# 2. Lisp プログラミング入門

## 概要

Lisp は記号の構造的な表現である S 式を操作するインタープリタ方式を基調とするプログラミング言語である. ここでは, 思考のツールとしての Lisp を強調した解説を行う.

### 1. Lisp のしくみ

Lisp で中心となるのは, S 式(Symbolic Expression)と呼ばれる記号の構造的な表現である. Lisp ユーザはインタープリタを使って, S 式を作り出したり, 変形したり, 探索したりできる.

Lisp の特色の一つに, リフレクションがある. S 式はデータであるが, プログラムを表現するためにも用いられる. プログラムを表現した S 式を eval という関数を通して実行すると, プログラムとして解釈実行される. Lisp インタープリタは, read-eval-print ルーチンを繰り返す. すなわち, 入力として S 式であらわされたプログラムとデータを受け取り, その解釈実行を行って, 結果を返す.

### 2. Lisp インタープリタの動かし方

では, 早速, Lisp インタープリタを動かしてみよう. Lisp インタープリタ[1]を起動すると, ユーザとやりとりするためのプロンプトが現れる.

```
GCL (GNU Common Lisp)  2.6.7 ANSI    Dec 27 2007 19:55:59
Source License: LGPL(gcl,gmp), GPL(unexec,bfd)
Binary License:  GPL due to GPL'ed components: (UNEXEC)
Modifications of this banner must retain notice of a compatible license
Dedicated to the memory of W. Schelter

Use (help) to get some basic information on how to use GCL.

>
```

最後の>のあとに, ユーザが S 式を入力すると, その評価が行われ, その結果が表示される. 停止するためには, (bye)という S 式を入力するか, もっと乱暴にウィンドウを閉じてもよい.

### 2.1 「電卓」としての Lisp

Lisp インタープリタの最も単純な使い方は電卓である. 例えば, ユーザが`(expt 3 4)`という S 式を入力したときのやりとりは,

```
>(expt 3 4)   … 波線部分(プラス Enter)はユーザからの入力であることを示す.

81 … Lisp からの出力. 上記の S 式を「評価」した結果が 81 という S 式で表されていることを示す.

>             … 次の入力待ちであることを示すプロンプト
```

[1] 以下では, Gnu Common Lisp による.

となる． これは，Lisp インタープリタがユーザの入力(read)した S 式`(expt 3 4)`の表す計算式 $\mathrm{expt}(3,4) \equiv 3^4$ の値を計算（以下では，「評価」(eval[uate])と呼ぶ）した結果である 81 を `81` という S 式で表示(print)したことを示している．

最も原始的な S 式は，アトムと呼ばれるものであり，

```
3
4
81
34.015
x
x13g6
“a string”
```

などがある．このなかで，`x` や `x13g6` はリテラルアトムと呼ばれ，評価された場合は変数として扱われる[2]．それ以外は定数であり，それを評価するとそれ自体になる[3]．

```
>81

81                ⋯ 81 を評価すると，81 になる．
```

リテラルアトムへの値の対応づけを行うためには，`(setq` 〈リテラルアトム〉〈S 式〉`)` という S 式を用いる．

```
>(setq x12 100)

100               ⋯ リテラルアトム x12 に S 式 100 を評価した 100 という値が対応づけられた．
                    (setq x12 100) という S 式自体を評価した値も 100 である．

>x12

100               ⋯ リテラルアトム x12 を評価すると，100 になる．
```

アトム以外の S 式は，リスト形式の S 式は括弧に挟まれ，個々の要素は 1 つまたはそれ以上の空白によって区切られている．

(〈1 番目の S 式〉〈2 番目の S 式〉⋯ 〈*n* 番目の S 式〉)

たとえば，次のものはすべて S 式である．

```
(1 2 3)
(A (B C) (1 2 3) (0.1 (0.001 X)))
((((((A B) C) D) E) F) G)
```

[2] リテラルアトムの場合は，大文字と小文字の区別はされない．
[3] 真を表す `T` と偽および空リストを表す `NIL` は例外的に定数として扱われる．

これらは単なるデータであるが，先にみたように，Lisp インタープリタに与えられて，評価の対象となるときは，式としての形式と意味を持つように作られなければならない．一般に，Lisp では，
$f(a_1,\cdots,a_n)$という関数は，

(〈$f$ の S 式表現〉〈$a_1$ の S 式表現〉 … 〈$a_n$ の S 式表現〉)

という格好の S 式で表される．
加算 $a_1+\cdots+a_n$ と乗算 $a_1\times\cdots\times a_n$ に対応する S 式はやや例外的であり，いずれも不定個の引数を取る関数 $+(a_1,\cdots,a_n)$，$*(a_1,\cdots,a_n)$ として次のように S 式表現される．

(+〈$a_1$ の S 式表現〉 … 〈$a_n$ の S 式表現〉) … $a_1+\cdots+a_n$ の S 式表現
(*〈$a_1$ の S 式表現〉 … 〈$a_n$ の S 式表現〉) … $a_1\times\cdots\times a_n$ の S 式表現

これらを組み合わせると，次のような S 式表現を構成できる．

(+ 1 2 3 4 5) … $1+2+3+4+5$
(+ (* 2 3) (* 4 (- 5 6))) … $2\times3+4\times(5-6)$
(sqrt (+ (expt 3 2) (expt 4 2))) … $\sqrt{3^2+4^2}$

以上を組み合わせると次のような処理ができる．

```
>(setq x 3)

3

>(setq y 4)

4

>(sqrt (+ (expt x 2) (expt y 2)))

5.0

>(setq z (/ (+ x y) 2))

7/2

>(* z 2)

7
```

### 2.2 関数を定義する

関数の定義には

(defun〈関数名を表すリテラルアトム〉〈引数リスト〉〈S 式〉 … 〈S 式〉)

をもちいる. この関数は与えられた引数リストへの値の対応づけに基づいて〈S 式〉 … 〈S 式〉を順に評価し, 最後の〈S 式〉の値をその関数の値とする. 普通の場合は, 〈S 式〉は 1 個である. 複数の〈S 式〉を指定できるようにした理由は割愛する. 例えば,

```
>(defun average (x y) (/ (+ x y) 2))

AVERAGE   … 上の S 式を評価したときの値はリテラルアトム average となる(重要ではない).

>(average 4 8)

6
```

### 2.3 条件分岐

他のプログラミング言語と同様に Lisp にも様々な条件分岐の方法がある. 取りあえず, 次の 2 つを押さえておこう. 第一番目は,

```
(if 〈S 式 1〉 〈S 式 2〉 〈S 式 3〉)
```

〈S 式 1〉を評価した結果が非 NIL であれば, 〈S 式 2〉を評価した結果をこの式の値とし, 〈S 式 1〉を評価した結果が NIL であれば, 〈S 式 2〉を評価した結果をこの式の値とする. 例えば,

```
>(setq x -5)

-5

>(if (> x 0) 1 -1)

-1
```

より多くの条件分岐を一度に計算するのが cond[itional]である.

```
(cond (〈S 式 1A〉 〈S 式 1B〉) … (〈S 式 nA〉 〈S 式 nB〉))
```

〈S 式 $i_A$〉を $i$=1 から順に評価し, 初めて非 NIL になった $i$ に対して, 〈S 式 $i_B$〉を評価した結果をこの式の値とする. 例えば,

```
>(setq x -5)

-5

>(cond ((= x -7) 0) (= x -6) 1) (= x -5) 2) (t 0))

2
```

となる.

Lisp は帰納関数論から生まれているので，再帰的な定義は自在にできる．$n$ の階乗を計算する関数 `factorial` を定義してみよう．数学的には，

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{factorial}(n-1) & \text{otherwise} \end{cases}$$

であるから，関数 `factorial` の中心部分は，

```
(cond ((= n 0) 1) (T (* n (factorial (1- n)))))
```

であることがわかる．このモチーフに基づいて Lisp インタープリタを動かせば次のようになる．

```
>(defun factorial (n)
    (cond ((= n 0) 1) (T (* n (factorial (1- n))))))

FACTORIAL

>(factorial 5)

120

>(factorial 40)

815915283247897734345611269596115894272000000000
```

関数`(trace x)`は，$x$ で与えられた関数の動きをトレース（追跡）する．

```
>(trace factorial)

(FACTORIAL)

>(factorial 6)

  1> (FACTORIAL 6)
    2> (FACTORIAL 5)
      3> (FACTORIAL 4)
        4> (FACTORIAL 3)
          5> (FACTORIAL 2)
            6> (FACTORIAL 1)
              7> (FACTORIAL 0)
              <7 (FACTORIAL 1)
            <6 (FACTORIAL 1)
          <5 (FACTORIAL 2)
        <4 (FACTORIAL 6)
      <3 (FACTORIAL 24)
    <2 (FACTORIAL 120)
  <1 (FACTORIAL 720)
720
```

### 2.4 S 式の基本操作

Lisp という名前は List Processor（リスト処理システム）に由来している．リスト構造を自在に扱えるところが Lisp の強みである．歴史的な理由もあり，S 式の操作にはやや微妙なところもあるが，はじめのうちはなるべくそこに立ち入らないようにしておきたい．

最も基本的なものは，変数への S 式のセット，S 式の組み立て，S 式の各部へのアクセスである．S 式はデータであるので，変数 x に S 式(A (B C) (D E))を対応づけようとして，(setq x 5)の時と同様に

```
>(setq x (A (B C) (D E)))
```

と入力すると，Lisp インタープリタから次のようなエラーメッセージがたちどころに返される．

```
Error in SETQ [or a callee]: The function A is undefined.

Fast links are on: do (use-fast-links nil) for debugging
Broken at SETQ.  Type :H for Help.
 1 (Abort) Return to top level.
dbl:>>
```

これは Lisp インタープリタが，(A (B C) (D E))を式だと思って評価しようとして，関数 A の定義を探そうとして見つからなかったため，エラーの起きたところでブレークループに入ったことを示している．

Lisp インタープリタに S 式 $x$ をデータとして渡すためには，

```
(quote x)
```

あるいは，

```
'x
```

という表現を使う．

上の例の場合は，

```
>(setq x '(A (B C) (D E)))

(A (B C) (D E))

>(length x)

3
```

となる．

リスト構造を作り上げるための基本的な関数は list と append である．すなわち，

(list 〈$a_1$ の S 式表現〉 … 〈$a_n$ の S 式表現〉)

⇒ (〈$a_1$ の S 式表現の評価結果〉 … 〈$a_n$ の S 式表現の評価結果〉)

となる.

```
>(list 1 2 (1+ 2) (* 2 2) (sqrt (+ (expt 3 2) (expt 4 2))))

(1 2 3 4 5)
```

となる.

(append 〈$a_1$ の S 式表現〉 … 〈$a_n$ の S 式表現〉)
⇒ 〈$a_1$ の S 式表現の評価結果〉 … 〈$a_n$ の S 式表現の評価結果〉を一つのリストにつないだもの

例

```
>(append '(1 2) '((3) (4)) (list 'a '(b c)))

(1 3 (3) (4) a (b c))
```

このようにして作られたリストから要素を取り出したり, 型判定を行ったりするには, 次のような関数を用いる.

(atom 〈S 式〉)
⇒ 〈S 式〉がアトムならば T, さもなければ NIL を返す.

例えば,

```
>(atom 3.14)

T

>(atom (list 'a '(b c)))

NIL
```

(null 〈S 式〉)
⇒ 〈S 式〉が NIL ならば T, さもなければ T を返す. NIL と要素をもたないリスト()は同一であることに注意.

例えば,

```
>(null NIL)

T

>(null '())
```

```
T
```

(first 〈S 式〉) (second 〈S 式〉) … (tenth 〈S 式〉)
⇒ それぞれ, 〈S 式〉(リストでなければならない)の 1 番目～10 番目のアイテムを返す.

例えば,

```
>(sixth '(1 2 3 4 5 6 7 8 9 0))

6
```

(nth *n x*)
⇒ リスト x の前から n+1 番目の要素を返す.

例えば,

```
>(nth 5 '(1 2 3 4 5 6 7 8 9 0))

6
```

例えば, (fifth *s*)は(nth 4 *s*)と同じであり, (nth 5 *s*)とは異なるので注意.

(last *x*)
⇒ リスト *x* の最後の要素を返す.

例えば,

```
>(last '(1 2 3 4 5 6 7 8 9 0))

0
```

### 2.5 リストとドット対

前項が示唆しているように, Lisp におけるリストは左右非対称の構造になっている. それは, 例えば,

```
(a b c d e)
```

というリスト構造は, 実は, ドット対

```
(a . (b . (c . (d . (e . NIL)))))
```

のマクロ表現であることに対応している. これに応じて,

(cons *x y*)
⇒ *x* と *y* から成るドット対(*x* . *y*)を作る.

例えば,

```
>(cons 'a 'b)

(A . B)

>(cons 'a '(b c))
```

(A B C) … なぜならば, 上記の結果は, (A . (B . (C . NIL)))であり, これは(A B C)に等しい.

`($a_1$ … $a_n$)
⇒ $a_1$ … $a_n$から成るリストを構成する. ここで $a_i$ は〈S 式〉または ,〈S 式〉という格好をしている. 後者は S 式の前にコンマがついている. $a_i$ が〈S 式〉のときは, そのデータをそのまま使用する. $a_i$ が ,〈S 式〉という格好をしているときは, その S 式を評価した値を用いる.

例えば,

```
>`(= (+ 1 2 3) ,(+ 1 2 3))

(= (+ 1 2 3) 6)
```

となる. 非常に便利なマクロである.

(car $s$)
⇒$s$ はドット対($x$ . $y$)でなければならない. このとき(car $s$)はドット対第 1 番目の要素 $x$ を返す.

例えば,

```
>(car '(a . b))

A

>(car '(1 2 3 4 5))

1
```

(car $s$)
⇒$s$ はドット対($x$ . $y$)でなければならない. このとき(car $s$)はドット対第 2 番目の要素 $y$ を返す.

例えば,

```
>(cdr '(a . b))

B

>(cdr '(1 2 3))
```

(2 3) … 引数の値は, (1 . (2 . (3 . NIL)))であるから cdr 値は(2 . (3 . NIL))に等しい.

練習問題 与えられた S 式 $s$ の中にアトム $x$ があれば T, さもなければ NIL を返すプログラム (INCLUDED? $x$ $s$)を定義せよ.

練習問題 与えられた S 式 $s$ に含まれるアトム $x$ の個数を数えるプログラム(HOW-MANY $x$ $s$)を定義せよ.

## 2.6 便利なデータ構造

いくつかのデータ構造があり, プログラミングの補助として使える.

(make-array (list $i_1$ … $i_n$))
⇒第 1 次元が 0～$i_1$-1, …, 第 $n$ 次元が 0～$i_n$-1 までのインデクスをもつ $n$ 次元配列オブジェクトを生成する.

(aref $a$ $x_1$ … $x_n$)
⇒$n$ 次元配列オブジェクト $a$ から, 第 1 次元のインデクスが $x_1$, …, 第 $n$ 次元のインデクスが $i_n$ の要素を取り出す.

(setf (aref $a$ $x_1$ … $x_n$) $v$)
⇒$n$ 次元配列オブジェクト $a$ から, 第 1 次元のインデクスが $x_1$, …, 第 $n$ 次元のインデクスが $i_n$ の要素を $v$ にセットする.

例

```
>(setq x (make-array '(4 4 4)))

#3A(((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)
    (NIL NIL NIL NIL))
   ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)
    (NIL NIL NIL NIL))
   ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)
    (NIL NIL NIL NIL))
   ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)
    (NIL NIL NIL NIL)))

>(aref x 1 2 3)

NIL

>(setf (aref x 1 2 3) 'a)

A

>x

#3A(((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)
```

```
    (NIL NIL NIL NIL))
   ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL A)
    (NIL NIL NIL NIL))
   ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)
    (NIL NIL NIL NIL))
   ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)
    (NIL NIL NIL NIL)))

>(aref x 1 2 3)

A
```

(assoc $x$ $a$)
⇒ ((k1 . v1) … (kn . vn))と言う格好をした連想リスト $a$ から $k_i$=$x$ となる最初の $i$ を探し，($k_i$ . $v_i$)を返す.

例

```
>(assoc 'i '((she . kanojo) (he . kare) (i . watashi)))

(I . WATASHI)

>(assoc 'thou '((she . kanojo) (he . kare) (i . watashi)))

NIL
```

## 2.7 おいもプログラム

Lisp は関数型プログラミング言語と呼ばれているが，思考のツールとしてはそれだけでは使いづらいので，普通の副作用型のプログラミング機構も用意されている.

(let* (($i_1$ $v_1$) … ($i_m$ $v_m$)) $s_1$ … $s_n$)
⇒局所変数 $i_1$ … $i_m$ に $v_1$ … $v_m$ の各々を評価した値を順に対応づけたうえで，S 式 $s_1$ … $s_n$ を順に評価し，最後に評価した $s_n$ の値をこの式の値とする.

例

```
>(let* ((x 2) (y 3) (z nil)) (setq z (+ x y)) `(,x + ,y = ,z))

(2 + 3 = 5)
```

(loop $s_1$ … $s_n$)
⇒S 式 $s_1$ … $s_n$ を順に評価しつづける. そのなかで(return $v$)が実行されると，$v$ の評価した値をもってこの式の実行は終わる.

例

```
>(let* ((x 1)) (loop (if (= x 100) (return 99) nil)
```

```
                        (setq x (1+ x))))

99
```

(dotimes ($x$ $n$) $s_1$ … $s_n$)
⇒変数 $x$ の値を 0 から $n$ を評価した値まで順に増加させていき，それぞれの値に対して， $s_1$ … $s_n$ を順に評価する．ただし，(return $v$)が実行されると，$v$ の評価した値をもってこの式の実行は終わる．

例

```
>(dotimes (x (* 5 2)) (print x))

0 … (print 0)が実行された副作用
1 … (print 1)が実行された副作用
2 … (print 2)が実行された副作用
3 … (print 3)が実行された副作用
4 … (print 4)が実行された副作用
5 … (print 5)が実行された副作用
6 … (print 6)が実行された副作用
7 … (print 7)が実行された副作用
8 … (print 8)が実行された副作用
9 … (print 9)が実行された副作用
NIL … 式全体の評価結果

>(dotimes (x (* 5 2)) (print x) (if (> x 3) (return)))

0
1
2
3
4
NIL
```

(dolist ($x$ $s$) $s_1$ … $s_n$)
⇒S 式 $s$ を評価して得られたリスト($a_1$ … $a_n$)の要素 $a_i$ を順に変数 $x$ に対応づけた上で， $s_1$ … $s_n$ を順に評価する．ただし，(return $v$)が実行されると，$v$ の評価した値をもってこの式の実行は終わる．

例

```
>(dolist (x '(1 2 3 4 5)) (print x))

1
2
3
4
```

```
5
NIL

>(dolist (x '(1 2 3 4 5)) (print x) (if (= x 3) (return 'three)))

1
2
3
THREE
```

### 2.8 おいもプログラミング

Gnu Common Lisp には Break Package があり，エディタで少しプログラムして，実行し，エラーの発生した環境（変数の束縛環境）で原因を調べるばかりか，プログラミングの訂正や新規のプログラミングまでしてしまえる．

実行系を終了すると，折角与えた定義は消えてしまう．これでは困るので，普通のプログラミングでは，定義式をファイルにあらかじめ与えておいて，読み込むという手順が踏まれる．このとき，Common Lisp 特有の面白い性質は，エラーが生じない限り，定義式を与えたファイルを読ませることはコンソールの入力を（コマンド実行結果に関わらず）与えることと全く同じであることだ．たとえば，My documents/gcl/の下に，sample1.lsp というテキストファイルを作成し，そのなかに先に与えた (defun factorial …)を書いておいて，gcl から，それを読み込めばよい．

上でデフォールトパス名として，“/Users/Nishida/Documents/gcl”を与えて，いちいちこの部分を繰り返さなくてもよいようにするためには変数*default-pathname-defaults*に当該ファイルのあるディレクトリパスをセットしておく．例えば，

```
(setq *default-pathname-defaults* “C:/Users/Nishida/Documents/gcl/”)
```

を gcl 立ち上げ時に実行しておくと，

```
(load “sample1”)
```

によって， C:/Users/Nishida/Documents/gcl/sample1.lsp にある lisp 式が read-eval-print によって実行される(.lsp は省略可能)．sample1.lsp には lisp インタープリタが実行できる式であれば何を書いても構わない．通常は，(defun factorial …)といった定義式を書いておいて，定義式が実行され，定義が行われるようにする．

さらに発展させると，プログラムを部品から作るのは正反対に，トップレベルから順に作っていく「トンネル掘りプログラミング」ができてしまう．意図的にエラーを引き起こす関数として，(break *s*)がある．

```
(break s)
```

⇒この式が評価された時点で S 式 *s* の評価値が表示され，ブレークパッケージに入る．ブレークパッケージでは，

〈S 式〉 … その環境で〈S 式〉を評価する
:r … 処理を続行する
:q … トップレベルに帰る
:b … どの計算過程でエラーになったか，呼び出し関係を表示する

## 3. 例 1: チューリングマシンのシミュレータ

概ねプログラムの外部仕様を決めた後，内部で使用するデータ構造のデザイン，アルゴリズムのデザイン，実装，検証の順序で進める．大体こんなものだろうと大まかなところを決め，それを詳細化すればよい．データ構造をきちんと定義すればかなり効率的なプロトタイピングができる．

### 3.1 外部仕様

ここで，実現するチューリングマシンシミュレータは，チューリングマシンの仕様，ヘッドの現在位置をマークした入力テープを入力とし，シミュレーション過程を逐次出力するプログラムとする．入出力を気の利いたものにしようとすると，プログラムも複雑になるので，ここでは，プログラミングの平易さに重点を置き，ユーザが指定したステップ数までシミュレーションするプログラムとする．

#### 3.1.1 入力テープ表現

入力テープの表現は，簡単である．ここでは，入力テープ上の左右の無限の空白列を切り落とした記号の並びを S 式で表すこととし，ヘッドが初めに置かれる位置を*で示すこととする（ヘッドはそのすぐ右隣のマス目をスキャンしている）．例えば，

(1 2 3 * 4 5 6 7)

は，ヘッドが現在スキャンしているマス目の値が 4 であり，その左のマス目にはヘッドのある位置から順に，3, 2, 1 という値が入っており，右には，4, 5, 6, 7 という値が入っていること，および，その卒側のマス目はすべて空白であることを示す．

#### 3.1.2 状態遷移機械の動作表の表現

状態遷移機械の動作表を表現するためには少し複雑な S 式が必要である．その一例を図 1 に示す．これはチューリングマシンの数学的な表現に近いものであるが，これが唯一の表現方法ではなく，プログラマーが自由に設計すればよい．

### 3.2 主要な内部変数，その意味，データ構造

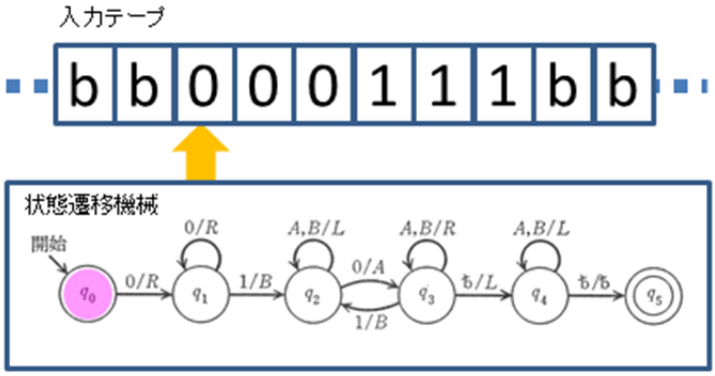
チューリングマシンの各ステップは，そのときの入力テープのマス目の内容，スキャン位置，状態遷移機械の状態によって表現される．ここで示すプログラムでは，

current-state: 現在の状態（リテラルアトム）
left-stack: 入力テープのヘッドよりも左にあるマス目の内容（右→左，リスト）
current-cell: 現在ヘッドがスキャンしているマス目の内容（リテラルアトム）
right-stack: 入力テープのヘッドよりも右にあるマス目の内容（左→右，リスト）

の 4 つの変数を用いてチューリングマシンの実行状態を管理している．また，左右に無限に続く空白については，非空白記号およびそれらに挟まれた空白記号（これは，_というリテラルアトムで表す）だけをリストに載せておくこととする．

### 3.3 シミュレーションアルゴリズム

シミュレータの本体を構成するシミュレーションアルゴリズムについては，与えられた入力を内部データ構造に変換したのちに，プログラムの核心部分，すなわち，「『チューリングマシンのヘッドがスキャンしている入力テープ上の記号と，状態遷移機械の状態が与えられたとき，状態遷移機械




(a) チューリングマシンの例

```
((start q0)        … 初期状態が q0 である
 (terminal q5)     … 停止状態は q5 である.
 (states           … 以下では, 各状態における状態遷移規則を表す
  (q0 (0 (Right) q1))   … 状態 q0 で, 0 を見たらヘッドを右に移動し, 状態 q1 に遷移
  (q1 (0 (Right) q1)    … 状態 q1 で, 0 を見たらヘッドを右に移動し, 状態 q1 に遷移
      (1 B q2))         … 状態 q1 で, 1 を見たら B と書き換え, 状態 q2 に遷移
  (q2 (A (Left) q2)     … 状態 q2 で, A を見たらヘッドを左に移動し, 状態 q2 に遷移
      (B (Left) q2)     … 状態 q2 で, B を見たら B と書き換え, 状態 q2 に遷移
      (0 A q3))         … 状態 q2 で, 0 を見たら A と書き換え, 状態 q3 に遷移
  (q3 (1 B q2)          … 状態 q3 で, 1 を見たら B と書き換え, 状態 q2 に遷移
      (A (Right) q3)    … 状態 q3 で, A を見たらヘッドを右に移動し, 状態 q3 に遷移
      (B (Right) q3)    … 状態 q3 で, B を見たらヘッドを右に移動し, 状態 q3 に遷移
      (_ (Left) q4))    … 状態 q3 で, 空白_を見たらヘッドを左に移動し, 状態 q4 に遷移
  (q4 (A (Left) q4)     … 状態 q4 で, A を見たらヘッドを左に移動し, 状態 q4 に遷移
      (B (Left) q4)     … 状態 q4 で, B を見たらヘッドを左に移動し, 状態 q4 に遷移
      (_ _ q5))))       … 状態 q4 で, 空白_を見たら, 状態 q5 に遷移
```

(b) 上の示したチューリングマシンの状態遷移機械の S 式表現

図 1: チューリングマシンの状態遷移機械の S 式表示例

の動作表に基づいて入力テープに対する操作, ヘッドの動き, 次状態を決定する』という作業を停止状態になるまで繰り返す」を実行すればよい.

プログラムを書き始める前に, 手作業でこの作業をたどってみよう. シミュレーションは, 入力テープに 000111 という記号が置かれ, チューリングマシンは初期状態 q0 にあり, ヘッドは左端のゼロをスキャンしているという状態から始まる. これを下のように図示する.

```
000111
↑
q0
```

動作表によれば, このとき, ヘッドを 1 コマ右に進め, 状態 q1 に遷移するので, 次は

```
000111
 ↑
 q1
```

となる. 同様の作業を進めていくと,

```
000B11        0AABB1        _AAABBB
   ↑     …      ↑     …     ↑
  q2            q3          q5
```

となり, 停止状態 q5 に入って停止することがわかる. q5 は予め指定された停止状態であるので, これは「受理」停止, すなわち, このチューリングマシンが入力テープ上にある記号列は $0^n1^n$ というパターンをしていることを知らせてくれたことになる.

具体的なイメージができたら後は楽にプログラミングできるはずだ.

## 参考文献

[Hopcroft 2002] ホップクロフト, モトワニ, ウルマン. オートマトン・言語理論・計算論 I, II 第 2 版, サイエンス社, 原著 2001 年, 翻訳版 2003 年.