_))}
  def args = "("~>repsep(expr, ",")<~")"
  def fact = func|bool|real|int|str|name|"("~>expr<~")"
  def func = pars~"=>"~expr^^{case p~_~e => Def(e, p)}
  def pars = "("~>repsep(name, ",")<~")"^^(_.map(_.ident))
  def bool = ("true"|"false")^^(b => Lit(b.toBoolean))
  def real = """(\d+\.\d*|\d*\.\d+)""".r^^(d => Lit(d.toDouble))
  def int  = """\d+""".r^^(i => Lit(i.toInt))
  def str  = stringLiteral^^(s => Lit(s.tail.init))
  def name = ident^^(Id(_))
}
```

詳細は Scala の API 仕様書に譲るが、終端記号を正規表現で記述する仕組みなので字句解析器が不要である。後は、第 6.1 節の**抽象構文木**と併せてコンパイラを構成し、第 5 章の仮想機械とともに**対話環境**に連接する。

fava.scala

```
def main(args: Array[String]) {
  val jline = new scala.tools.jline.console.ConsoleReader
  jline.setExpandEvents(false)
  jline.setPrompt(s"${Console.BLUE}fava$$ ${Console.RESET}")
  while(true) Try {
    val codes = Parser.parse(jline.readLine).code(Def(null))
    println(s"${Console.CYAN}${VirtualMachine(codes)}")
  }.recover{case ex => println(Console.RED + ex.getMessage)}
}
```

完成した言語処理系は、LGPL の許諾のもと Git で頒布している。紙面の都合で省略した部分を閲覧できる。

```
$ git clone https://github.com/nextzlog/fava
$ java -jar fava/build/libs/fava.jar
fava$
```

6.1 抽象構文木の実装

実用的なコンパイラでは、構文解析の結果をまず**抽象構文木**と呼ばれる木構造に書き下してから処理を施す。下記は、式 (1+2)*(10-20) の抽象構文木の例である。Lit は定数を、Add と Mul は加減算と乗除算を表す。

```
Mul(*,Add(+,Lit(1),Lit(2)),Add(-,Lit(10),Lit(20)))
```

最終的に**コード生成器**で下記の命令列に変換される。第 1 章の場合は、構文解析器がコード生成器も兼ねた。

```
Push(1) Push(2) Add() Push(10) Push(20) Sub() Mul()
```

fava の抽象構文木は、下記の AST トレイトを継承する。code メソッドを実行すると、命令列が生成される。

fava.scala

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

暗黙の引数 env は、式に変数を提供しうる最も内側の関数である。では、定数を表す Lit クラスを実装する。

fava.scala

```
case class Lit(value: Any) extends AST {
  def code(implicit env: Def) = Seq(is.Push(value))
}
```

下記の Bin クラスは、2 項演算の抽象構文木である。被演算子の命令列を展開した後に演算命令を挿入する。

fava.scala

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

下記の Unary クラスは、単項演算を表す。被演算子を補足して、2 項演算に変換した後、命令列に変換する。

fava.scala

```
case class Unary(op: String, expr: AST) extends AST {
  def code(implicit env: Def): Seq[Code] = op match {
    case "+" => Bin("+", Lit(0),    expr).code
    case "-" => Bin("-", Lit(0),    expr).code
    case "!" => Bin("^", Lit(true), expr).code
  }
}
```

下記の Id クラスは、識別子の抽象構文木である。環境を辿り、該当の仮引数を探し、Load 命令を生成する。

fava.scala

```
case class Id(ident: String) extends AST {
  def find(implicit env: Def): is.Load = {
    val idx = env.pars.indexOf(ident)
    if(idx >= 0) return is.Load(0, idx)
    if(env.out != null) return {
      val load = find(env.out)
      is.Load(load.nest + 1, load.id)
    } else throw new NameError(ident)
  }
  def code(implicit env: Def) = Seq(find)
}
```

最も外側の環境まで遡っても、同名の仮引数を発見できない場合は、下記の NameError を投げて警告する。

fava.scala

```
class NameError(id: String) extends ScriptException(s"$id is not defined")
```

下記の If クラスは、条件分岐を表す抽象構文木である。第 5.1 節で実装した分岐命令は、ここで使用する。

fava.scala

```
case class If(cond: AST, val1: AST, val2: AST) extends AST {
  def code(implicit env: Def): Seq[Code] = {
    val (code1, code2) = (val1.code, val2.code)
    val jmp1 = is.Jmpf(2 + code1.size) +: code1
    val jmp2 = is.Jmp (1 + code2.size) +: code2
    cond.code ++ jmp1 ++ jmp2
  }
}
```

下記の Call クラスは、関数適用の抽象構文木である。関数と実引数を命令列に変換し、Call 命令を加える。

fava.scala

```
case class Call(func: AST, args: Seq[AST]) extends AST {
  def code(implicit env: Def): Seq[Code] = {
    val vals = args.map(_.code).fold(Seq())(_++_)
    func.code ++ vals :+ is.Call(args.size)
  }
}
```

下記の Def クラスは、関数を定義する。Def 命令を先頭に、関数の内容を挟んで、末尾に Ret 命令を加える。

fava.scala

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

out は、外側の関数を参照する。暗黙の引数 env を記憶することで、Id クラスでの識別子の探索に役立てる。

6.2 遅延評価への変更

第 6.2 節では、関数が仮引数を参照する時点まで、対応する実引数の評価を遅延させる**遅延評価**を実現する。次式を *call by value* で評価すると、式 1+2 の値 3 と式 3+4 の値 7 は、Call 命令を実行する前に計算される。

```
fava$ ((x,y)=>x*y)(1+2,3+4)
21
```

同じ式を *call by name* で評価すると、実引数は Call 命令の実行後、Load 命令で参照する際に計算される。これは**非正格評価**とも呼ばれる。*call by value* でも、高階関数を利用することで、同等の挙動を再現できる。

```
fava$ ((x,y)=>x()*y())(()=>1+2,()=>3+4)
21
```

実引数の評価を遅延する目的で利用される関数閉包を**サンク**と呼び、引数の値を計算する操作を**強制**と呼ぶ。ただし、引数を参照する度に評価する上記の実装は、同じ引数を何度も参照する場合には、非効率的である。

```
fava$ ((x)=>x()*x())(()=>3+3)
36
```

実引数を評価した際に、その値を環境に登録して、評価の回数を抑制する非正格評価を *call by need* と呼ぶ。第 6.2 節では *call by name* を実装する。まず Id クラスを改修し、Load 命令の直後に Call 命令を挿入する。

fava.scala

```
case class LazyId(ident: String) extends AST {
  def code(implicit env: Def): Seq[Code] = {
    Seq(Id(ident).find, is.Call(0))
  }
}
```

同様に、関数適用の Call クラスも改修する。関数定義の Def クラスを利用して、実引数を関数定義で包む。

fava.scala

```
case class LazyCall(func: AST, args: Seq[AST]) extends AST {
  def code(implicit env: Def): Seq[Code] = {
    Call(func, args.map(Def(_))).code
  }
}
```

第 6 章の構文解析器を改修して、Call 命令を LazyCall 命令に、構文木 Id を構文木 LazyId に置き換える。

fava.scala

```
  def call = fact~rep(args)^^{case f~a => a.foldLeft(f)(LazyCall(_,_))}
  def name = ident^^(LazyId(_))
```

僅かな改修で fava は *call by name* に移行した。Y コンビネータの評価が無限再帰に陥る問題も解消される。

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

ここから発展して *call by need* を実装する際は、実引数を関数で包む代わりに、下記の Cache クラスで包む。

fava.scala

```
case class Cache(thunk: Closure) {
  var cache: Option[Any] = None
}
```

Load 命令で参照する際に、cache が None であれば thunk を呼び出し、Some であれば値を取り出せば良い。

第7章　ガベージコレクタの実験

関数適用は、引数の数だけ多くのメモリ領域を消費する。関数適用が終了すれば、メモリ領域は解放できる。だが、下記の高階関数では、外側の関数が終了しても、内側の関数は束縛変数 $x = 2$ を参照する必要がある。

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

第 7 章で実装する**ガベージコレクタ**は、変数のメモリを解放しても良いか、自動的に判断する仕組みである。自作せずとも Java の高性能なガベージコレクタの恩恵に与れるが、折角なので C++で簡単に再現してみる。

7.1　計数型の実装

C++11 の `shared_ptr<>`は *reference count* と呼ばれるガベージコレクタの実装である。Fig. 7.1 に図示する。

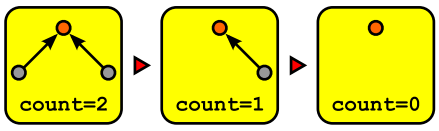


Fig. 7.1: reference-counting garbage collector.

オブジェクトは個別に**参照カウンタ**を持ち、被参照数が増減した際に、直ちに参照カウンタの値に反映する。参照の増減を監視する機構が必要だが、C++の場合は、コンストラクタやデストラクタを用いて実現できる。

count.cpp

```
template<typename T>
shared_ptr<T>::shared_ptr(T *ptr): ptr(ptr) {
  count[ptr]++;
}

template<typename T>
shared_ptr<T>::~shared_ptr() {
  if(!--count[ptr]) delete ptr;
}
```

`count` は大域的な配列であり、`shared_ptr` の生成と破棄により、指定のポインタに対応する値が増減する。使用例を以下に示す。1 行目で確保したメモリは `ptr1` と `ptr2` で共有されるが、6 行目で `ptr2` が脱落する。

count.cpp

```
shared_ptr<int> ptr1(new int);
{
  shared_ptr<int> ptr2 = ptr1;
  *ptr2 = 114514;
}
std::cout << *ptr1 << std::endl;
```

reference count は、オブジェクトが不要になると迅速に解放できるが、**循環参照**を解放できない難点がある。

7.2 走査型の実装

Fig. 7.2 の *mark and sweep* は、第 7.1 節の *reference count* と並び、主要なガベージコレクタの手法である。

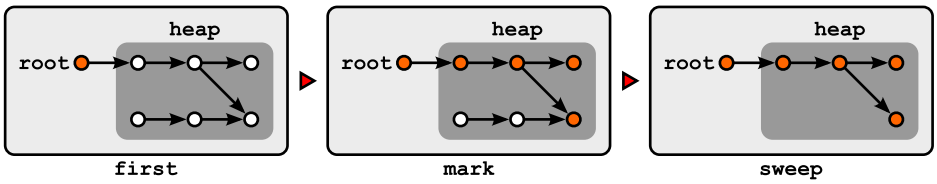


Fig. 7.2: mark-and-sweep garbage collector.

まず、スタックを起点に参照を辿り、巡回したオブジェクトに生存の印を付ける。この操作を**マーク**と呼ぶ。次に、ヒープ領域を走査し、生存の印が付いていないオブジェクトを削除する。この操作を**スイープ**と呼ぶ。第 7.1 節の *reference count* と比べ、停止時間が長くなるが、循環参照を回収できる。では、実装してみよう。

noah.hpp

```
#include <cstdlib>
#include <vector>
namespace noah {
void* alloc(size_t size);
void  clean(void *from, void *last);
};
```

上記の noah::alloc 関数は stdlib の malloc に相当し、noah::clean 関数は *mark and sweep* を実行する。次に、noah::alloc 関数で割り当てたポインタの範囲と生存の印を管理するための構造体 Head を定義する。

noah.cpp

```
namespace noah {
struct Head {
  bool marked;
  void *from;
  void *last;
};
```

構造体 Head は noah::alloc 関数を呼び出す度にベクタに追加する。include 宣言は紙面の都合で省略する。

noah.cpp

```
std::vector<Head*> heads;
```

noah::alloc 関数は、指定されたバイト数のメモリをヒープ領域に確保し、heads ベクタの末尾に登録する。

noah.cpp

```
void* alloc(size_t bytes) {
  Head *head = new Head;
  heads.push_back(head);
  void *p = malloc(bytes);
  size_t a1 = (size_t) p;
  size_t a2 = a1 + bytes;
  head->from = (void*) a1;
  head->last = (void*) a2;
  head->marked = false;
  return p;
}
```

以降は *mark and sweep* を実装する。下記の noah::owner 関数は、指定されたポインタの Head を検索する。

noah.cpp

```
static Head* owner(void *ptr) {
  for(Head* &head : heads) {
    void *from = head->from;
    void *last = head->last;
    if(ptr < from) continue;
    if(ptr > last) continue;
    return head;
  }
  return NULL;
}
```

noah::mark 関数は、指定されたポインタをメンバに持つオブジェクトに印を付け、その参照先を巡回する。

noah.cpp

```
static void mark(void *ptr) {
  Head *head = owner(ptr);
  if(head == NULL) return;
  if(head->marked) return;
  head->marked = true;
  void *from = head->from;
  void *last = head->last;
  size_t s = (size_t) from;
  size_t e = (size_t) last;
  for(size_t i=s; i<e; i++) {
    mark(*(void**)i);
  }
}
```

noah::sweep 関数は、指定された Head を確認し、無印ならばメモリを回収し、heads ベクタから削除する。

noah.cpp

```
static void sweep(size_t idx) {
  Head *head = heads[idx];
  auto it = heads.begin();
  if(!head->marked) {
    heads.erase(it + idx);
    free(head->from);
    delete head;
  } else head->marked = false;
}
```

noah::clean 関数は、指定された範囲に所在の全てのポインタを起点として、*mark and sweep* を実行する。

noah.cpp

```
void clean(void *from, void *last) {
  size_t s = (size_t) from;
  size_t e = (size_t) last;
  for(size_t i=s; i<e; i++) {
    mark(*(void**)i);
  }
  size_t i = heads.size();
  while(i-- > 0) sweep(i);
}
};
```

以上は、スタックに積まれた値が全てポインタであると仮定する**保守的なガベージコレクタ**の実装例である。数値の場合、*mark and sweep* をしても無駄であるが、言語処理系の協力がなければ、判別は不可能である。とまれ、*mark and sweep* 方式のガベージコレクタが完成した。shared オプションを付けてコンパイルする。

```
$ g++ -shared -fPIC -o libnoah.so noah.cpp
```

共有ライブラリ libnoah.so が生成される。動作確認を兼ねて、サンプルアプリケーションを実装してみる。

test.cpp

```
#include "noah.hpp"

struct cons {
  cons *car;
  cons *cdr;
};

int main(void) {
  void **root = new void*[2];
  cons *cons1 = (cons*) noah::alloc(sizeof(cons));
  cons *cons2 = (cons*) noah::alloc(sizeof(cons));
  cons *cons3 = (cons*) noah::alloc(sizeof(cons));
  cons *cons4 = (cons*) noah::alloc(sizeof(cons));
  cons *cons5 = (cons*) noah::alloc(sizeof(cons));
  root[0] = cons1;
  root[1] = cons2;
  cons3->car = cons4;
  cons4->car = cons5;
  cons5->car = cons3;
  noah::clean(root, &root[2]);
}
```

libnoah.so を言語処理系に組み込む場合は、定期的に noah::clean 関数を実行するように実装すると良い。test.cpp は下記のコマンドで実行する。環境変数 LD_LIBRARY_PATH を設定し、libnoah.so をリンクする。

```
$ g++ -L. -lnoah -o test test.cpp
$ export LD_LIBRARY_PATH=.
$ ./test
```

最後に、*mark and sweep* と並ぶ主要な走査型のアルゴリズムとして、Fig. 7.3 の *stop and copy* を紹介する。

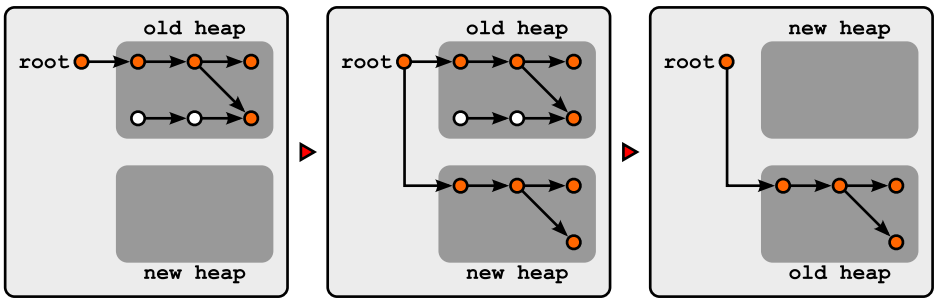


Fig. 7.3: stop-and-copy garbage collector.

まず、スタックを起点に参照を辿り、巡回したオブジェクトを新しい領域に移す。この操作を**コピー**と呼ぶ。転写に漏れたオブジェクトは、古いヒープ領域と共に破棄する。*mark and sweep* に比べ、軽快に動作する。転写の際に、オブジェクトの参照関係に応じて位置を調整すれば、キャッシュヒット率の改善も期待できる。

付録 A　リスト指向言語の処理系

付録 A では、本稿の本筋とは趣向を変え、構文を自在に拡張できるプログラミング言語の処理系を実装する。

```
elva$ (+ 1 2 3)
6
```

これは独自の LISP であり、elva と名付ける。elva は Lisp-1 であり、関数と変数は共通の名前空間に属す。

```
elva$ (set 'function-in-variable list)
<function>
elva$ (function-in-variable 1 2 3 4 5)
(1 2 3 4 5)
```

Scheme と同様に、静的スコープであり、関数閉包にも対応する。関数は、define-lambda により定義する。

```
elva$ (define-lambda fact (x) (if (eq x 1) x (* x (fact (- x 1)))))
(lambda (x) (if (eq x 1) x (* x (fact (- x 1)))))
```

elva では**マクロ**を定義して、構文を自在に拡張できる。例えば、変数宣言に利用する setq もマクロである。

```
elva$ setq
(syntax (name value) (list (quote set) (list (quote quote) name) value))
```

syntax 関数はマクロを生成する。関数閉包は lambda 関数で生成する。名前は setq を利用して与えている。

```
elva$ define-lambda
(syntax (name pars body) (list (quote setq) name (list (quote lambda) pars body)))
elva$ define-syntax
(syntax (name pars body) (list (quote setq) name (list (quote syntax) pars body)))
```

ドット対は、Common Lisp や Scheme と異なり、隠蔽される。代わりに、リストを基本型として提供する。

A.1　式の構文解析

構文解析器は、S 式を読み取って、LISP の構文木たるリスト構造を構築する。マクロ文字には未対応である。

elva.scala

```
object Parser extends util.parsing.combinator.JavaTokenParsers {
  def parse(str: String): Any = parseAll(sexp, str) match {
    case Success(exp, _) => exp
    case Failure(msg, _) => sys.error(s"error: $msg")
    case Error  (msg, _) => sys.error(s"fatal: $msg")
  }
  def sexp: Parser[Any] = quot|text|real|list|name
  def quot = "'"~>sexp^^(Seq('quote, _))
  def text = stringLiteral^^(_.tail.init)
  def real = """\d+\.?\d*(?=$|\s|\(|\))""".r^^(BigDecimal(_))
  def list = "("~>rep(sexp)<~")"
  def name = """[^\(\)\s]+""".r^^(Symbol(_))
}
```

識別子には、空白や () 以外の任意の印字可能な ASCII 文字を利用でき、数字で始まる識別子も許容される。実数値との区別のため、正規表現に位置指定子 (?=) を指定し、実数値の直後に空白文字か () を必須とした。

A.2 環境と評価器

Bind クラスは環境の実装で、識別子とその値を管理する。静的スコープを備え、外の環境への参照を持つ。

elva.scala

```
case class Bind(out: Eval, params: Seq[(Symbol, Any)]) {
  val table = scala.collection.mutable.Map(params: _*)
  def apply(s: Symbol): Any = {
    if(table.isDefinedAt(s)) table(s)
    else if(out != null) out.binds(s)
    else sys.error(s"$s not defined")
  }
  def update(s: Symbol, v: Any): Any = {table(s) = v; v}
}
```

第 5.2 節とは異なり、変数の値を更新する update メソッドを備える。続く Eval クラスは**評価器**を実装する。評価器は、LISP の構文木を受け取り、その値を求める。識別子なら値を取得し、関数なら関数適用を行う。

elva.scala

```
case class Eval(binds: Bind = Bind(null, Seq())) {
  def apply(sexp: Any): Any = sexp match {
    case s: Symbol => binds(s)
    case s: Seq[_] if s.isEmpty => s
    case s: Seq[_] => this(s.head) match {
      case f: Lambda => f(s.tail, this)
      case f: Syntax => f(s.tail, this)
      case f: System => f(s.tail, this)
      case _ => sys.error(s"raw list: $s")
    }
    case _ => sexp
  }
}
```

構文木は Any 型である。例えば、リストは Seq で、識別子は Symbol で、実数値は BigDecimal で表現する。

A.3 関数とマクロ

elva のシステム関数は、System クラスで実装する。関数は、実引数のリストと環境を受け取り、値を返す。

elva.scala

```
case class System(func: (Seq[Any], Eval) => Any) {
  def apply(args: Seq[Any], eval: Eval) = func(args, eval)
}
```

Lambda クラスは関数閉包を表す。実引数を評価して新たな環境に登録し、新たな環境で value を評価する。

elva.scala

```
class Lambda(val params: Seq[Any], val value: Any, scope: Eval) {
  def apply(args: Seq[Any], eval: Eval) = {
    Eval(Bind(scope, params.zip(args).map {
      case (p: Symbol, a) => p -> eval(a)
      case (p, a) => sys.error(s"$p=$a?")
    }))(value)
  }
}
```

Syntax クラスはマクロを表す。実引数を評価せずに新たな環境に登録し、新たな環境で value を評価する。

elva.scala

```
class Syntax(val params: Seq[Any], val value: Any, scope: Eval) {
  def apply(args: Seq[Any], eval: Eval) = eval(
    Eval(Bind(scope, params zip args map {
      case (p: Symbol, a) => p -> a
      case (p, a) => sys.error(s"$p=$a?")
    }))(value)
  )
}
```

これで、実引数を式のままマクロの内部に展開できる。その後、展開されたマクロを以前の環境で評価する。

A.4 システム関数

Scala で quote や list などのシステム関数を実装し、対話環境のトップレベルを示す環境 root に登録する。

elva.scala

```
val root = Bind(null, Seq())
```

quote は、引数を評価せずに値として返す。これは、引数を評価しない構文である**特殊形式**の代表例である。

elva.scala

```
root('quote) = System((args, eval) => args.head)
```

list は、引数を評価して、値を順番に並べたリストを返す。特に、S 式を生成する際に重要な役割を果たす。

elva.scala

```
root('list) = System((args, eval) => args.map(eval(_)))
```

他にも、car や cdr など、リストを操作する関数を定義すると良い。算術演算子も定義する。+の例を示す。

elva.scala

```
root('+) = System((args, eval) => args.map(eval(_) match {
  case real: BigDecimal => real
  case sexp => sys.error(s"non-number $sexp in $args")
}).reduce((a, b) => a + b))
```

Scala ならば、map や reduce などのメソッドを駆使して、手軽に実装できる。続いて、eq 関数を実装する。

elva.scala

```
root('eq) = System((args, eval) => eval(args(0)) == eval(args(1)))
```

if は、第 1 引数を評価して true ならば第 2 引数を、false ならば第 3 引数を評価して返す特殊形式である。

elva.scala

```
root('if) = System((args, eval) => eval(args(0)) match {
  case true  => eval(args(1))
  case false => eval(args(2))
  case sexp  => sys.error(s"not boolean: $sexp")
})
```

lambda は、関数閉包を生成する特殊形式である。第 1 引数が仮引数になり、第 2 引数が関数の本体となる。

elva.scala

```
root('lambda) = System((args, eval) => new Lambda(args.head match {
  case pars: Seq[_] if pars.forall(_.isInstanceOf[Symbol]) => pars
  case sexp => sys.error(s"not symbol list: $sexp")
}, args(1), eval))
```

syntax は、マクロを生成する特殊形式である。第 1 引数が仮引数になり、第 2 引数がマクロの本体になる。

elva.scala

```
root('syntax) = System((args, eval) => new Syntax(args.head match {
  case pars: Seq[_] if pars.forall(_.isInstanceOf[Symbol]) => pars
  case sexp => sys.error(s"not symbol list: $sexp")
}, args(1), eval))
```

set は、識別子を値で束縛する。変数の宣言だけでなく、関数閉包やマクロに名前を与える際にも利用する。

elva.scala

```
root('set) = System((args, eval) => eval(args.head) match {
  case name: Symbol => eval.binds(name) = eval(args(1))
  case sexp => sys.error("not symbol: $sexp")
})
```

A.5 処理系の完成

暗黙の型変換を利用して、任意の型の S 式を文字列に変換する機構を用意すると良い。以下に実装例を示す。

elva.scala

```
implicit class PrettyPrintableRawValueWrapper(rawValue: Any) {
  def p: String = rawValue match {
    case v: Symbol => v.name
    case v: String => '"' + v + '"'
    case v: System => s"<function>"
    case v: Seq[_] => s"(${v.map(_.p).mkString(" ")})"
    case v: Lambda => s"(lambda ${v.params.p} ${v.value.p})"
    case v: Syntax => s"(syntax ${v.params.p} ${v.value.p})"
    case v: Any => v.toString
  }
}
```

最後に、対話環境を実装する。プロンプトを表示して、S 式を読み取り、評価した結果を文字列で表示する。

elva.scala

```
def main(args: Array[String]) {
  val console = new scala.tools.jline.console.ConsoleReader
  console.setExpandEvents(false)
  console.setPrompt(s"${Console.BLUE}elva$$ ${Console.RESET}")
  while(true) try {
    println(Eval(root)(Parser.parse(console.readLine)).p)
  } catch {
    case ex: Exception => println(Console.RED + ex.getMessage)
  }
}
```

付録B　セルオートマトンの世界

本稿の前半では、スタック機械やチューリング機械を紹介したが、他にも面白い計算模型は数多く存在する。Fig. B.1 に示す**セルオートマトン**も計算模型のひとつで、ある種の規則はチューリング機械と等価でもある。

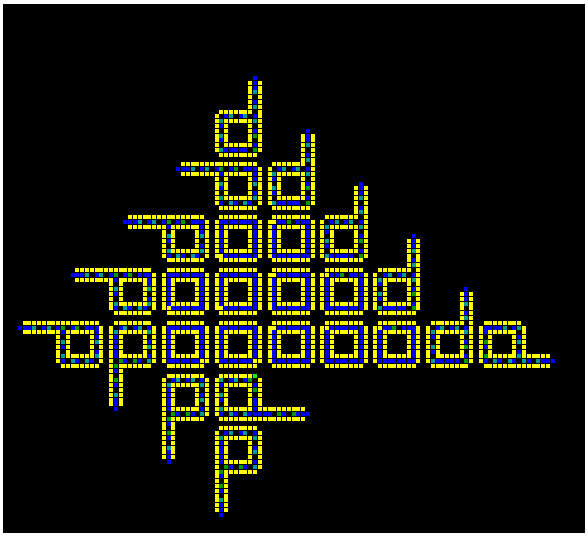
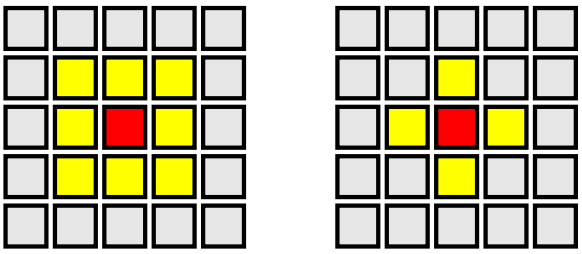


Fig. B.1: Langton's loops on cellular automata.

セルは、ある時刻における近傍セルの状態を観測し、遷移規則に従って、次の時刻における状態を決定する。状態遷移はセル間で同期する。2 次元の場合、各セルは Fig. B.2 に示す通り、8 個か 4 個のセルと隣接する。



(a) Moore's (8 cells).　(b) Neumann's (4 cells).

Fig. B.2: neighborhood cells.

セルが k 通りの状態を持つ場合、遷移規則は**ムーア近傍**なら k^8 通り、**ノイマン近傍**なら k^4 通り存在しうる。付録 B.2 節で述べる**ライフゲーム**では、セルは生か死の状態を有し、近傍の個体密度に応じて状態遷移する。

誕生　近傍に生きたセルが 3 個あれば、生命が誕生
維持　近傍に生きたセルが 2 個あれば、現状を維持
死滅　過疎や過密の場合

現実のセルオートマトンでは、格子が有限なので、上下端や左右端を接続して**円環面**(トーラス)を設定する場合もある。

B.1 セル空間の実装

以下に実装する Rule クラスは、付録 B.2 節で実装する各種の 2 次元セルオートマトンの規則の雛形である。引数 w と h はセル空間の規模を表し、states は各セルの状態を保持する。update メソッドで状態遷移する。

Rule.scala

```
abstract class Rule(val w: Int, val h: Int) {
  val states = Array.ofDim[Int](w, h)
  val buffer = Array.ofDim[Int](w, h)
  def update() {
    for(x <- 0 until w) {
    for(y <- 0 until h) {
      buffer(x)(y) = nextAt(x, y)
    }
    }
    for(x <- 0 until w) {
    for(y <- 0 until h) {
      states(x)(y) = buffer(x)(y)
    }
    }
  }
```

抽象メソッドとして、nextAt メソッドを定義する。個別のセルの遷移規則は、nextAt メソッドで記述する。

Rule.scala

```
  def nextAt(x: Int, y: Int): Int
```

本稿に掲載の実行結果は、下記の svg メソッドで描画した。実装には scala-xml ライブラリが必要である。

Rule.scala

```
  def svg(u: Int = 8) = <svg xmlns='http://www.w3.org/2000/svg'> {
    for(x <- 0 until w; y <- 0 until h) yield <rect
      x={(u * x).toString}
      y={(u * y).toString}
      width ={(u-1).toString}
      height={(u-1).toString}
      fill={"#%06x".format(states(x)(y))}
    />
  }</svg>
}
```

なお、マウスによる初期状態の入力やアニメ表示に対応するには、JavaFX の Canvas クラスが便利である。

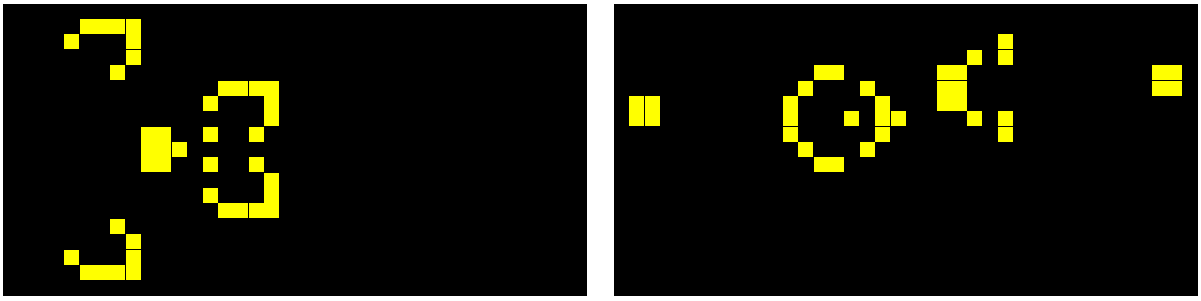
Grid.scala

```
class Grid(rule: Rule, u: Double = 8) extends Canvas {
  val gc = getGraphicsContext2D()
  for(x <- 0 until rule.w; y <- 0 until rule.h) {
    val r = rule.states(x)(y) >> 16 & 0xFF
    val g = rule.states(x)(y) >>  8 & 0xFF
    val b = rule.states(x)(y) >>  0 & 0xFF
    gc.setFill(Color.rgb(r, g, b))
    gc.fillRect(u * x, u * y, u - 1, u - 1)
  }
}
```

ただし、Rule クラスの仕様として、セルの状態は、赤と緑と青に 8bit ずつ割り当てた RGB 色空間で表す。

B.2 実際の遷移規則

Conway's Game of Life とは、**生物群集**の増殖と死滅を簡素な遷移規則で表現したセルオートマトンである。



(a) puffer train. (b) glider gun.

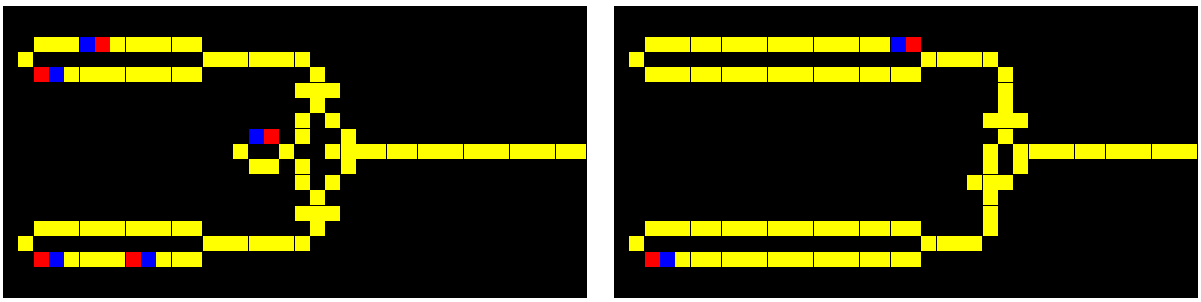
Fig. B.3: infinite growth pattern.

`Life` クラスに実装する。Fig. B.3 の**シュシュポッポ列車**や**グライダー銃**が際限なく繁殖する様子が楽しめる。

Life.scala

```
class Life(w: Int, h: Int) extends Rule(w, h) {
  val (l, d) = (0xFFFF00, 0x000000)
  override def nextAt(x: Int, y: Int): Int = {
    var count = 0
    def nx(off: Int) = (x + off + w) % w
    def ny(off: Int) = (y + off + h) % h
    count += states(nx(-1))(ny(-1)) / l
    count += states(nx( 0))(ny(-1)) / l
    count += states(nx(+1))(ny(-1)) / l
    count += states(nx(-1))(ny( 0)) / l
    count += states(nx(+1))(ny( 0)) / l
    count += states(nx(-1))(ny(+1)) / l
    count += states(nx( 0))(ny(+1)) / l
    count += states(nx(+1))(ny(+1)) / l
    if(count == 2) states(x)(y) else if(count == 3) l else d
  }
}
```

Silverman が考案した**ワイヤワールド**は、電子回路を模したセルオートマトンである。Fig. B.4 に例を示す。



(a) NAND gate. (b) SR latch.

Fig. B.4: logic circuit elements.

黒のセルは**絶縁体**を、黄のセルは**電導体**を、赤と青のセルは**電子**の頭と尾を表す。`Wire` クラスに実装する。

Wire.scala

```
class Wire(w: Int, h: Int) extends Rule(w, h) {
  val (b, c) = (0x000000, 0xFFFF00)
  val (l, f) = (0xFF0000, 0x0000FF)
  override def nextAt(x: Int, y: Int): Int = {
    var count = 0
    def nx(off: Int) = (x + off + w) % w
    def ny(off: Int) = (y + off + h) % h
    if(states(nx(-1))(ny(-1)) == l) count += 1
    if(states(nx( 0))(ny(-1)) == l) count += 1
    if(states(nx(+1))(ny(-1)) == l) count += 1
    if(states(nx(-1))(ny( 0)) == l) count += 1
    if(states(nx(+1))(ny( 0)) == l) count += 1
    if(states(nx(-1))(ny(+1)) == l) count += 1
    if(states(nx( 0))(ny(+1)) == l) count += 1
    if(states(nx(+1))(ny(+1)) == l) count += 1
    if(states(x)(y) == b) return b
    if(states(x)(y) == l) return f
    if(states(x)(y) == f) return c
    return if(count == 1 || count == 2) l else c
  }
}
```

Scherer は、論理回路の実装例を多岐にわたり紹介している[1]。例えば、Fig. B.5 は、カウンタの実装である。

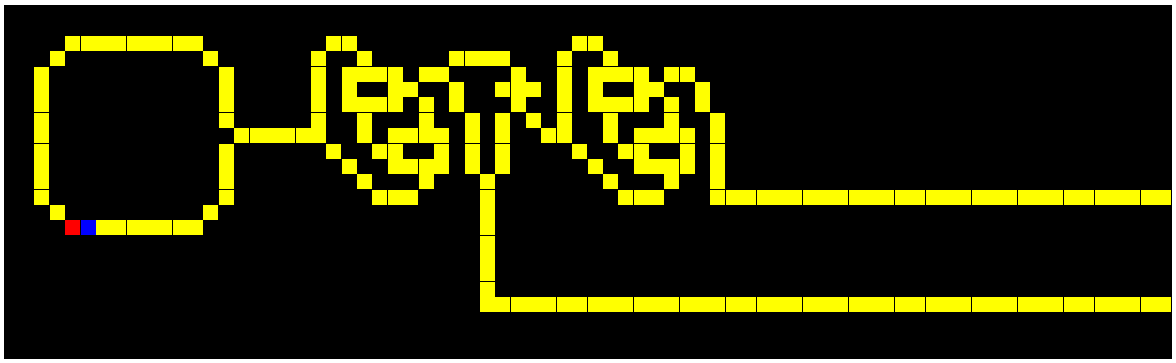


Fig. B.5: binary counter.

Fig. B.6 は、直列加算器である。左側が入力で、右側が出力である。入出力とも最下位ビットを先頭とする。

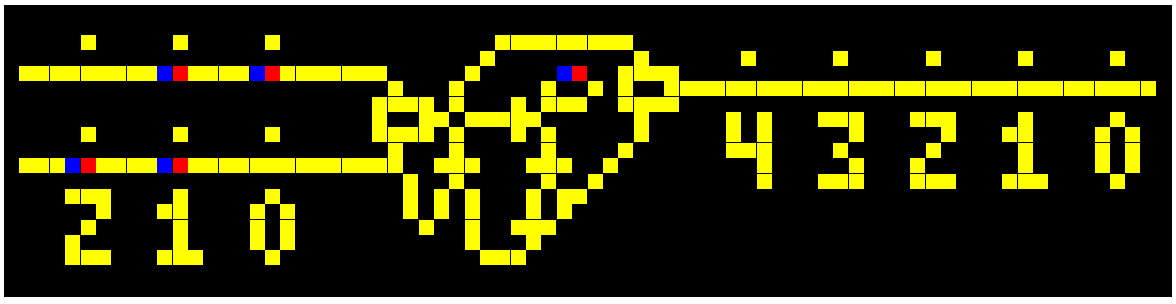


Fig. B.6: binary adder.

絶縁体は、常にその状態を維持する。電導体は 1 個か 2 個の電子の頭が近傍にあれば、電子の頭に変化する。また、電子の頭と尾は、次の時刻で電子の尾と電導体に変化する。これにより、電子は進行方向に移動する。

[1] http://karlscherer.com/Wireworld.html