LISP-STATの利用方法 (v00-19)

上條哲男 (上智大学経済学部)

2004年4月5日

1 はじめに

コンピュータ・ソフトウエアであるXLISP-STATの利用方法について説明する。

ミネソタ大学のSchool of StatisticsのLuke TierneyがLISP-STATを考案し、コンピュータ・ソフトウエアであるXLISP-STATを開発した(LISP-STATについては、竹村(1997), 垂水(1999), および Tierney(1990)などを参照)。XLISP-STATは、LISP言語の方言の1つであるXLISP言語を使用している(LISPについては、栗原(1993), Steele(1990), 竹内(1986), Winston and Horn(1989), および湯浅, 萩谷(1986)などを参照)。

2では、XLISP-STATの起動手順および終了手順を、3では、XLISP-STATの基本的なインタプリタ機能を、4では、XLISP-STATの追加的なインタプリタ機能を、5では、データを直接入力し、保存し、ロードする方法を、6では、エディタによってデータ ファイルを作成し、それをロードする方法を、7では、XLISP-STAT自身のデータ生成機能を、8では、データ編集機能を、そして9では、LISP-STATによるプログラミングを説明する。

[問題1]

3.5インチのフロッピーデスクを2枚準備し、フロッピーデスクにシールを貼り、「No.1：オリジナル」、そして「No.2：バックアップ」と記入すること。それぞれのフロッピーデスクに、クラス内番号（個人で学習する場合には、クラス内番号を1とする）および氏名を記入すること。なお、「No.3：提出用」のフロッピーデスクを準備する必要がある場合もある。No.1のフロッピーデスクにつぎのようなフォルダ

LS-tk

を作成すること。ここで、LS は LISP-STAT を示している。tk は著者である上條哲男の名前と姓のローマ字表記である tetsuo kamijo のイニシャルである。tk の部分は、各人のイニシャルを用いなさい（以下でも同様である）。つぎに、フォルダ LS-tk の下に、つぎのようなフォルダ（サブフォルダ）

drib2004

を作成すること。ここで、drib は dribble を、2004 は西暦 2004 年を示しており、西暦 2005 年の場合は 2005 とすること。なお、以下では、このサブフォルダを、それに対するフォルダを前置して、

LS-tk¥drib2004

と表示する場合がある。No.1 のフロッピーデスクに、プログラムが保存されているフロッピーデスクのフォルダ

book-stat

をコピーすること。No.1 のフロッピーデスク全体を、バックアップ用である No.2 のフロッピーデスクに、随時コピーすること。

[問題 2]　「サイコロ投げの実験」
　サイコロ（正六面体のサイコロ）を 60 回投げるというサイコロ投げの実験を実行し、出た目の数を記録しなさい。

[問題 2 の解答]
　以下の説明において、この実験データをそれに sr0160tk というデータ名を付けて使用することにする。ここで、

sr:　s: サイコロ、r: red（赤）
0160　実験: 第 1 回から第 60 回まで
tk:　t: tetsuo, k: kamijo

である（tk の部分は各人のイニシャルを使用すること）。なお、sr0160tk のデータは、

6 2 1 3 5 5 4 5 2 3
2 6 4 5 6 5 5 4 5 6
1 1 6 1 5 4 5 4 5 5
3 1 1 2 6 4 2 6 4 3
3 5 1 3 1 1 4 4 1 5
3 2 6 2 3 6 1 1 2 1

である（以下の問題においては、各人のデータを使用すること）。

2 XLISP-STAT の起動手順および終了手順

XLISP-STAT の起動手順および終了手順を MS-Windows 95(Windows 95 と略す場合もある) を前提に説明する。

2.1 XLISP-STAT の起動手順

(1) XLISP-STAT が、スタートメニュー内のプログラムメニューに登録されている場合、(2) デスクトップに XLISP-STAT のショートカットが作成されている場合、あるいは (3)XLISP-STAT がプログラムメニューに登録されてなく、XLISP-STAT のショートカットも作成されていない場合とでは XLISP-STAT の起動方法が異なる。

(1) XLISP-STAT が、スタートメニュー内のプログラムメニューに登録されている場合

(a) Windows 95 のデスクトップにおいて、スタートボタンをクリックする。
(b) スタートメニューにおいて「プログラム」までマウスポインタを移動する。
(c) 「XLISP-STAT」をクリックする。
(d) XLISP-STAT が起動され、Listener が開く。起動画面には、つぎのようなメッセージおよび促進記号 (prompt, プロンプト) >が表示される。

```
XLISP-PLUS version 3.0
Portions Copyright (c) 1988, by David Betz.
Modified by Thomas Almy and others.
XLISP-STAT Release 3.50 (Beta).
Copyright (c) 1989-1994, by Luke Tierney.
Initialization may take a moment.

>
```

(2) デスクトップにショートカットを作成した場合

(a) ショートカットをダブルクリックする。
(b) XLISP-STAT が起動され、(1) の「XLISP-STAT が、スタートメニュー内のプログラムメニューに登録されている場合」と同じメッセージと促進記号が表示される。

(3) XLISP-STAT がプログラムメニューに登録されてなく、XLISP-STAT のショートカットも作成されていない場合

(a) Windows 95 のデスクトップにおいて、マイコンピュータをダブルクリックする。
(b) XLISP-STAT がインストールされているドライブ (例: Windows 95(C:)) をダブルクリックする。
(c) XLISP-STAT がインストールされているフォルダ (例: xlispsta) をダブルクリックする。
(d) 実行するためのファイル (例: LISP-STAT, wxls32.exe) をダブルクリックする。
(e) XLISP-STAT が起動され、(1) の「XLISP-STAT が、スタートメニュー内のプログラムメニューに登録されている場合」と同じメッセージと促進記号が表示される。

2.2 XLISP-STAT の終了手順

XLISP-STAT を終了するには、促進記号の後に、(exit) を入力し、「Enter」キーを押す。その結果、Windows 95 の場合には、デスクトップに戻る。すなわち、

```
> (exit)
```

とすると、XLISP-STAT が終了し、Windows 95 デスクトップの画面が現れる

3 XLISP-STAT の基本的なインタプリタ機能

XLISP-STAT システムの実行は、利用者と LISP インタプリタとの間の会話形式でなされる。 XLISP-STAT を起動すると、会話が始まり、つぎのような促進記号（prompt, プロンプト）>が行の先頭に表示される。

```
>
```

促進記号の後ろに式を入力すれば、インタプリタはその式を評価し、結果を表示する。例えば促進記号の後ろに数値を入力し「Enter」キーを押すと、その数値がつぎの行に表示され、そのつぎの行に促進記号が表示される。

なお、入力した数値が間違っていた場合には、「Enter」キーを押す前に、バックスペースキーで、間違えた数値を削除し、改めて正しい数値を入力する。以降においても、入力に間違えがあった場合には、バックスペースキーによって間違えた部分を削除してから、正しい入力を行うものとする。

```
> 1
1
>
```

数値演算は演算を示す記号と数値との組み合わせ、(+ 1 2) のような複合式 (compound expression) で行われる。

なお、以下の式において、最後の右括弧を入力し、「Enter」キーを押すと、対応する左括弧が反転表示されるので注意すること。このことにより、式が正しく入力されたかどうかをチェックできる場合がある。

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

この式 (+ 1 2) は、数値の 1 と 2 とを加える、という意味である。演算子 (operator、オペレータ) をオペランド (operands) の前に置く、このような記法を前置記法 (prefix notation) という。この記法は、標準的な数学の記法と異なるため、最初はわかりにくいかもしれない。しかし、この記法にはいくつかの利点がある。その 1 つは、オペランドがいくつでもよいことである。例えば、

```
> (+ 1 2 3)
6
> (* 2 3 4)
24
>
```

などである。いくつかの他の基本的なデータ形式はそれ自身で評価される。これらには、論理値も含まれる。

```
> t    ; true
T
> nil  ; false
NIL
```

なお、セミコロン；は、注釈文字であり、セミコロンの後ろは改行まで注釈となり、インタプリタから無視される。

文字列は、

```
> "Statistics is an important subject."
"Statistics is an important subject."
```

のように前後を引用符”で囲む。

インタプリタに記号 (Symbol) pi や x が与えられたならば、その記号が値と関連づけられているかがチェックされ、関連づけられていれば、その値が表示される。

```
> pi
3.141592653589793
```

与えられた記号が値と関連づけられていない場合には、つぎのようにエラーが表示される。

```
> x
Error: The variable X is unbound.
```

記号 pi は円周率 π として前もって定義されている。この時点では、記号 x はまだ値と関連づけられていない。記号 x に、例えば、値 1 を関連づけるには、

```
> (def x 1)
X
```

とする。ここで、X を入力すると、

```
> x
1
```

となり、記号 X が数値 1 と関連付けられていることがわかる。記号名を入力するときは大文字でも、小文字でも使用できる。LISP は内部的には小文字の記号名は大文字に変換している。

複合式 (compound expression) の評価は少し込み入っている。複合式はスペース (空白記号、タブ記号、あるいは改行記号) で区切られ、括弧で囲まれたリストである。リストの各構成要素は文字列か、数値か、あるいは他の複合式かである。つぎのような複合式

```
(+ 1 2 3)
```

を評価するとき、インタプリタはこのリストを関数呼び出し (function application) を表現しているものとして取り扱う。リストの最初の要素は LISP の関数を表わし、その他の各要素を引数として適用し、結果を戻す。上の例では、関数は記号 +で表現される加算関数である。リストの残りの要素は引数である。引数は 1 と 2 と 3 である。この式を評価するように要求されると、インタプリタは、加算演算を引数に適用し、

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

という結果を返す。関数の引数はその関数を適用する前に評価される。上の式では引数はすべて値であり、それ自身が評価されたものである。しかし、

```
> (+ (* 2 3) 4)
10
```

では、インタプリタはまず引数 (∗ 2 3) を評価し、その後、関数 + を適用する。数値、文字、そして記号は LISP で使用できる基本的なデータの型である。これらの基本的なデータの型は「単純データ (simple data)」として表現できる。LISP では、「複合データ (compound data)」と呼ぶもっと複雑なデータを取り扱える。複合データのもっとも基本的な形式はリストである。リストは list 関数で作られる。

```
> (list 1 2 3 4)
(1 2 3 4)
```

インタプリタで表示された結果は複合式に似ている。すなわち、一連の要素がスペースで区切られ、括弧で囲まれている。LISP の式は、単純な list である（LISP は LISt Processing に対する頭字語である。この言語は、数式の微分を数値計算するのではなく、数式そのものとして処理するというような、記号処理のタスクのためのツールとして、もともと開発された）。

記号 y にリスト (list 1 2 3 4) を関連付けると、

```
> (def y (list 1 2 3 4 5))
Y
```

となる。ここで、y を入力すると、

```
> y
(1 2 3 4 5)
```

となり、記号 y がリスト (1 2 3 4 5) と関連付けられていることがわかる。リスト以外にもベクトルや多次元配列のような、種々の複合データがある。

インタプリタに (+ 1 2) というリスト自身を結果として戻し、表示させるにはどうしたら良いのだろうか。このままタイプしてインタプリタにわたしてしまうと、インタプリタはそのリストを解釈し、つぎのような結果を戻すことになる。

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

インタプリタにリストを評価させないように伝える方法は、「引用 (quoting)」という方法である。式そのものを扱うのに、式を引用符で囲むが、それと同じようなことである。例えば、引用符を使用したつぎの２つの文章の意味は非常に異なっている。

```
Say your hobby!        (あなたの趣味を言いなさい)
Say "your hobby"!      (「あなたの趣味」と言いなさい)
```

LISP で引用 (quote) するためにはつぎのようにする。

```
> (quote (+ 1 2))
(+ 1 2)
```

この quote は関数ではない。したがって、quote は、関数を評価する、というすでに述べた規則には従わない、すなわち、その引数は評価されない。quote は特別形式 (special form) と呼ばれる。特別という意味はその引数の取り扱いの規則が特別であることを意味している。上述の基本的な評価規則と特別形式とで、LISP 言語のシンタックスができている。特別形式 quote に関しては、引用符を式の前につける簡便的な記法がしばしば使用される。

```
> '(+ 1 2)
(+ 1 2)
```

これは (quote (+ 1 2)) と同じである。この引用符は前だけで、後ろには必要ない。

4 XLISP-STAT の追加的なインタプリタ機能

XLISP-STAT における追加的なインタプリタ機能として、(1) 作業の保存方法、(2) コマンドの履歴を取る方法、(3) ヘルプの使用方法、および (4) 変数の表示および削除方法を説明する。

(1) 作業の保存方法

作業（ユーザーとインタプリタとの間でのセッション）を記録し、保存したい場合には、dribble 関数を使用する。

dribble 関数の実行を終了するには、

```
> (dribble)
```

とする。

[問題 3]

XLISP-STAT において、dribble 関数を実行し、つぎの式

```
(+ 1 2 3 4 5)
```

を、促進記号（prompt,プロンプト）>の後ろに、入力し、「Enter」キーを押しなさい。その結果は、

```
> (+ 1 2 3 4 5)
15
>
```

となる。ここで、dribble関数の実行を終了させ、エディタ（例えば、メモ帳、秀丸、WZ、あるいはMIFESなど）を用いて、dribbleのためのファイル（dribble関数の実行を記録し、保存したファイル）を開き、その内容を確認しなさい。ここで、メモ帳とは、Windows 95のアクセサリ内にあるメモ帳のことである（以下でも同じである）。

[問題3の解答]

dribble関数を実行するために、まず、フロッピーディスク内にフォルダLS-tk（なお、tkの部分は各人のイニシャルとすること）を作成し、さらにそのフォルダ内にサブフォルダdrib2004を作成しておくこと ([問題1]において出題済みである。なお、drib2004における2004は、西暦2004年を示している)。

実行開始した月日時刻が1月24日の11時12分として、dribble関数を実行する手順を説明する（各人は、実行開始した月日時刻を使用すること）。

dribbleのためのファイルの名前（dribble関数の実行結果を記録し、保存するためのファイルの名前）を、

```
01241112
```

とする（ファイル名に拡張子は付けないことにする。なお、01241112は、01月24日、11時12分を示している）。同じファイル名を使用すると、上書きされてしまい、以前の内容は消去されてしまうので、注意すること。

dribble関数を実行する手順はつぎのとおりである。XLISP-STATのFileをクリックし、Dribbleをクリックすると、保存する場所という画面が表示されるので、ファイルの種類: All Files(*.*), フロッピーディスクドライブa: 、フォルダLS-tkをクリックし、さらにdrib2004をクリックし、ファイル名 01241112 を入力すると、記録が始まり、タイプしたものや表示された結果は、上記ファイルに記録され、保存される (ファイル名に拡張子は付けないことにする)。

なお、式

```
(dribble "a:/LS-tk/drib2004/01241112")
```

としても、dribble 関数は実行される。

つぎの式

```
(+ 1 2 3 4 5)
```

を、促進記号（prompt,プロンプト）>の後ろに、入力し、「Enter」キーを押しなさい。その結果は、

```
> (+ 1 2 3 4 5)
15
>
```

となる。つぎに、dribble のためのファイル（ ファイル名：01241112 ）を開くために、dribble 関数の実行を終了させる。そのために、つぎのように入力する。

```
> (dribble)
```

dribble のためのファイルの内容を見るために、エディタ（例えば、メモ帳、秀丸、WZ、あるいは MIFES など）を用いて、

01241112

を開きなさい（なお、dribble のためのファイルを開く場合には、「ファイルの種類」を「すべてのファイル」とすること)。その結果は、

```
(+ 1 2 3 4 5)
15
> (dribble)
```

である。

(2) コマンドの履歴を取る方法

Common LISP にはコマンドの履歴を取る簡単なメカニズムが備わっている。この目的のために、- , +, ++, +++, *, **, および *** が使われる。各記号の意味はつぎのとおりである。

-	現在の入力行
+	最後に入力した行
++	+ の一つ前の行
+++	++ の一つ前の行
*	最後に評価された結果
**	* の一つ前に評価された結果
***	** の一つ前に評価された結果

記号 *, **, および *** がよく使われる。例えば、ある変数の自然対数 (log) を計算させたものの、結果の値を残す変数を指定するのを忘れていた場合、自然対数を再計算させる代わりに、上記の履歴機能を使うことができる。

```
> (def z (list 10 11 12))
Z
> z
(10 11 12)
> (log z)
(2.302585092994046 2.3978952727983707 2.4849066497880004)
> (def log-z *)
LOG-Z
```

この操作で、変数 log-z に z の自然対数が記録される。

ここで、log-z は、

```
> log-z
(2.302585092994046 2.3978952727983707 2.4849066497880004)
```

となっている。また、(def log-z *) において使用されている * という記号は、関数としては乗算を表わし、値としてはインタプリタが最後に評価した結果である。

LISP では、同じ名前で関数と値（それは def で定義された値かもしれない）との両方を定義できる。例えば、

```
> (def list '(1 2 3 4 5 6))
```

と入力し、list という名前で (1 2 3 4 5 6) というリストを定義しても、関数 list が壊れることはない、というメリットがある。

(3) ヘルプの使用方法

XLISP-STAT には、オンラインヘルプ機能があり、どの関数を使えば良いか、その関数はどのように使えば良いかを簡単に教えてくれる。
オンラインヘルプ機能として、(a) help、 (b) help* 、 および (c) apropos の３種類が使用できる。

(a) help

例えば、平均値を求める mean という関数については、help はつぎのような情報を与えてくれる。

```
> (help 'mean)
MEAN                                    [function-doc]
Args: (x)
Returns the mean of the elements x. Vector reducing.
NIL
```

mean の前に付いている引用符 (') は重要である。 help は関数であり、その引数は関数 mean を示す記号である。記号 mean の値ではなく、記号の名前を示すためには、名前の前に、引用符 (') が必要である。

(b) help*

関数名を正確には記憶していない場合、help* 関数を使用して情報を得ることができる。例えば、ヒストグラムに関する情報を調べたいとしよう。関数名として、"hist" までわかっているものとする。そこで、式、

```
> (help* 'hist)
```

は、文字列 "hist" を含んでいるすべての記号についての情報を表示する。

```
-----------------------------------------------------------------------
 Sorry, no help available on CHANGE-HIST-BINS-ITEM-PROTO
-----------------------------------------------------------------------
 Sorry, no help available on HIST
-----------------------------------------------------------------------
 HISTOGRAM                              [function-doc]
  Args: (data &key (title "Histogram"))
  Opens a window with a histogram of DATA. TITLE is the window
  title. The number of bins used can be adjusted using the histogram
  menu. The histogram can be linked to other plots with the link-
```

```
  views command. Returns a plot object.
-----------------------------------------------------------------------
 Sorry, no help available on HISTOGRAM-INTERNALS
-----------------------------------------------------------------------
 HISTOGRAM-PROTO                    [variable-doc]
 Histogram prototype
-----------------------------------------------------------------------
```

(c) apropos

関数 help* とは別に、関数 apropos を使えば、文字列 "hist" を含んでいる記号の一覧を得ることができる。

```
> (apropos 'hist)
CHANGE-HIST-BINS-ITEM-PROTO
HIST
HISTOGRAM-PROTO
HISTOGRAM
HISTOGRAM-INTERNALS
```

この後、help を使って特定のコマンドの、より詳細な情報を得ればよい。

(4) 変数の表示および削除方法

作業をしていると、def を使ってどんな名前の変数を定義したかを見たくなることがある。この場合には、関数 variables を使う。

```
> (variables)
(LOG-Z X Y Z)
```

使わない変数を消去し、スペースを空けたい場合がある。この場合には、関数 undef を使用する。変数 log-z を消去するには、

```
> (undef 'log-z)
LOG-Z
```

とする。変数 log-z が消去できたかどうかをチェックする。

```
> (variables)
(X Y Z)
```

確かに、変数 log-z は消去されている。

5　データを直接入力し、保存し、ロード する方法

5.1では、データを直接入力する方法を、5.2では、データを保存する方法を、そして5.3では、データ ファ イルをロード (load) する方法を説明している。

5.1 データを直接入力する方法

[問題4]
XLISP-STAT を起動し、[問題2] の「サイコロ投げの実験」における第1回から第30回までの実験データを、XLISP-STAT において直接入力しなさい。その後、データの表示および合計の計算によるチェックもしなさい。

[問題4の解答]
データ名を sr0130tk とする（なお、tk の部分は各人のイニシャルを使用すること）。実験データを、つぎのように、XLISP-STAT において直接入力する（実験データも各人のものを使用すること）。

```
> (def sr0130tk (list
                  6 2 1 3 5 5 4 5 2 3
                  2 6 4 5 6 5 5 4 5 6
                  1 1 6 1 5 4 5 4 5 5))
SR0130TK
```

ここで、

```
sr:     s: サイコロ;  r: red(赤)
0130:   実験: 第1回から第30回まで
tk:     t: tetsuo,  k: kamijo
```

である。各データが正しく入力されているかどうかをチェックするために、データを表示してみると、

```
> sr0130tk
(6 2 1 3 5 5 4 5 2 3 2 6 4 5 6 5 5 4 5 6 1 1 6 1 5 4 5 4 5 5)
```

となっている。チェックとして合計 (sum) を計算してみると、

```
> (sum sr0130tk)
121
```

となっている。

5.2 データを保存する方法

データを保存するためには、savevar 関数を使用する。この関数によって、1 つまたは複数の変数をファイルに保存することができる。この関数はファイルへの上書きになるので、既存のファイルと同じ名前のファイルを指定すれば、既存のファイルの内容は消えてしまう。

[問題 5]
[問題 4] に引き続いて、サイコロ投げの実験データである sr0130tk を、フロッピード ライブ a: にあるフロッピー内のフォルダ LS-tk にデータ ファイル名を tkamijo として保存しなさい (tk および tkamijo の部分は各人のものを使用すること)。

[問題 5 の解答]
XLISP-STAT において、

```
> (savevar 'sr0130tk "a:/LS-tk/tkamijo")
```

と入力し、「Enter」キーを押す。ここで、tk および tkamijo の部分は各人のものを使用すること。その結果、フロッピード ライブ a: にあるフロッピー内のフォルダ LS-tk にデータ ファイル tkamijo が、

```
tkamijo.lsp
```

のように、拡張子 lsp が付いて保存される。なお、savevar のあとの ' の後ろの sr0130tk がデータ名で、"a:/LS-tk/tkamijo"における tkamijo はデータ・ファイル名であり、savevar 関数を実行することにより、このデータ・ファイル名に拡張子 lsp が付くことになる。

[問題 6]
エディタ（例えば、メモ帳、秀丸、WZ、あるいは MIFES など）を用いて、フロッピード ライブ a: にあるフロッピー内のフォルダ LS-tk に保存されたデータ ファイル tkamijo.lsp の内容を確認しなさい（tk および tkamijo の部分は各人

のものを使用すること)。

[問題 6 の解答]
　エディタ（例えば、メモ帳、秀丸、WZ、あるいは MIFES など）を用いて、フロッピードライブ a: にあるフロッピー内のフォルダ LS-tk に保存されたデータ ファイル tkamijo.lsp を開くと、

```
(DEF SR0130TK (QUOTE
  (6 2 1 3 5 5 4 5 2 3 2 6 4 5 6 5 5 4 5 6  1 1 6 1 5 4 5 4 5 5)))
```

のようになっていることがわかる。なお、「ファイルの種類」を「すべてのファイル」とすること。

5.3 データ ファイルをロード (load) する方法

[問題 7]
　XLISP-STAT 内に、これからロードするデータ名 sr0130tk のデータのみが入力されているものと仮定する。variables 関数によって、sr0130tk が入力されていることを確認し、undef 関数により、その sr0130tk を削除しなさい。つぎに、variables 関数によって、sr0130tk が削除されていることを確認しなさい。その後で、[問題 5] において保存されたデータ ファイル tkamijo.lsp をロードすることにより、データ名 sr0130tk のデータをロードしなさい。さらに、データの表示および合計の計算によるチェックもしなさい。なお、tk および tkamijo.lsp の部分は各人のものを使用すること。

[問題 7 の解答]

(1) XLISP-STAT にロードするデータと同じデータがあるかどうかをチェックし, あればそれを削除する。ここでは、データ名 sr0130tk のデータを削除する。念のため削除されているかどうかをチェックする。
　(a) variables 関数によって、sr0130tk が入力されていることを、つぎのように確認する。

```
> (variables)
(SR0130TK)
```

(b) undef 関数によって、sr0130tk を、つぎのように削除する。

```
> (undef 'sr0130tk)
SR0130TK
```

(c) variables 関数によって、sr0130tk が削除されていることを、つぎのように確認する。

```
> (variables)
NIL
```

(2) XLISP-STAT において、File をクリックし、つぎに load をクリックする。ファイルを開くという枠が開くので、ファイルの場所:をデスクトップから出発して、該当するファイルまで、ダブルクリックしながら指定し、最後に、該当するファイルを開く。ここでは、ファイルを開くという枠において、フロッピーディスクドライブ a:のフォルダ LS-tk にある LISP 用ファイル

```
a:\LS-tk\tkamijo.lsp
```

をロードする（なお、tk および tkamijo.lsp の部分は各人のものを使用すること）。

ファイルの場所がデスクトップであるところから出発することにする。まず、マイコンピュータをダブルクリックする。つぎに、3.5 インチ FD(A:) をダブルクリックし、フォルダ LS-tk をクリックし、tkamojo.lsp をクリックし、「開く」をクリックする。

その結果、XLISP-STAT には、つぎのように、ファイル tkamijo.lsp がロードされる。

```
; loading A:\LS-tk\tkamijo.lsp
```

XLISP-STAT では、ここで促進記号が表示されない。そこで、(と) とを入力し、「Enter」キーを押すと、NIL が表示され、さらに、促進記号が、表示される。その後、XLISP-STAT は、そのファイル sr0130tk.lsp 内のデータ sr0130tk を利用することができる。

入力されているかどうかを確認すると、

```
> sr0130tk
(6 2 1 3 5 5 4 5 2 3 2 6 4 5 6 5 5 4 5 6 1 1 6 1 5 4 5 4 5 5)
```

となっており、データ sr0130tk が入力されていることがわかる。

データ sr0130tk の合計 (sum) を計算すると、

```
> (sum sr0130tk)
121
```

となっている。

6 エディタによってデータ ファイルを作成し、それをロード する方法

[問題 8]

エディタ (例えば、メモ帳、秀丸、WZ、あるいは MIFES など) を用いて、[問題 2] の「サイコロ投げの実験」における第 31 回から第 60 回までの実験データを、[問題 5] において作成されたファイル tkamijo.lsp 内に入力しなさい（なお、tkamijo.lsp は各人のものを使用すること)。

[問題 8 の解答]

(1) フロッピードライブ（例えば、ドライブ a: ）に [問題 5] で使用したフロッピーディスクを入れる。

(2) Windows95 のアクセサリ内のメモ帳を使用して、ファイル tkamijo.lsp にデータを入力する。

Windows95 のデスクトップにおいて、スタートをクリックする。プログラム、アクセサリ、そしてメモ帳を順次選択し、クリックする。メモ帳 が開くので、メモ帳において、ファイル tkamijo.lsp を開くと、

```
(DEF SR0130TK (QUOTE
  (6 2 1 3 5 5 4 5 2 3
   2 6 4 5 6 5 5 4 5 6
   1 1 6 1 5 4 5 4 5 5)))
```

のように、データ名 SR0130TK(sr0130tk) のデータがすでに入力されていることがわかる。

60 回サイコロを投げる、というサイコロ投げの実験における第 31 回から第 60 回までの tk(tetsuo kamijo) の実験データのデータ名を sr3160tk とする。ここで、

```
sr:      s: サイコロ;  r: red(赤)
3160:    実験: 第31回から第60回まで
tk:      t: tetsuo,  k: kamijo
```

である。データ名SR0130TKのデータのつぎに、データ名sr3160tkのデータを、つぎのように入力する。

```
(def sr3160tk (list
       3 1 1 2 6 4 2 6 4 3
       3 5 1 3 1 1 4 4 1 5
       3 2 6 2 3 6 1 1 2 1) )
```

つぎに、ファイルを上書き保存することにする。メモ帳において、「上書き保存」をクリックすると、ファイル名tkamijo.lspのファイルはフォルダLS-tk内に上書き保存される（各人のフォルダ内に、各人のファイル名で保存される）。

ファイルtkamijo.lspの内容が、

```
(DEF SR0130TK (QUOTE
  (6 2 1 3 5 5 4 5 2 3
   2 6 4 5 6 5 5 4 5 6
   1 1 6 1 5 4 5 4 5 5)))
(def sr3160tk (list
       3 1 1 2 6 4 2 6 4 3
       3 5 1 3 1 1 4 4 1 5
       3 2 6 2 3 6 1 1 2 1) )
```

のようになっていることを確認すること。

[問題9]

XLISP-STATを起動し、[問題8]で保存されたファイルtkamijo.lsp（ただし、各人のファイルを使用すること）をロードしなさい。その際に、variables関数を、必要であればundef関数を実行しなさい。さらに、データの表示および合計の計算によるチェックなどを実行しなさい。

[問題9の解答]

(1) フロッピードライブ（例えば、ドライブ a: ）に[問題8]で使用したフロッピーディスクを入れる。

(2) ロードするデータと同じデータがあるか否かをチェックする。

XLISP-STATにロードするデータと同じものがあるかどうかをvariables関数でチェックし, あればそれをundef関数で削除する。

(3) XLISP-STATにおいて、ファイルをロードする。

フロッピードライブ a: のフォルダLS-tkに保存したファイルtkamijo.lspを、[問題7の解答]において説明したと同様の方法でロード (load) する。

XLISP-STATにおいて、Fileをクリックし、つぎにloadをクリックする。ファイルを開くという枠が開くので、ファイルの場所:をデスクトップから出発して、該当するファイルまで、ダブルクリックしながら指定し、最後に、該当するファイルを開く。ここでは、ファイルを開くという枠において、フロッピードライブ a: のフォルダLS-tkにあるLISP用ファイル

```
a:\LS-tk\tkamijo.lsp
```

をロードすることにする。ファイルの場所がデスクトップであるところから出発することにする。まず、マイコンピュータをダブルクリックする。つぎに、3.5インチFD(A:)をダブルクリックし、フォルダLS-tkをクリックし、tkamijo.lspをクリックし、「開く」をクリックする。その結果、XLISP-STATには、つぎのように、ファイル、tkamijo.lspがロードされる。

```
>
; loading A:\LS-tk\tkamijo.lsp
```

XLISP-STATでは、ここで促進記号が表示されない。そこで、(と) とを入力し、「Enter」キーを押すと、NILが表示され、さらに、促進記号が、表示される。その後、XLISP-STATは、そのファイルtkamijo.lspを使用することができる。

(4) データ表示および合計の計算によるチェックを行う。　促進記号 (prompt, プロンプト) >の後ろにsr3160tkを入力し、[Enter]キーを押すと、

```
> sr3160tk
(3 1 1 2 6 4 2 6 4 3 3 5 1 3 1 1 4 4 1 5 3 2 6 2 3 6 1 1 2 1)
```

となっており、データsr3160tkが入力されていることがわかる。

データの合計 (sum) を計算すると、

```
> (sum sr3160tk)
87
```

となっていることがわかる。

7 XLISP-STAT 自身のデータ生成機能

XLISP-STAT 自身のデータ生成機能として、(1) において、体系的なデータの生成機能について、そして (2) において、乱数の生成機能について説明する。

(1) 体系的にデータを生成する機能

(a) 関数 iseq

関数 iseq (integer-sequence の省略形) は与えた二つの整数の間の数をリストとして作り出す。一般形式はつぎのとおりである。

```
( iseq 〈 start 〉〈 end 〉 )
```

例えば、1 から 10 までの整数のリストを作るには、つぎのようにする。

```
> (iseq 1 10)
(1 2 3 4 5 6 7 8 9 10)
```

関数 iseq は引数が 1 個でもよい。その場合、引数の値は正整数でなければならず、0, 1, 2, ⋯ , n-1 の n 個の整数からなるリストが結果として戻ってくる。例えば、つぎの様である。

```
> (iseq 10)
(0 1 2 3 4 5 6 7 8 9)
```

(b) 関数 rseq

関数 rseq の一般形式は、つぎのとおりである。

```
( rseq 〈 a 〉〈 b 〉〈 n 〉 )
```

関数 rseq は、数値〈 a 〉と数値 〈 b 〉 との間を〈 n 〉個に等分した値のリストを作ってくれる。例えば、つぎのとおりである。

```
> (rseq 1 2 5)
(1.0 1.25 1.5 1.75 2.0)
```

(c) 関数 repeat

関数 repeat は、特定のパターンの列を生成するために使われる。一般形式はつぎのとおりである。

```
(repeat 〈 vals 〉〈 pattern 〉)
```

パラメータ 〈 vals 〉は、単純なデータ項か、項のリストである。2 番目のパラメータ 〈 pattern 〉は 1 個の正整数か、正整数のリストでなければならない。〈 pattern 〉が 1 個の正整数の場合、〈 vals 〉が単に 〈 pattern 〉回繰り返される。例えば、つぎのとおりである。

```
> (repeat 3 5)
(3 3 3 3 3)
> (repeat (list 3 4 5) 3)
(3 4 5 3 4 5 3 4 5)
```

〈 pattern 〉がリストの場合、〈 vals 〉は 〈 pattern 〉と同じ長さのリストでなければならず、この場合、〈 vals 〉の各要素に対応する 〈 pattern 〉の要素の回数だけ繰り返される。例えば、つぎのとおりである。

```
> (repeat (list 4 5 6) (list 1 2 3))
(4 5 5 6 6 6)
```

(2) 乱数の生成

疑似乱数を生成するために多数の関数が用意されている。ここでは、一様分布のみ説明する。例えば、0 と 1 との間の一様分布に従う独立な乱数を 5 個生成させるには、つぎのようにする。

```
> (uniform-rand 5)
( 0.6787434954496435    0.06480958196617274
  0.7667714171778037     0.17435839065028547
  0.5806454873319216)
```

8 データ編集機能

データ編集機能として、(1) では、部分リストの作成とリストの一部の削除について、(2) では、リストの結合について、そして (3) では、データの変更（修正）について説明する。

(1) 部分リストの作成とリストの一部の削除

(a) 関数 select

関数 select は、リストから 1 個ないし、数個の要素を選択するために使われる。 例えば、xを、

```
> (def x (list 6 2 1 3 9 7 8 5))
```

と定義し、(select x i) とすれば、xの 第 i 番目の要素が得られる。LISP では、FORTRAN 言語とは異なり C 言語と同様に、リストの最初は第 0 番目である。すなわち、xの添字番号は 0, 1, 2, 3, 4, 5, 6, そして 7 である。このため、つぎのようになる。

```
> (select x 0)
6
> (select x 2)
1
```

一度に、複数の要素を得るためには、1 個の添字番号の代わりに、添字番号のリストを使う。

```
> (select x (list 0 2 4))
(6 1 9)
```

(b) 関数 remove

関数 remove は、リストから、指定された値を持つ要素を除いたリストを戻す。例えば、つぎのとおりである。

```
> (remove 3 (list 1 2 3 4 5 6))
(1 2 4 5 6)
> (remove 2 (list 1 2 2 3 3 2 4 5 3 6))
(1 3 3 4 5 3 6)
```

(c) リストから、ある添字番号以外の要素を抜き出す方法。

例えば、リスト x

```
> x
(6 2 1 3 9 7 8 5)
```

の、添字番号２以外の要素をすべて抜き出すことにする。ここで、要素の総数であるリストの長さを求める関数 length を使用する。例えば、x の長さはつぎのとおりである。

```
> (length x)
8
```

以下に、2 つの方法を説明する。

方法 1

まず、関数 select の ２番目の引数で指定する添字番号のリストを、関数 remove で作成する。すなわち、つぎのとおりである。

```
> (remove 2 (iseq (length x)))
(0 1 3 4 5 6 7)
```

得られたリストは、2（すなわち、第２番目）以外の数字から成るリストである。すなわち、添字番号２以外の要素をすべて抜き出すには、つぎのとおりとする。

```
> (select x (remove 2 (iseq (length x))))
```

結果として、つぎのリストが得られる。

```
(6 2 3 9 7 8 5)
```

方法２

まず、比較の関数 /= (意味は not equal) を使い、２に等しくない添字を表示させる。すなわち、つぎのようにする。

```
> (/= 2 (iseq (length x)))
(T T NIL T T T T T)
```

関数 which は要素が nil でない添字番号リストを求めるために使われる。

```
> (which (/= 2 (iseq (length x))))
(0 1 3 4 5 6 7)
```

これらを使って、第２番目以外の要素を選択することができる。

```
> (select x (which (/= 2 (iseq (length x)))))
(6 2 3 9 7 8 5)
```

この書き方は１個の要素を削除する際には、めんどうすぎるかもしれない。which はより汎用的に使用できる。つぎにその方法を説明する。

(d) リストからある条件を満足する要素を選択する方法

例えば、リスト x

```
> x
(6 2 1 3 9 7 8 5)
```

から、５より大きい要素を抜き出すにはつぎのようにする。

```
> (select x (which (< 5 x)))
(6 9 7 8)
```

(2) リストの結合

複数の短いリストをまとめて一つの長いリストにしたい場合は、関数 append か 関数 combine を使う。例えば、

```
> (def x1 (list 1 2 3))
X1
> (def x2 (list 4))
X2
> (def x3 (list 5 6))
X3
```

のとき、つぎのようになる。

```
> (append x1 x2 x3)
(1 2 3 4 5 6)
> (combine x1 x2 x3)
(1 2 3 4 5 6)
```

この二つの関数の違いは、リストのなかにリストがある場合の取り扱い方にある。append は一番外側のリストだけをまとめて一つのリストにするの対し、combine は再帰的にサブリストにも作用し、単純な要素からなる一つのリストにまとめる。例えば、

```
> (def y1 (list (list 1 2) 3))
Y1
> (def y2 (list 4 5))
Y2
```

```
> (def y3 (list 6))
Y3
```

のとき、

```
> (append y1 y2 y3)
((1 2) 3 4 5 6)

> (combine y1 y2 y3)
(1 2 3 4 5 6)
```

となる。なお、append の引数はリストでなければならない。したがって、

```
(append 1 2 (list 3 4 5 6))
```

は誤りで、つぎのようなエラーメッセージが示される。

```
> (append 1 2 (list 3 4 5 6))
Error: bad argument type - 1
Happened in: #<Subr-APPEND: #7617f8>
```

combine は、引数として、リストでない数値や、単純アイテムを使用でき、例えば、

```
(combine 1 2 (list 3 4 5 6))
```

は、

```
> (combine 1 2 (list 3 4 5 6))
(1 2 3 4 5 6)
```

となる。

[問題 10]

XLISP-STAT において、フロッピードライブ a: にあるフロッピー内のフォルダ LS-tk に保存されているファイル tkamijo.lsp をロードしなさい(各人のファイルを使用しなさい)。つぎに、そのファイル内のデータ ファイル sr0130tk および sr3160tk を「combine」し、それをつぎのように入力しなさい。

```
> (def sr0160tk (combine sr0130tk sr3160tk))
```

sr0160tk を、つぎのように表示させて、確認しなさい。

```
> sr0160tk
(6 2 1 3 5 5 4 5 2 3 2 6 4 5 6 5 5 4 5 6 1 1 6 1 5 4 5 4 5 5
 3 1 1 2 6 4 2 6 4 3 3 5 1 3 1 1 4 4 1 5 3 2 6 2 3 6 1 1 2 1)
```

sr0160tk が正しいことが確認されたなら、

```
(def sr0160tk (combine sr0130tk sr3160tk))
```

を、ファイル tkamijo.lsp に「貼り付け」を実行しなさい。

[問題 10 の解答]

ファイル tkamijo.lsp の内容は、

```
(DEF SR0130TK (QUOTE (6 2 1 3 5 5 4 5 2 3
                      2 6 4 5 6 5 5 4 5 6
                      1 1 6 1 5 4 5 4 5 5)))
(def sr3160tk (list
       3 1 1 2 6 4 2 6 4 3
       3 5 1 3 1 1 4 4 1 5
       3 2 6 2 3 6 1 1 2 1) )
(def sr0160tk (combine sr0130tk sr3160tk))
```

のようになっている。

(3) データの変更（修正）

特別形式 (special form) setf は、リストの一つまたは複数の要素の値を変更するために使われる。リスト x をつぎのように定義しておく。

```
> (def x (list 6 2 1 3 9 7 8 5))
X
> x
(6 2 1 3 9 7 8 5)
```

この中で、第 2 番目の値 1（なお、6 が第 0 番目の値である）を 4 に変更することにする。式

```
(setf (select x 2) 4)
```

は、この変更を行ってくれる。その結果はつぎのようになる。

```
> (setf (select x 2) 4)
4
> x
(6 2 4 3 9 7 8 5)
```

setf の一般形式は、

```
(setf 〈form〉〈value〉)
```

である。ここに、〈form〉はリストから 1 個 (ないし複数) の変更される要素を選択するために使われる式であり、〈value〉は変更後の値（ないし値のリスト）である。例えば、xが、

```
(def x (list 6 2 1 3 9 7 8 5))
```

で定義されるとき、式、

```
(setf (select x (list 0 2)) (list 10 20))
```

は、第 0 番目と第 2 番目の要素の値を、10 と 20 とに変更する。すなわち、

```
> (setf (select x (list 0 2)) (list 10 20))
(10 20)
> x
(10 2 20 3 9 7 8 5)
```

となる。

ここで、LISP における記号 (名前) について注意しておくことにする。LISP における記号 (名前) は種々のアイテムにつけたラベルである。あるアイテムに def コマンドで名前をつけるとき、新しいアイテムが作られるわけではない。例えば、つぎの二つの式、

```
(def x (list 1 2 3 4 5 6))
```

と

```
(def y x)
```

とを評価すると、記号xとyとは同じリストに二つの異なった名前を付けただけとなる。
すなわち、

```
> x
(1 2 3 4 5 6)
> y
(1 2 3 4 5 6)
```

となっている。その結果、xで参照される要素の値を変更した場合、xとyとは同じリストを示しているので、yで参照される要素の値も変更されることになる。

もし、xを変更する前に、xのコピーを作成し、それをyに保存したいならば、関数 copy-list を用いて、明示的にそうしなければならない。式、

```
(def y (copy-list x))
```

は、xのコピーを作り、そのコピーをyの値とするということを定義している。この場合、xとyとは、別のアイテムを参照しており、xを変更してもyには影響を与えない。特別形式 setf で値を記号に結びつけることができる。

```
> (setf z 5)
5
> z
5
```

変数を定義するのに setf ではなく、def を使うメリットは、 def は定義した名前を記憶してくれることである。すなわち、関数 variables を使うことによって、 def で定義して使用可能である変数を知ることができる。

9 LISP-STATによるプログラミング

9.1 はじめに

LISP プログラミングの基本的な操作は、関数定義である。ここでは、関数定義のための道具および LISP プログラミングにおいて使用されるプログラミング技法と原理を、Tierney 著の *LISP-STAT* に基づいて説明する (Tierney(1990) 参照)。使用するソフトウエアは、Tierney が開発した XLISP-STAT である。

9.2 では、単純な関数の作成について、9.3 では、述語と論理式について、9.4 では、条件付き評価について、そして 9.5 では、再帰と繰返しについて説明している。

9.2 単純な関数の作成

LISP 関数は、特殊形式 defun を用いて定義される。defun に対する基本的なシンタックスは、

$$(defun\ \langle name\rangle\ \langle parameter\rangle\ \langle body\rangle)$$

である。ここに、$\langle name \rangle$は関数を参照するために使用される記号であり、$\langle parameter \rangle$は関数のパラメターに名前を付けるために使用される記号のリストであり、$\langle body \rangle$は 1 つあるいはそれ以上の式から構成される。関数が呼び出されると、本体 ($\langle body \rangle$) 内の式が次々に評価され、最後の式を評価することによって得られる値が関数の結果として返される。defun はその引数を評価しないので、defun のいずれの引数にも引用符を付ける必要はない。

数字 (number) の 2 乗 (square) を計算する関数は、

```
(defun square (number)
   (* number number)
)
```

と定義される。5 の 2 乗は、

```
> (square 5)
25
```

と計算される。

[問題 11]

データセット x の平均値を計算する関数 mean-tk を作成すること (tk の部分は、作成者の氏名のイニシャルとする)。XLISP-STAT には、平均値を計算する関数 mean が組み込まれているので、その関数の定義を失わないように、最後に作成者のイニシャルを追加した関数名 mean-tk を使用することにする。

x のデータの個数（大きさ）を n とすると、x の平均値は、

$$\sum_{i=1}^{n} x_i/n$$

と定義される。

[問題 11 の解答]

XLISP-STAT による関数は、

```
(defun mean-tk (x)
   (/ (sum x) (length x)
   )
)
```

となる。関数 sum は、その引数の要素の合計を計算し。関数 $length$ は、その引数の要素の個数 (大きさ) を計算している。1,2,3,4,5, および 6 の平均値は、

```
> (mean-tk (list 1 2 3 4 5 6))
3.5
```

と計算される。

9.3 述語と論理式

述語は条件が真であるかないかを決定する関数である。それらは、真に対しては、NON-NIL 値、しばしば記号 T を返し、偽に対しては NIL を返す。数の比較において、述語 $<$, $>$, $=$, $/=$, およびその他の比較のための述語を使用することができる。これらの述語はいずれも、2 つあるいはそれ以上の引数をとる。

述語 $<$ はすべての引数が昇順になっているときは真で、T を返し、そうでないときは偽で、NIL を返す。例えば、つぎのとおりである。

```
> (< 1 2 3)
T
> (< 1 3 2)
NIL
```

述語 $>$ も同様である。述語 $=$ は全ての引数が等しいときは真で、T を返し、そうでないときは偽で、NIL を返す。例えば、つぎのとおりである。

```
> (= 1 1)
T
> (= 1 1 1)
T
> (= 1 2)
NIL
```

述語 $/=$ は全ての引数が等しくないときは真で、T を返し、そうでないときは偽で、NIL を返す。例えば、つぎのとおりである。

```
> (/= 1 2)
T
> (/= 1 2 3)
T
> (/= 1 1)
NIL
> (/= 1 1 1)
NIL
```

特殊形式 and と or および関数 not は、それらを組み合わせることによって、より複雑な述語を作成することができる。

9.4 条件付き評価

LISP における条件付き評価の基本的な構成体は、cond である。
cond 式の一般形式は、

$$
\begin{array}{lll}
(& cond & (\ \langle \ p \ \ 1 \ \rangle \ \langle \ e \ \ 1 \ \rangle \ ) \\
 & & (\ \langle \ p \ \ 2 \ \rangle \ \langle \ e \ \ 2 \ \rangle \ ) \\
 & & \ldots \\
 & & (\ \langle \ p \ \ n \ \rangle \ \langle \ e \ \ n \ \rangle \ ) \\
) & &
\end{array}
$$

である。リスト $(\langle p \ \ 1 \ \rangle \ \langle e \ \ 1 \ \rangle)$ などは、条件節と呼ばれる。条件節のそれぞれの第 1 の式 $\langle p \ \ 1 \ \rangle$, ..., $\langle p \ \ n \ \rangle$ は、真に対しては NON-NIL，あるいは T を与え、偽にたいしては NIL を与える述語式である。cond は、1 つの述語式が真になるまで述語式を順に評価する。もし、ある述語、例えば、$\langle p \ \ i \ \rangle$ が真であるとすれば、それに対応する式 $\langle e \ \ i \ \rangle$ が評価され、その式の結果が cond の式の結果として返される。もしすべての述語式が真でないならば、NIL が返される。

cond を用いて、引数の絶対値を求める関数を定義すると、

```
(defun abs-cond-01-tk (x)
   (cond ( (> x 0) x)
         ( (= x 0) 0)
         ( (< x 0) (- x)
```

```
            )
        )
    )
```

となる。abs-cond-01-tk を用いると、

```
> (abs-cond-01-tk 2)
2
> (abs-cond-01-tk 0)
0
> (abs-cond-01-tk -3)
3
```

となる。

cond に加えて LISP にはいくつかの他の条件付き評価の構成体がある。それらのうちで最も重要なものは if である。特殊形式 if は 3 つの式、述語式 (predicate)、結果式 (consequent)、および代替式 (alternative) をとる。最初に述語式が評価される。もしそれが真であるなば、結果式が評価され、その結果が if 式の結果として返される。そうでなければ代替式が評価され、その結果が返される。もし、オプションである代替式がないならば、NIL が返される。if 式の一般形式は、

$$(if\ \langle predicate\rangle\ \langle consequent\rangle\ \langle alternative\rangle)$$

である。if 式を用いて、絶対値を計算する関数 abs-if-01-tk を定義すると、

```
(defun abs-if-01-tk (x)
       (if (> x 0)
           x
           (- x)
       )
)
```

となる。abs-if-01-tk を用いると、

```
> (abs-if-01-tk 2)
2
```

```
> (abs-if-01-tk 0)
0
> (abs-if-01-tk -3)
3
```

となる。

9.5 再帰と繰返し

LISP は再帰的言語 (recursive language) である。その結果として、LISP では、繰返しを必要とするような数値計算を、再帰的に実行することができる。

ここで、正整数 n の階乗 $n!$ の計算について考えてみることにする。階乗 $n!$ は、

```
0! = 1
n! = n*(n-1)*(n-2)* ··· *3*2*1
```

と定義される。$n!$ は、

```
n! = n*(n-1)!   (ただし、0! = 1)
```

と計算される。すなわち、n の階乗は、n に $(n-1)$ の階乗を乗ずることよって計算される。

正整数 n の階乗を計算する LISP 関数 factorial-01-tk を定義すると、

```
(defun factorial-01-tk (n)
   (if (= n 0)
       1
       (* n (factorial-01-tk (- n 1)
            )
       )
   )
)
```

となる。factorial-01-tk を用いると、

```
> (factorial-01-tk 3)
6
> (factorial-01-tk 10)
3628800
> (factorial-01-tk 20)
2432902008176640000
```

となる。関数 (factorial-01-tk n) を用いて、1000 の階乗を計算しようとすると、エラーとなる。すなわち、

```
> (factorial-01-tk 1000)
error: system stack overflow
```

となる。

反復的な計算のための特殊形式に do がある。do 式の一般形式は、

$$
\begin{array}{l}
(\ do\ \ (\ \ (\ \ \langle\ name\ 1\rangle\ \ \langle\ initial\ 1\rangle\ \ \langle\ update\ 1\rangle) \\
\qquad\quad \ldots \\
\qquad\quad (\ \ \langle\ name\ n\rangle\ \ \langle\ initial\ n\rangle\ \ \langle\ update\ n\rangle) \\
\qquad\ )\\
\qquad\ (\qquad \langle\ test\rangle \qquad \langle\ result\rangle \qquad) \\
)
\end{array}
$$

である。do には、2 つの引数がある、終了テスト式 (test) を従えている 3 つの要素から成るリストおよび結果式 (result) である。3 つの要素のそれぞれは、状態変数に対する名前 (name)、初期設定式 (initial)、および更新式 (update) である。do は初期設定式を評価し、状態変数をこれらの値とする。つぎに、終了テスト式を評価する。もし終了テスト式を評価して真であるならば、結果式が評価され、その結果が返される。もし終了テスト式が真でなければ、更新式が評価され、変数が新しい値となり、終了テスト式が評価される。同様な手順が繰り返される。do を使用して階乗関数を定義すると、

```
(defun factorial-do-01-tk (n)
  (do ( (counter 1 (+ 1 counter)
        )
        (product 1 (* counter product)
```

```
          )
        )
        ( (> counter n) product)
    )
  )
```

となる。factorial-do-01-tk を用いると、

```
> (factorial-do-01-tk 3)
6
> (factorial-do-01-tk 10)
3628800
> (factorial-do-01-tk 20)
2432902008176640000
```

となる。関数 (factorial-do-01-tk n) を用いて、1000 の階乗を計算するとつぎのようになる。すなわち、

```
> (factorial-do-01-tk 1000)
402387260077093773543702433923003985719374864210714632543799910429938 5
1239862902059204420848696940480047998861019719605863166687299480855890
1323829669944590997424504087073759918823627727188732519779505950995276
1208749754624970436014182780946464962910563938874378864873371191810458
2578364784997701247663288983595573543251318532395846307555740911426241
7474349347553428646576611667797396668820291207379143853719588249808126
8678383745597317461360853795345242215865932019280908782973084313928444
0328123155861103697680135730421616874760967587134831202547858932076716
9132448426236131412508780208000261683151027341827977704784635868170164
3650241536913982812648102130927612448963599287051149649754199093422215
6683257208082133318611681155361583654698404670897560290095053761647584
7728421889679646244945160765353408198901385442487984959953319101723355
5566021394503997362807501378376153071277619268490343526252000158885351
4733161170210396817592151090778801939317811419454525722386554146106289
2187960223838971476088506276862967146674697562911234082439208160153780
8898939645182632436716167621791689097799119037540312746222899880051954
```

```
4441428201218736174599264295658174662830295557029902432415318161721046
5832036786906117260158783520751516284225540265170483304226143974286933
0616908979684825901254583271682264580665267699586526822728070757813918
5817888965220816434834482599326604336766017699961283186078838615027946
5955131156552036093988180612138558600301435694527224206344631797460594
6825731037900840244324384656572450144028218852524709351906209290231364
9327349756551395872055965422874977401141334696271542284586237738753823
0483865688976461927383814900140767310446640259899490222221765904339901
8860185665264850617997023561938970178600408118897299183110211712298459
0164192106888438712185564612496079872290851929681937238864261483965738
2291123125024186649353143970137428531926649875337218940694281434118520
1580141233448280150513996942901534830776445690990731524332782882698646
0278986432113908350621709500259738986355427719674282224875758676575234
4220207573630569498825087968928162753848863396909959826280956121450994
8717012445164612603790293091208890869420285106401821543994571568059418
7274899809425474217358240106367740459574178516082923013535808184009699
6372524230560855903700624271243416909004153690105933983835777939410970
0277534720000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
```

である。

繰返し構成体に loop がある。loop の一般形式は、

$$(loop\ \langle body \rangle)$$

である。ループ内で、各回において順番に $\langle body \rangle$ 内の各式の実行を無限に繰り返す。ループを終了するためには、本体 (body) に、return 式を含めておけばよい。loop を使用して、階乗関数を定義すると、

```
(defun factorial-loop-01-tk (n)
  (let ( (i 1)
         (n-fac 1) ;;; 初期値
       )
```

```
        (loop (if (> i n) (return n-fac)
              )   ;;; n 回の繰返し
              (setf n-fac (* i n-fac)
              )
              (setf i (+ i 1)
              )
        ) ;;; loop
    ) ;;; let
) ;;; defun
```

となる。factorial-loop-01-tk を用いると、

```
> (factorial-loop-01-tk 3)
6
> (factorial-loop-01-tk 10)
3628800
> (factorial-loop-01-tk 20)
2432902008176640000
```

となる。関数 (factorial-loop-01-tk n) を用いて、1000 の階乗を計算すると、関数 (factorial-do-01-tk n) を用いて、1000 の階乗を計算した結果と同じ結果が得られる。

参考文献

[1] Harvey, B. and Wright, M.(1994), *Simply Scheme: Introducing Computer Science*, The MIT Press.

[2] 栗原正仁 (1993), 『対話による Common Lisp 入門』, 森北出版.

[3] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I.(1962), *LISP 1.5 Programmer's Manual*, The MIT Press.

[4] Soft Warehouse(1992), *muLISP 90, Reference Manual*, Soft Warehouse.

[5] Springer, G. and Friedman, D.P.(1989), *Scheme and the Art of Programming*, The MIT Press.

[6] Steele, Jr., G.L.(1990), *Common LISP: The Language*, Second Edition, Digital Press.
邦訳: 井田昌之 翻訳監修 (1991),『COMMON LISP』, 第 2 版, 共立出版.

[7] 竹村彰通 (1997),『統計』, 共立出版.

[8] 竹内郁雄 (1986), 『初めての人のための LISP』, サイエンス社.

[9] 垂水共之 (1999), 『Lisp-Stat による統計解析入門』, 共立出版.

[10] Tierney, L.(1990), *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*,Wiley.
邦訳: 垂水共之, 鎌倉稔成, 林 篤裕, 奥村晴彦, 水田正弘 共訳 (1996), 『LISP-STAT 入門』, 共立出版.

[11] Winston, P.H. and Horn, B.K.P.(1989), *LISP*, 3rd Edition, Addison Wesley.
邦訳: 白井良明, 安部憲広, 井田昌之 共訳 (1996), LISP, 原書第 3 版, ( )( ).

[12] 湯浅太一 (1991), 『Scheme マニュアル』, 岩波書店.

[13] 湯浅太一 (1991), 『Scheme 入門』, 岩波書店.

[14] 湯浅太一, 萩谷昌己 (1986),『Common Lisp 入門』, 岩波書店.