SS 入門

佐藤　定夫[1]

2020 年 1 月改定版

[1]東京電機大学　情報システムデザイン学系, 〒 350-0394　埼玉県比企郡鳩山町石坂:
sato@u.dendai.ac.jp

始めに

このノートは，筆者の ss(Lisp の 1 つの方言）を初めて学ぶ人のために基本的な Lisp の文法と ss におけるグラフィックの取り扱いについて簡単にまとめたものである。このノートは、最初 stk について 2002 年に書かれた。stk を参考にして開発したのが ss である。特に TCL/TK 関係は stk とまったく同じインターフェイスを採用したのですでに stk について知っている読者は容易に理解可能であろう。難しい構文などはあとまわしにして，実際的にどのようなプログラムスタイルが望ましいかを筆者の独断と偏見で書いてある。そのため，あまり Scheme らしくないとの指摘もあるとは思うがお許しいただきたい。

2005 年 4 月,　佐藤　定夫

2007 年 5 月,　改定版

2015 年 10 月,　改定版

2020 年 1 月,　改定版

Contents

Chapter 1

ssを使う

1.1 ssとは

ssはscheme(Lisp言語の一種）にグラフィックツールのTcl/TK、OpenGLの機能を内蔵させた言語である。従って，従来Lispの持っている

1. 簡潔な文法

2. インタープリタであるためコンパイルなしで，実行できる。

3. 扱うデータ型が柔軟である。

4. メモリー管理をユーザーがする必要がない。

5. 記号処理に適している。

といった特徴は当然，受け継いでおり，さらにTcl/T kと同様に

簡単にGUIが書ける。

さらに

簡単な通信機能を持っている。

などの利点がある。このような言語として、筆者は従来stkを使用してきたのだがstkの欠点を補足する処理系を新しく作ったのがssである。この名前はsmall schemeとでも解釈していただきたい。stkに無かった特徴としては

コンパイラがある。インタプリタにおいてもコマンドラインは1パスコンパイラによって処理されてから実行される。

日本語が使える。scheme関数はもちろんTCL/TKも最新versionを使用している。

有理数演算、複素数演算ができる。

OpenGLも内蔵されている。

欠点としては、compileしている関係でdebugがややしずらい。

1.2 Install

現在Linux,FreeBSD,Cygwinに対応しており、

http://www.u.dendai.ac.jp/~sato/

からDownloadできる。

1.3 起動と終了

ターミナル ウインドゥのコマンドラインから

ss

と入力する。すると起動の後
user >

となって，コマンド入力を受け付ける状態になる。ここで，ためしに
user >(+ 3 4)

のようにタイプしてリターンキーを押すと

7

と応答が帰ってきて，ふたたび
user >

とプロンプト（入力を促す文字列）があらわれて入力可能な状態になる。このように，入力に対しそれを計算した結果を返すということを繰り返すのが基本的な動作である。終了するときは
user >(bye)

と入力すればよい。

1.4 プログラムの load

さて，毎回プログラムを手でタイプするのでは大変である。そこで，適当なエディタ（たとえば Win なら notepad, ... Unix なら vi,emacs,...) を使ってコマンドをそこに書いておく。ss には，**ssed** という ss で書かれたエディタが付属されているのでコマンドラインから

ssed

として使ってもよい。たとえば

```
(write (+ 3 4))
(write "Hello!")
```

と書いたファイルを作り，test.ss のような名前を付けて保存しておく。この拡張子.ss は.lsp,.stk または.scm でもよい。そして，ss を起動して
user >(load ”test”)

と入力すると，ss はこのファイルの内容を順番に処理してくれる。こうして大きなプログラムを作ることもできるのである。C 言語などとは違って，scheme の場合 main 関数のようなものはなく，単にファイルに書かれたコマンドを順に実行していくことに注意されたい。もし実行がすべて終了すれば，一般に scheme は load 関数の返す値を画面に出力する。この例では，write という画面出力の命令を使っているので実行すると

7 ”Hello!” user>

のように画面に出力が現れる。ここで

```
value: #<>
```

が表示されるかもしれない。この#<> は undefined という特殊なアトムが値として返されていることを表示している。これは C 言語の void 関数のようなもので、値に意味がない関数を示している。write などがそうである。write は、出力に意味があって返す値には意味がない。ちょっと見にくいのは，改行命令がないためなので test.ss を

```
;;   改良版 test.ss
(write (+ 3 4))
(newline)      ;;; 改行
(display "Hello!")
(newline)
```

に変更してから，再度 load を実行すると

```
 7
 Hello!
user>
```

と見やすくなる。これで，いわゆるハローワールドのプログラムができたわけである。display は write に似ているが，引用符”(ダブルクオート）を出力しない。**;** は，プログラムに注釈をつけるのに使われる。scheme は; から行末までを無視する。注釈には日本語を書いてもよい。

1.5 compiler

ss は 1-pass compiler を内蔵しており、コマンドラインは内部コードに変換されてから実行される。scheme のプログラム ソースをあらかじめコンパイルしておくと実行を高速にすることができる。test.ss をコンパイルするには、ターミナルのコマンドラインから

makesr test

と実行する。これにより

test.sr2

という名前のファイルが作られる。これがコンパイルされたファイルである。ss の load 命令は、拡張子を省略すると、.sr2 を最優先に load する。makesr は、ファイルの更新時間をチェックするので、もし test.ss を変更したら単に

makesr

とすると、自動的に test.sr2 を作ってくれる。このように引数なしで makesr を実行すると、その directory にあるすべての scheme ソースについてこれが行われる。
load 命令ではなく

ss test

のように、コマンドラインで指定して起動してもよい。

1.6 scheme の文法：S 式

前節で見たように scheme は, ある入力に対してそれを実行（評価という）してその結果を画面に出力する。このとき入力されるものは S 式でなければならない。S 式の定義は簡単である。

(S1) **アトム**は S 式である。

(S2) 任意の個数の s1,s2,s3,$\cdots$ を S 式とすると (s1 s2 s3 $\cdots$) は S 式である。これを**リスト**という。

(S3) 2 つの S 式 s1,s2 に対して (s1 . s2) は S 式である。これを**ドット対**という。ドットの前後には空白が必要である。

ここで，アトムとは

1. 数 1,-2,3/14,3.14,+i,3-i ···

2. シンボル x,y,z, +, -, write ··· これは変数や関数などの名前を表す。scheme では名前に+,-のような記号を使ってもよい。ss では、大文字と小文字を区別する。従って、x と X は異なるシンボルを表す。システムがあらかじめ定義している、ほとんどのシンボルは小文字からなっている。

3. 文字列 "Hello!"など。ダブルクォートで囲まれたもの。

4. 文字 #\a(小文字の a),#\A,#\newline (改行文字）, #\ あ など。

5. ベクトル #(1 2 3) のように表される

6. 真偽値 #t（真）,#f（偽）

などである。第 2 の規則 (S2) で作られるものをリストと言い，特に S 式が 0 個の場合

()

は**空リスト**と呼ばれる。古典的な Lisp ではこれは **NIL**(**nil**) とも呼ばれる。ドット対については，後で詳しく述べるのでしばらく無視してほしい。以上の規則により，たとえば

```
(a)
3.14
"test"
 ((a 8)b c xy)
(* 1 2 3)
```

などは S 式である。
ためしに
user >(+ 1 2

と入力してリターンキーを押してみよう。何も起こらず止まっているのがわかる。これは S 式ではないからなので，さらに次の行に

3 4 5)

と入力すると，

15

と結果があらわれて，プロンプトが現れる。つまり，scheme は入力の S 式が完全でないあいだは改行などを無視して入力を待つのである。このように scheme は

1. S 式を読む。(read)

2. その S 式を評価する。(eval)

3. 評価の結果得られた値を画面に出力する。(write)

を無限に繰り返す。これを read-eval-write ループと呼ぶ。

1.7 コマンドライン・ヒストリー ヘルプ

ss のコマンドラインは、カーソルキーを使って修正できるようになっている。また、過去の入力を Page up,Page down キーを使って表示し、修正して実行することもできる。この機能は、scheme で書かれており、そのソースは **ss2lib** (通常 linux では/usr/local/lib/ss2lib/,mac では/opt/local/lib/ss2lib/) にある。変数 ∗**SSLIBDIR**∗ で確認できる。もし、xterm 以外の端末を使用していて、うまく動作しないときは ss2lib の関連ファイルを変更すれば対応可能である。

ss2lib/Doc には、ss の関数や機能の解説が書かれた多くの文書が置いてある。この本を読んだあとそれらに目をとおすと良いだろう。また、ss を起動した後

user >(help)

とするとそれらを読むことができる。キーワードを指定して、文書を選択することもできる。

user >(help 'format)

1.8 評価：eval

scheme がどのように S 式を評価するかを少し詳しく見よう。まずアトムの場合である。シンボル以外の数，文字列，文字，ベクトルなどについては単純にそれ自身が評価の値である。これを**即値データ**という。

user >"hello"
"hello"
user >

のようになる。シンボルの場合は少し難しい。たとえば

user >x

とするとエラーが起こる。シンボルは一般に任意の値（S 式）を格納することができる。これはシンボルを変数として使用することである。変数の評価はその変数が持っている値である。この例では，まだ変数 x に何も値が定義されていないのでエラーとなったわけである。

user >(define x 10)
x
user >x
10

ここで define は特殊形式と呼ばれるものの 1 つで変数の宣言をして初期値を指定する。x の値が 10 になっているので，今度は x の評価は 10 になる。さらに

user >(set! x "hello")
x
user >x
"hello"

ここで set!は変数の値を変更する特殊形式である。こんどは x の値として文字列が代入された。

[**変数の規則**] 変数は宣言されなければならない。宣言された変数は set! によって値が変更できる。値は任意の S 式

変数の宣言は define 以外にもいろいろあり，後で詳しく述べる。次にリストの評価である。まず空リストは即値データである。

user >()

()

[**注意**] 純粋の scheme では、() は即値データではなく、() を eval することはエラーになる。なので MIT scheme などを併用しているときは (define nil '()) として、nil を使うのが良いだろう。

空でないリストを評価するとき scheme は，まずその先頭要素を見る。これは

1. 特殊形式

2. 関数

3. マクロ

のいずれかでなければならない。特殊形式はその名のとおり，特殊な評価手順を踏む。英語の不規則動詞みたいなものである。特殊形式の数は少ない（２０以下）が，いずれも重要である。

define, set!, quote, let, let∗,if,letrec,lambda,begin

などがそうで，システムに最初から備わっている。使い方はだんだんと説明していく。マクロは有る意味で，リストの変換規則を表すもので，scheme はマクロによって与えられた S 式を変換してその結果の S 式を評価する。最初からシステムにある組み込みのマクロは

do, cond

などで，これらも特別な振る舞いをするので後で詳しく述べる。関数やマクロは自分で定義することもできる。さて，関数の場合は，評価の仕方は決まっている。規則動詞みたいなものである。システムに最初から入っている組み込み関数はたくさん（３００くらい）あり

+,-,∗,/, write, car,cdr,cons,

などが利用できる。よくつかうものは５０個程度である。さらにユーザは define 特殊形式を使って自分用の関数を定義できる。

scheme のプログラムの大部分は，関数を定義することである。

さて，変数 x には１０が入っているとしよう。

user >(+ x (∗ x 3))

40

この意味は誰でもわかると思うがなぜそうなるかを少し詳しく見よう。scheme は最初の行のリストを見て，その先頭が+という関数であることを見る。評価する関数は eval と呼ばれるので，評価するということを eval するともいう。さて，先頭が関数の場合 eval はそのリストの残りの要素（引数という）をすべて eval する。（ただしどの引数から eval するかの順序は決まっていない。）そして，その結果得られた値の列（順番とおりにしたもの）を先頭にある関数に渡す。関数は引数からある値を計算して結果を eval に返す。

今の場合 x と (∗ x 3) の２つが残りの要素（S 式）である。x はシンボルであるから，eval は x を評価してその値１０を得る。(∗ x 3) はどうだろうか？これはリストであり，その先頭 ∗ は掛け算の関数である。したがって eval はこのリストの残りの要素 x と 3 を eval する。その結果得られた１０と 3 を関数 ∗ に渡すと ∗ は３０を返す。こうして結局１０と３０を＋関数に渡すので＋は４０を返しそれがこのリストの値になる。このように，引数の中にまた関数があって，それがどんなに深いとしても eval は自分自身をその評価のために呼び出すことですべて評価されることになる。このような呼び出しを**再帰呼び出し**という。
これに対し，eval は特殊形式やマクロが与えられたときは，その引数を評価せずにそのままの形で特殊形式やマクロに渡すのである。そしてマクロの場合は，そのマクロから帰ってきた値（S 式）を eval した値が eval の返す評価値となる。

user >(define y (+ 2 3))

の場合，eval は何もせずに y と (+ 2 3) を define に与える。define は eval を呼び出して (+ 2 3) を評価し, 得られた値 5 を変数 y の値にする。もし仮に define が関数だとすると，eval は y を eval しようとするのでエラーになってしまう。エラーでないとしても define に渡されるのは y ではなくその値だからうまく動かないのである。

リストを評価するとき，その先頭は必ずしもシンボルである必要はない。実際には先頭が最初に eval されてその値が特殊形式，関数，マクロのどれかであればよいのである。実際，

user >define

#[special 2 0 define]

user > +

#[closure system 0 1 ((:T . +))]

などと入力してもらいたい。なにか得体のしれないものが表示されている。これが eval にとっては必要なもので，define や＋の実体 (特殊な形式のアトム）なのである。

user >(define y +)

y

user >(y 4 2)

6

のようになる。ここでシンボル y には，＋を eval した値，つまり足し算の関数そのものが入っている。

練習

(1) 計算 (1/3+2/5)∗21 を ss でしなさい。

(2) x=12.5,y=3.1, z=x-y,z∗z を ss でしなさい。

1.9 クォート

さて，次の例をみてみよう。

user >(define x '(sato 49 man))

x

user >x

(sato 49 man)

この例では変数 x の値をリスト

(sato 49 man)

にしている。これは sato という人のデータをリストにして表している。ここで最初の命令の中に**クォート** ' があることに注目しよう。これは S 式の定義にはなかったものである。クォートは read マクロの 1 種で，実は

'S 式

という形を読むと read 関数はこれを

(quote S 式)

という形に変換する。だから

user >'(sato 49 man)

(sato 49 man)

と

user >(quote (sato 49 man))

(sato 49 man)

は全く同じことになる。'S 式は (quote S 式) の省略形である。この quote は特殊形式で単にその与えられた S 式をそのまま返す働きをする。

(quote S 式)

を eval した結果は S 式そのものである。もし上の例でクォートを使わずに

user >(define x (sato 49 man))

としたら, scheme は sato という関数は存在しないというエラーを返すことになる。つまり

(define 変数 S 式)

という特殊形式は S 式を eval した値を変数に代入するのである。クォートはこの S 式の eval を停止する役目をすることになる。これは set!でも同様で

(set! 変数 S 式)

は，S 式を eval した値を変数に代入する。したがって，

user >(set! x 'y)

x

user >x

y

となる。このとき y をあらかじめ宣言する必要はない。値の代入されるシンボル（変数）はあらかじめ宣言しなければならないが，そうでなければそれは変数ではないのでかまわないのである。

練習

(1) ''(a b c) を評価すると?

(2) 変数 x の値を シンボル hoge と宣言した後でシンボル piyo に変更しなさい。

1.10 関数の定義

ユーザーは自分用の関数を

user >(define (add1 x)(+ x 1))

add1

user >(add1 10)

11

のように定義できる。一般形は

(**define** (関数名 引数 1 引数 2 ...) *special form*

S 式 1 S 式 2 ...)

の形である。引数 1，引数 2，... はシンボルであり**仮引数**と呼ばれる。この関数を eval するとき eval はその呼び出しで実際に与えられた引数 (**実引数**という) をすべて評価してそれらを仮引数の値にする。（これを変数の**バインド**という）そして，関数の本体を順に評価する。そして最後に評価した S 式の値がこの関数の返す値となる。

上の例の関数 add1 では，変数 x が仮引数として宣言されている。(add1 10) の呼び出しのときこの変数 x の値は 10 にバインドされる。このように仮引数は関数の中で自由に代入などができる変数になる。そして，関数の評価が終了すると，この変数 x のバインドはなくなり有効性は消滅する。このような変数を**局所変数**という。

引数の個数が一定でない関数の定義も可能である。

user >(define (lis . x) x)

```
      lis
user >(lis 1 2 3 4 5)
      (1 2 3 4 5)
```

ここで，1 行目で，関数 lis を定義していて，そこでは仮引数のあるべき所にドット対が現れている。この意味は後で詳しく述べる。この関数 lis はシステム関数 list と同じ動作をする。

練習

(1) x-1 を計算する関数 sub1 を定義せよ。

(2) x∗x を計算する関数 square を定義せよ。

(3) (20-1)^2 を sub1 と square を使って計算せよ。

1.11 let と let∗

関数の中で一時的にある変数を使用したいことはよくある。このようなとき特殊形式 let,let∗ が役にたつ。

```
user> (define (pita x y)
              (let ((s 0))
                 (set! s (+ (* x x) (* y y)) )
                 (sqrt s)
              )
           )
  pita
user> (pita 3 4)
  5
```

この例では，変数 s を計算の途中の結果を入れるために使っている。最後の sqrt は平方根を求める関数である。let の一般形は

(**let** ((変数 1 s1)(変数 2 s2)...) *special form*

```
        form1
        form2
        ...)
```

である。変数 l，... はこの let 式の中でのみ存在する局所変数で，s1,s2,... を eval した値がそれぞれにバインドされる。ただし s1,s2,.... をどれから先に eval するかはわからない。これに対し，**let∗** 形式では，s1,s2,... はその順に評価されバインドが行われる。let,let∗ は form1,... を順に評価して最後の評価値をその値として返す。

```
 user>(let* ((x 2)(y (* x x)))
         (+ x y)
      )

  6
```

のようになる。

関数の中で define を使って変数を宣言するのはあまり好ましくない。この理由は後でまた述べる。

局所変数は let または let∗ を使う。

練習

(1) $D = b^2 - 4ac$ を計算する関数 (D a b c) を定義せよ。

(2) (sqrt x) は x の平方根を返すシステム関数である。これを用いて方程式 $ax^2+bx+c = 0$ の 2 個の解をリストにして返す関数を書け。

1.12 car,cdr,cons

リストは，scheme の最も基本的な構造でそれを操作する関数は基本的に重要である。最初に

user >(car '(2 3 5))
2

関数 **car** は，1 個のリストを引数に取り，このようにリストの最初の要素を返す関数である。この例で，'(2 3 5) の eval は (2 3 5) というリストそのものであることに注意する。

user >(cdr '(2 3 5))
(3 5)

関数 **cdr** は，1 個のリストを引数に取り，このようにリストの最初の要素を除いた残りの引数からなるリストを返す関数である。

user >(cadr '(2 3 5))
3

cadr はリストの 2 番目を返す。これは

user >(car (cdr '(2 3 5)))
3

と同じである。つまり

cadr=car cdr

である。同様に，a と d の組み合わせで 3 個までの関数がシステムに用意されている。

caar, cddr,cdar,cdddr,..

ところで car はカーと発音し，cdr はクダーと発音する。これらの名前は昔の Lisp が初めて搭載されたＩＢＭの計算機の機械語に由来している。

car と cdr の逆をする関数 **cons** を見よう。

user >(cons 2 '(3 5))
(2 3 5)

つまり cons は，2 個の引数を取り，2 番目はリストである。結果は第 1 の引数が car であり第 2 の引数が cdr となるようなリストである。あるいは**スタック**と思ってもよい。つまり，いくつかのデータ（書類）があるときそれを机の上に積んでいくことを想像してみよう。(2 3 5) というリストは最初に 5 の書類を置き，その上に 3 を置き，さらに 2 を置いた状態と思う。1 番上の書類 2 はすぐに見ることができる。それが car である。2 をめくると (3 5) のリストになる。これが cdr である。これをスタックを

pop

するという。この上に 2 を置くのは cons である。これは **push** とも呼ばれる。

1 番上に 2 を置くと (3 5) は見えにくくなる。たとえば名前と年齢からなる次のリストを考えよう。

user >(define hito '((sato 49)(arai 43)))

これは 1 種のデータベースである。ここに新しい知識

(mati 55)

を付け加える。push には

user >(set! hito (cons '(mati 55) hito)))

とすればよい。もし arai の年が 54 であると修正したいとする。このとき hito をきちんと作りなおしても良いが，単にこれを push してみよう。

user >(set! hito (cons '(arai 54) hito)))

hito
user >hito
((arai 54)(mati 55)(sato 49)(arai 43))
このデータベースから arai の年を知りたいときは先頭から，見ていくことにする。すると arai は 54 とわかるので古いデータ 43 は無視されることになる。これを **shadow** というのである。
また，これは arai が 54 にバインドされていると見てもよい。これを pop すると 54 のバインドはなくなり，前のバインド 43 が復活することになる。これが変数のバインドの原理である。

練習
(1) x=(a (b c) d) のとき x に car,cdr,cddr,cdddr,cadr をした結果は何か。
(2) x=(a (b c)) のとき b を取りだす関数は何か。
(3) x=(a (b c)), y=(d (e f)) のとき (cons x y) の結果は何か。

1.13 list,append,reverse など

さらにリストの操作をする関数を説明する。
user >(list 1 2 3 4)
(1 2 3 4)
関数 **list** は任意個の引数を取りそれをリストにして返す。
user >(append '(1 2) '(3) '(4 5 6))
(1 2 3 4 5 6)
関数 **append** は任意個のリストを引数に取りそれらを連結したリストを返す。もし append でなく list を使うと
user >(list '(1 2) '(3) '(4 5 6))
((1 2) (3) (4 5 6))
となる。
user >(reverse '(1 2 3 4))
(4 3 2 1)
関数 **reverse** は 1 個のリストを引数に取りそれを逆順にして返す。
user >(list-ref '(1 2 3 4) 0)
1
関数 **list-ref** はリストと番号を引数に取り，リストの要素を返す。番号は先頭を 0 とする。
user >(length '(1 2 3 4))
4
関数 **length** はリストの長さを返す。

練習
(1) (list 'define (list 'sq 'x) (list '* 'x 'x)) の結果は何か。
(2) x=((3 4)) と宣言した後、x,car,append を使ってリスト (3 4 (3 4) 3 4) を作れ。

1.14 ドット対

cons の第 2 引数にアトムを与えてみよう。
user >(cons 1 2)
(1 . 2)
このようにドット対が現れる。次のような規則を考えよう。

(D1) cons A B=(A . B)

(D2) (A . ())= (A)

(D3) (A . (B ···))=(A B ···)

たとえば，このルールに従うと

(1 2 3)=(1 . (2 3))=(1 . (2 . (3)))=(1 . (2 . (3 . ())))

のようにリストはドット対で表現できることになる。これが cons の本当の意味である。つまり cons は 2 つの引数が与えられたときそのドット対を作るのである。もし第 2 引数がリスト (B C) だと

(A . (B C))=(A B C)

となり前に説明した push が行われることがわかる。また，car はドット対の最初を取りだす。cdr はドット対の第 2 要素を取り出す。ドット対表現は見ずらくわかりにくいのでリスト表現が好まれるが，本当はドット対がその構成原理となっている。この見方だと S 式は

(1) アトム（ただし () もアトムである。）は S 式

(2) 2 個の S 式 S1,S2 のドット対は S 式

と簡潔に定義できることになる。この簡単な文法表現が計算機言語の基礎として重要であることを認識したのが **J.McCarthy**(1959) であり，それはすぐに IBM 計算機に実装されたのである。これが、歴史的には最初の LISP で LISP1.5 と呼ばれる。
アトムに ()(つまり NIL）と lambda と変数のみを許す S 式の全体は純 Lisp とか原始 Lisp と呼ばれ**ラムダ計算**という数学のモデルとなる。

さて，

(f a b c)=(f . (a b c))

をもう一度みよう。

(lis 1 2 3)=(lis . (1 2 3))

これで，前にあげた

user >(define (lis . x) x)
lis
user >(lis 1 2 3)
(1 2 3)

がうまい定義であることがわかるでしょう！別の例

user >(define (lis2 x . y) (cons x y))
lis2
user >(lis2 1 2 3 4 5)
(1 2 3 4 5)

も，もうわかると思います。

(lis2 1 2 3 4 5)=(lis2 1 . (2 3 4 5))

と思うわけです。ただし，(D3) により

(a b . e)=(a . (b . e))

だと解釈します。この式を scheme に入力してみます。

user >'(1 . (2 . 3))
(1 2 . 3)

まさに scheme はドット対で動いています。関数 lis2 では引数が最低 1 個必要になることに注意します。このようにリストの最終部がドット対で終るとき、そのようなリストを**非真正リスト**という。

ところで

(car ()) や (cdr ())

はエラーです。() はドット対ではないからです。() はリストの終わりであり，始りでもある (しかもアトム) 特別な存在です。すべては無から始まる！

練習 x=1, y=2 とする。
(1) リスト (1 2) を cons,x,y, () を使って作れ。
(2) 非真正リスト (1 2 . 2) を同様に作れ。

1.15 真偽と条件式

scheme においては，真と偽は特別なアトム

#t,**#f**

で表される。これらは即値データである。また**#f 以外のすべてのＳ式は真である**と解釈する。#t は真の代表である。真偽値を返す多くの関数は？で終わる名前を持っている。例えば

(**eq?** x y) *procedure*

user >(eq? 'x 'x)

#t

eq?は，２つのデータがメモリー内のアドレスの意味で等しければ（すなわち完全に同じものであるとき）真を返す。これはシンボル, 小整数、文字の比較に使う。

user >(eqv? 2 2)

#t

eqv?は，eq?で真であるか等しい数のとき真を返す。

user >(equal? '(1 2) '(1 2))

#t

equal?は，eqv?で真か，Ｓ式として等しいとき真を返す。equal?は汎用的であり，eq?は高速である。シンボルはシステム内で完全に管理されていて同じ名前のものは 1 つしかない。ss では文字と小整数（62bit または 30bit) も eq?で比較できる。これに対し文字列やリストは，内容が同じであっても別のメモリー上にあることが多いのである。

数の比較には

=,**<**,**>**,**<=**,**>=**

などが使える。これらは 3 個以上の引数でも良い。
また多くのデータ型判定用の関数がある。

(**null?** x) *procedure*

user >(null? ())

#t

null?は () のときだけ真を返す。null?の反対に **pair?**はドット対に対して真を返す。これらは，リスト処理の終わりや継続の判定で多用される。

symbol?,**number?**,**string?**, **char?**,**vector?**

などはデータの型がそうなら真を返す。

(**not** x) *procedure*

は論理を反転する。

練習

(1) (eq? '(1) '(1)) は?
(2) (eq? 1/2 1/2) は?
(3) (symbol? #\A) は?

1.16 if

if は特殊形式で
(**if** 条件式 form1 [form2]) *special form*

条件式が真であれば form1 を評価し，偽であれば form2 を評価します。帰す値は評価した式の値となる。省略形

(if 条件式 form1)

も使える。これは，条件式が真なら form1 を評価してその値を返す。ただし条件式が偽のときは undefined を返す（偽ではない）ので注意が必要。

例.

```
(define (hantei x)
         (if (>= x 60) 'gokaku 'rakudai)
)
```

ところで if の実行文が 1 行ですまないときは特殊形式
(**begin** f1 f2 ...) *special form*

を使うとよい。これは let の変数宣言の無い形と思えばよい。f1,f2, ... が順に評価されて最後に評価した値が返ってくる。
(**when** 条件 *macro*

```
form1
...
)
```

条件が真であれば、form1 ... を順に評価して最後に評価した値を返す。

これは

(if 条件 (begin form1 ...))

と等価なマクロである。
次に，
(**and** f1 f2 ...) *macro*

f1 を評価し，真なら f2 を評価し,... と偽が出るまで評価を続ける。すべて真なら，最後の評価した値を返す。

同様に，
(**or** f1 f2 ...) *macro*

真が出るまで評価を続ける。すべて偽なら偽が値として帰る。

and, or は単に論理値を出すのではなく, 実行の流れも制御していることに注意してください。
cond は，場合わけに用いる構文です。
(**cond** (条件 1 f1 f2 ..) *macro*

```
(条件 2  f1 f2 ..)
 ...
(else  f1 f2 ..)
)
```

条件の中で最初に真になるものがあれば，それに続く f1 f2 ... を評価して最後に評価した値を返す。最後に else とあるのは，例外の処理でそれまでのすべての条件が偽でも else は真でありその後の f1,f2,... が必ず実行されることになる。これは決まりではないが，cond 構文は else(または#t) で終わるのが望ましい。

例 **1.1.**
```
(define (->string x)
   (cond ((string? x) x)
         ((char? x) (make-string 1 x))
         ((symbol? x ) (symbol->string x))
         ((number? x) (number->string x))
         (else (error E_notstr x))
   )
)
```

これは与えられたデータを文字列に変換する関数である。ss において

```
(->string 'hello)
(->string 3.14)
(->string #\a)
(->string '(1 2))
```

などを実行してみてほしい。**error** は ss 固有のシステム関数でその引数はエラー番号とオブジェクトであり，error 状態を起こす。各番号はシンボル E_xxxxx として定義されている。詳しくは Doc を見ること。

ところで when,and,or，cond は ss ではマクロとして定義されています。実際, これらは if を使って簡単に書きなおせるからです。

練習

(1) x が数のとき x を返し、それ以外は#f を返す関数 (pass-number x) を定義せよ。

(2) 0<=x<60 なら”不合格”, 60<=x<=100 なら”合格”を返し、それ以外はエラー E_illegal とする関数 (hantei x) を定義せよ。

1.17 再帰呼び出し

英語で recursive call と呼ばれる。一般に関数が自分自身を呼び出すことである。たとえば n の階乗は

n!=n∗(n-1)∗....∗2∗1

であるが，これは

```
n!=1 (if n=0)
  =n*(n-1)!
```

と定義することができる。このように定義の中に自分自身の入っている形を再帰的定義という。このとき，自明なときの値が指定されていること。そして，右辺の自分の呼び出しがより簡単な場合であることが必要である。問題によっては再帰的な定義しかできないことはしばしば起こる。scheme や最近の言語は，この再帰呼び出しが許されているのでこのような関数を定義するのは容易である。

```
(define (factorial n)
   (if (= n 0) 1
          (* n (factorial (- n 1)))
   )
)
```

定義をそのままプログラムにできるのである。このような関数の動作を確認するために

(trace factorial)

のように **trace** 関数を使う。そして

user >(factorial 5)

とすると，factorial が次々と呼び出されて答えが得られるのを見ることができる。trace をやめるには

user >(untrace factorial)

とすればよい。もう 1 つ例をあげよう。list の中の数だけを取り出したいとしよう。

user >(take-num '(x 2 y 4))

(2 4)

のような関数を定義したい。これは次のように定義できる。

```
take-num list=  ()   ;; if list = ()
             =  (cons num (take-num (cdr list) )  ;; if list = (num ...)
             =  (take-num (cdr list))   ;; if list = (not-num ...)
```

これをすなおにプログラムする。

```
(define (take-num x)
   (if (null? x) x
       (if (number? (car x))
               (cons (car x) (take-num (cdr x)))
               (take-num (cdr x))
       )
   )
)
```

練習

(1) (factorial 5) を trace しなさい。

(2) $1+2+\cdots+n$ の値を返す関数 (nsum n) を作れ。

(3) 2 項係数 $\binom{n}{r}$ をパスカルの 3 角形を用いて再帰的に求める関数 (combi n r) を定義せよ。

(4) 要素が数だけからなるリストから、正の数だけを取りだしたリストを返す関数 take-positive を定義せよ。

(5) 要素が数だけからなるリストの各要素の 2 乗の和を求める関数 (square-sum lis) を定義せよ。

1.18 while, do

繰り返し（ループ）の構文である。

while は単純で

(**while** 条件 *macro*

```
  f1
  f2
  ...
  )
```

条件が真であるあいだ本体 f1,f2,... の評価を繰りかえす。

```
(while (pair? x)
    (format #t "~s~%" (car x))
    (set! x (cdr x))
 )
```

はリスト x の要素を画面に出力する。

do の一般形は

(**do** ((変数 1 初期値 1 更新値 1) *macro*

```
       (変数 2 初期値 2 更新値 2)
       ...
      )
      (終了条件  f1 ...)
     body1
     body2
     ...
 )
```

最初に初期値 1,... が評価されて変数 1,... にバインドされる。（ただし評価の順序は不定である。）次に終了条件が評価されて，それが偽ならば本体が順に評価される。 2 回目は更新値 1,... が評価されて変数に代入される。以下終了条件判定して同様に反復する。どこかで終了条件が真であれば f1,.... を順に評価して最後に評価した値が do の値となる。ただし，更新値の式は省略してもよい。n の階乗をこれを使って書くと

```
(define (factorial n)
  (if (= n 0) 1
     (do ((s n (* s n)))
         ((= n 1)  s)
         (set! n (- n 1))
     )
  )
)
```

while,do ループは，再帰呼び出しより一般に高速である。

更新値の式を書くことで，do の本体が短くなるが，かえってわかりにくいことが多い。

```
(define (factorial n)
  (if (= n 0) 1
     (do ((s n ))
         ((= n 1)  s)
         (set! n (- n 1))
         (set! s (* s n))
     )
  )
)
```

のように，更新を do の本体でするほうをお薦めする。筆者は、do の代わりに let∗ と while を使うことが多い。

練習

前のセクションの練習 (4),(5) を do を使って書け。

1.19 format 関数

画面出力のとき write や display だけではやや単純すぎる。C 言語の printf に相当するのが format 関数である。

(**format** output 制御文字列 obj1 obj2 ...) *procedure*

第 1 引数 output は出力先の指定で#t は画面, #f は文字列として出力を意味する。ファイルへの出力は後述する。制御文字列の中で

~a これは obj を display スタイルで印字

~s これは obj を S 式として印字

~% 改行

~t tab

などが使える。~a,~s があるときは対応する個数の引数 obj が必要である。

user >(format #t "~a is ~a years old. ~%" "sato" 49)
sato is 49 years old.
user >(format #f "~a~a" "sato" "sadao")
"satosadao"

詳細は Doc にある。

練習

(1) 3 個の数 a,b,c を引数に取り a+b=c のときは"正解"と表示し、そうでないときは a,b と正しい答 (a+b) を表示する関数を定義せよ。

(2) 九九の表を画面に表示する関数 (kuku) を作れ。

1.20 グローバル変数

ss のプロンプトに対する入力を**トップレベル**の入力と言う。トップレベルで define された変数はすべての関数から参照，変更ができるので**グローバル変数**と呼ばれる。このような変数はプログラム中で，特別な役割をしているので，間違った代入などを避けるために

user >(define *n* 19)

のように，* で囲った名前にする習慣である。ss は大文字、小文字を区別するので大文字を使うのも良いだろう。x,y などの簡単な名前をグローバルに使うと無用な混乱をひきおこす。関数の中で define を使うとどうなるか試してみよう。

```
(define *n* 19)
(define (test1)
   (define *n* 20)
   (test2)
   (format #t "test1 ~s~%" *n*)
   (set! *n* 0)
 )
 (define (test2)
   (format #t "test2 ~s~%" *n*)
   (set! *n* 50)
 )
```

(test1) を実行すると

```
test2 19
test1 20
```

test2 の見ている *n* はトップレベルの *n* であることがわかる。このためトップレベルの *n* は 50 に変化している。0 にはならない。関数 test1 の中の 4 行は test1 の**レベル**を構成している。test1 のレベルにおいては *n* はその内部において 20 で，最後に 0 になる。しかし、これは外側で定義されたグローバルな *n* には影響しない。したがって，define は局所変数の定義になっているのだが混乱をさけるために let,let* を使い，変数を関数内で define しないことをお奨めする。let や，let* のほうが，変数の使用範囲が明確だからである。なお、scheme では、局所的な define は他のすべての実行文よりも前におかなければならないと規定されている。

なお test1 の中に関数 test2 の定義を置くことはかまわない。このとき，test2 はグローバルではない，test1 のレベルの関数となり局所関数になる。

```
(define *n* 19)
(define (test1)
   (define *n* 20)
   (define (test2)
      (format #t "test2 ~s~%" *n*)
      (set! *n* 50)
    )
   (test2)
   (format #t "test1 ~s~%" *n*)
    (set! *n* 0)
 )
```

今度は test2 は test1 の *n* を見ることになる。局所関数 test2 を test1 の外から呼び出すことはできない。

1.21 数

ss で使える数は整数、有理数（無限長）と実数、複素数である。scheme の仕様 (R5RS) では、数は exact か inexact であるが、ss では単に整数、有理数を exact としている。(**number?** x) *procedure*

(**integer?** x) *procedure*

(**rational?** x) *procedure*

(**real?** x) *procedure*

は型の判定である。複素数は、

+i, -i, 1/2+i3/4, 0.1-4.5i

のように表記する。i はシンボルであって、虚数には+i,-i を使う。(**zero?** x) *procedure*

(**positive?** x) *procedure*

(**negative?** x) *procedure*

(**odd?** x) *procedure*

(**even?** x) *procedure*

(**max** n1 ...) *procedure*

(**min** n1 ...) *procedure*

(**abs** x) *procedure*

(+ n1 ...) *procedure*

(**-** n1 ...) *procedure*

(∗ n1 ...) *procedure*

(/ n1 ...) *procedure*

(**gcd** n1 ...) *procedure*

(**lcm** n1 ...) *procedure*

は，名前から理解できると思う。整数の割り算は有理数になる。実数が混ざった演算の結果は実数になる。整数型 (結果も）割り算は
(**quotient** n1 n2) *procedure*

である。剰余計算は
(**modulo** n1 n2) *procedure*

実数の整数化は
(**floor** x) *procedure*

や **round**,**ceilling**,**truncate** を使う。

複素関数

(**exp** z) *procedure*

(**log** z) *procedure*

(**sin** z) *procedure*

(**cos** z) *procedure*

(**tan** z) *procedure*

(**asin** z) *procedure*

(**acos** z) *procedure*

(**atan** z) *procedure*

(**sqrt** z) *procedure*

(**expt** a b) *procedure*

a**b, a の b 乗

以上は、複素数引数でもよく、その定義は Common Lisp 仕様に従っている。

(**number->string** x) *procedure*

数-> 文字列

(**string->number** str) *procedure*

文字列-> 数

練習

(1) log(-1) を求めよ。

(2) gcd を使わずに、2 個の整数の最大公約数を求める関数を定義しなさい。

1.22 文字

文字データの処理関数である。

(**char?** c) *procedure*

(**char=?** c1 c2) *procedure*

(**char<?** c1 c2) *procedure*

(**char>?** c1 c2) *procedure*

(**char<=?** c1 c2) *procedure*

(**char>=?** c1 c2) *procedure*

(**char-alphabetic?** c) *procedure*

(**char-numeric?** c) *procedure*

(**char->integer** c) *procedure*

(**integer->char** c) *procedure*

(**char-upper-case?** c) *procedure*

(**char-lower-case?** c) *procedure*

(**char-upcase** c) *procedure*

(**char-downcase** c) *procedure*

練習
(1) 文字#\a,#\3,#\ 電 について char-alphabetic?,char->integer の結果は何か。
(2) (integer->char 13) は何か。

1.23 文字列

文字列データの処理関数である。
(**string?** x) *procedure*

(**make-string** n char) *procedure*

(**string** char1 ...) *procedure*

(**string->list** str) *procedure*

(**list->string** char-list) *procedure*

(**string-length** str) *procedure*

(**string-ref** str index) *procedure*

(**string-set!** str index char) *procedure*

(**string-append** str1 ...) *procedure*

(**string-copy** str) *procedure*

(**string-index** str1 str2) *procedure*

str1 を str2 の中から捜し，あればその位置を返し，無ければ#f を返す。

string-length,string-ref などの関数は日本語に対応している。

(**substring** str start end) *procedure*

(**string<?** str1 str2) *procedure*

他に、**string>?**,**string<=?**など。

(**string-split** str [delimiter-str]) *procedure*

user >(string-split "/usr/local/bin" "/")

("usr" "local" "bin")

(**string-upcase** str) *procedure*

(**string-downcase** str) *procedure*

(**tk-string** x1 ...) *procedure*

x1,x2,... は文字列、文字、シンボル、数のどれかである。この関数は、引数を文字列に変換して連結した文字列を返す。

練習

(1) 文字列が"http"を含むとき http から始まる以降の部分を返しそうでなければ#f を返す関数を定義せよ。

(2) 英数字だけからなる文字列、たとえば x="ab12cd3"が与えられたときリスト ("ab" "12" "cd" "3") を返す関数を定義せよ。

1.24 vector

vector データの処理関数である。

(**vector?** x) *procedure*

(**make-vector** size init-data) *procedure*

user >(make-vector 3 0)
#(0 0 0)

(**vector** x1 ...) *procedure*

(**vector-length** vec) *procedure*

(**vector-ref** vec index) *procedure*

(**vector-set!** vec index obj) *procedure*

(**vector->list** vec) *procedure*

(**list->vector** lis) *procedure*

(**vector->copy** vec) *procedure*

ss では matrix も使えるが、それは直接 Doc を見てほしい。

練習

(1) ベクトル#(1 2 ... 52) を作って返す関数 (make-card) を作れ。

(2) ベクトル v の第 i 要素と第 j 要素を入れ換える関数 (vec-swap v i j) を作れ。

(3) 文字データのベクトル v と 1 個の文字 c を引数として、v に含まれる c の個数を返す関数 (count-v-char v c) を作れ。

1.25 ファイルからの入力

ためしに test というファイルを作っておきます。内容は

```
'test  3.14
 '( 1 2 3)
```

だとしましょう。

user >(define in (open-input-file "test"))
in

これで変数 in にはファイル test から入力を行うための**ポート**という特殊なデータが入る。
次に

user >(read in)
(quote test)

user >(read in)
3.14
のように，**read** 関数で読むことができる。read 関数は
(**read** [port]) *procedure*

port から S 式を 1 つ読み出す。このとき port 内部にあるポインタが読み出した S 式の次を指すようになる。もし port を指定しなければ read 関数はキーボード（標準入力）から S 式を読む。

さらに
user >(read in)
(quote (1 2 3))
user >(read in)
#<eof>
ファイルの終わりに達したわけだが，これの判定には
(**eof-object?** x) *procedure*

を使う。
user >(eof-object? (read in))
#t
そして，終了したら必ず
(**close-port** port) *procedure*

を使って
user >(close-port in)

と，port を閉じなければいけない。以上を使って load 関数のまねをする myload 関数を書くと

```
(define (myload file)
  (let ((in 0))
    (set! in (open-input-file file))
    (do ((s 0))
        ((eof-object? s) (close-port in))
        (set! s (read in))
        (write (eval s))
    )
))
```

となる。(myload "test") を実行してみてください。S 式以外のデータを読むときは
(**read-char** [port]) *procedure*

read-char は 1 文字を読み文字データを返す。

(**read-line** [port]) *procedure*

read-line は改行までの 1 行を読み文字列を返す。

文字列を入力ポートにするには
(**open-input-str** str) *procedure*

を使えば同様にできる。

練習

(1) 与えられた file のすべての行を画面に出力する関数 (cat file) を作れ。

(2) file の行で word を含んでいる行の行番号とその行を画面に出力する関数 (grep file word) を作れ。

(3) 2 つの file1,file2 を引数としてその内容が同じであれば#t, 異なっていればそれが何文字目かを返す関数 (cmp-file file1 file2) を作れ。

1.26 ファイルへの出力

出力ポートは

(**open-output-file** file) *procedure*

(**open-append-file** file) *procedure*

(**open-output-str** str) *procedure*

を用いる。これらは，ファイルまたは文字列の出力ポートを作る。書きこみのためには

(**write** obj [port]) *procedure*

(**display** obj [port]) *procedure*

(**format** port format-string obj1 ...) *procedure*

(**write-char** char [port]) *procedure*

などを使えばよい。終了したら (close-port port) を必ずする。なお、文字列出力のときは、その最終結果の文字列は close-port の帰り値として得られる。

次の例はある S 式をファイルに書き出すための関数である。

```
(define (out-s file sobj)
    (let ((out 0))
        (set! out (open-output-file file))
        (format out "~s~%" sobj)
        (close-port out)
    )
)
```

(out-s "test2" (fact 100)) のように使う。

ところで画面などの出力では，普通**バッファリング**動作と言ってデータがある程度まとまった所で (あるいは改行コードで) 実際の出力が実行される。このため出力結果がすぐには見えないことがある。このようなとき強制的にバッファーの中身を処理させるには

(**flush** [port]) *procedure*

を使う。たとえば

```
(define (scheme)
  (do ((s 0))
      ((equal? s '(bye)) 'end)
      (display "scheme> ")
      (flush)   ;;;;   プロンプトを強制出力
      (set! s (read))
      (if (equal? s '(bye))
             (format #t "See you!~%")
             (format #t "value: ~s~%" (eval s))
      )
  )
)
```

は read-eval-write ループのまねをする。

練習

(1) 1 から 100 までの sqrt の値をファイル"sqrt100.txt"に書き出してください。

(2) 文字列出力を用いて file の内容を 1 個の文字列にして返す関数 (file->string file) を定義せよ。

(3) file1 を file2 にコピーする関数 (copy-file file1 file2) を作れ。

1.27 ラムダ式

関数とは何だろうか？たとえば

f(x)=x∗x

は，２乗の関数だが，これは

x -> x∗x

の対応であり，名前 f は本質ではない。関数そのもの（実体）を表現するのが **lambda** 式である。scheme では，これを

(lambda (x) (∗ x x))

と表現する。lambda 式を eval すると関数実体（特殊なデータになる）が 作られる。

user >(define square (lambda (x) (∗ x x)))

square

user >(square 3)

9

ここでは，lambda 式を eval した結果がシンボル square にバインドされている。結果として，define による関数定義と同じことが起こっている。

lambda 式の一般形は

(**lambda** 仮引数リスト form1 ...) *special form*

である。仮引数リストは上の例のように，引数の個数が決まっていれば普通のリストである。もし任意個の（0 以上）の引数を取りたいときは

(. x)

または単にシンボル

x

を書く。

(define lis (lambda x x))

は，以前書いたのと同じで list 関数と同じことになる。同様に

(define lis2 (lambda (x . y) (cons x y)))

は 1 個以上の引数を取る例である。また
user >((lambda (x)(∗ x x)) 4)

16

のようなことも許される。
1 つの応用例として map 関数をとりあげよう。map 関数は
(**map** func lis-1 ... lis-k) *procedure*

のように使う。ただし func は k 個の引数をもつ関数でそのとき lis-1 ... lis-k は k 個の同じ長さのリストである。map は k 個のリスト先頭からなる組に対して func を適用し，以下同様にリストの長さの分だけ func を適用してそれらの結果を 1 個のリストにして返す。
user >(map number? '(2 x 1 y t 3))

(#t #f #t #f #f #t)

のようになる。さて数を要素とするリストが 2 個あってそれらを各々加えたいとしよう。
user >(map + '(1 2 3) '(5 3 6))

(6 5 9)

うまく動く。それでは 2 乗の和を作りたいとしたらどうするか
user >(define (square-add x y) (+ (∗ x x)(∗ y y)))

square-add

user >(map square-add '(1 2 3) '(5 3 6))

(26 13 45)

とできる。しかし，もし square-add がここでしか使わない関数だとするとこれはいかにも大げさではないだろうか？
user >(map (lambda (x y) (+ (∗ x x)(∗ y y))) '(1 2 3) '(5 3 6))

(26 13 45)

のように lambda を使える。lambda 式はプログラミングの中ではおおむねこのような局所的な関数定義として多く用いられる。ただ，以前に書いたように lambda 式は Lisp の本質的のところ（関数実体）からきている形なのである。define による関数定義の形式はここでの lambda 式によるものの簡略形である。

練習

(1) 関数 e^{-x^2} を lambda 式で表せ。

(2) 数からなるリスト lis が与えられたとき、その絶対値からなるリストを返す関数 (list-abs lis) を作れ。

(3) 数からなるリスト lis が与えられたとき、それらを 2 で割った余りからなるリストを返す関数 (mod2-list lis) を作れ。

1.28 member, remove, assoc

リストの中からある要素を見つけたいときに使うのが **member** である。

user >(member 'a '(4 c a b 8 a))

(a b 8 a)

見つかったときは，その要素から終わりまでのリストを返す。見つからなければ#f を返す。member は要素の比較に equal?を用いる。同様に **memq,memv** は eq?,eqv?を使う関数である。

remove は

user >(remove 'a '(4 c a b 8 a))

(4 c b 8)

のようにリストの中の一致する要素をすべて取り除いたリストを返す。remove は equal?を使い，**remq**,**remv** は eq?,eqv?を使う。

ドット対からなるリスト, たとえば

user >(define hito '((sato . 49) (arai . 54)))

のようなものは**連想リスト (association list)** と呼ばれるデータベースである。ドット対の car 部はキー，cdr 部はその値と呼ばれる。関数 **assoc** はキーからそのデータを見つける。

user >(assoc 'sato hito)

(sato . 49)

のように使う。assoc は equal?, **assq**,**assv** は eq?,eqv?を使う。見つからなければ#f を返す。

練習

(1) member は使わずに、自分用の mymember を書け。
(2) myremove を書け。
(3) myassoc を書け。
(4) 本分中の hito データベースから年令を取りだす関数 (hito-age data man) を書け。

1.29 apply と eval, funcall

関数 apply を取り上げる。apply は

(**apply** 関数 a-1 ... a-n) *procedure*

の形で使う。最後の引数 a-n はリストでなければならない。変数 x の値は 1 0 としよう。

user >(apply + x 2 3 ())

15

user >(apply + x '(2 3))

15

つまり，

(a-1 ... . a-n)

を引数リストとして関数が呼び出される。同じことは，**eval** だと

user >(eval (list + x 2 3))

6

user >(eval '(+ x 2 3))

6

のようになる。厳密には eval は 2 引数関数であり、第 2 引数として環境を取るのだがこれは上級編で述べよう。
(**funcall** fun p1 ...) *procedure*

funcall は間接的な関数呼び出しである。
user >(define sum '+)

とすると、sum の値はシンボル+であって、足し算関数そのものではない。従って
user >(sum 3 4)

とするとこれはエラーになる。このようなとき funcall を使うと
user >(funcall sum 3 4)
7
と正しく実行できる。funcall の第 1 引数は、関数実体に到達するまで繰り返し評価される。そして、残りの引数が eval されて関数に渡される。scheme では、関数実体は特殊なアトムであり、変数に代入することも、それを引数としてもあるいは、それらのリストや vector を作ることなどもすべて自由なのである。

練習

(1) 2 つの同じ長さの数だけからなるリストに対してその内積を計算する関数 (naiseki lis1 lis2) を定義せよ。

(2) キーボードから数と演算子 (+,-,*,/) を入力して、その計算結果を返す関数 (compute-input) を作れ。例えば、入力が

```
3 5 6 +
```

ならば結果は 14 となる。

1.30 古典マクロ

ss ではマクロとして、CommonLisp などで採用されていた古典マクロと R5RS 仕様の syntax macro の両方を使うことができた。しかし、ver2.0 以降では古典マクロは実際には存在せず、形式として残っている。以下はこの古典的マクロの形式について説明する。

マクロは関数とは異なる評価手続きを定義したいときに使う。また命令の省略形として使う。次を考えてみよう

```
user >(define (def0 x ) (list 'define x 0))
  def0
user >(def0 'y)
  (define y 0)
user >(eval (def0 'y))
```

実は、これは (本当は) エラーになる。eval は、新しい変数束縛を生成することを禁止されているからである。

ここで def0 は (define y 0) という S 式を返している。これを eval することでこの S 式を実行したかったのである。これをマクロ定義してみよう。古典マクロは **define-macro** 特殊形式を使う。

```
user >(define-macro (def0 x) (list 'define x 0))
  def0
user >(def0 y)
```

y
user >y
0

今度は eval 無しで，実行できた。マクロは，与えられた S 式を別の S 式に変換する手続きであるが，eval はそのできあがった S 式を自動的に eval するのである。またマクロの呼び出しでは y にクォートが無いことにも注意されたい。(def0 y) の eval は以下のように実行される。まず eval は def0 が 1 個の引数を持つマクロと知る。このとき実引数 y を関数の評価のときとは違って，eval せずにそのまま def0 の仮引数 x にバインドする。x の値は y である。そして eval はマクロ定義の本体を順に評価する。ここでは 1 行しかないので (list 'define x 0) が評価されて eval は

(define y 0)

という S 式を得る。そこで eval はこの式を評価してその値を返す。(この eval は、ユーザー関数の eval ではない。) マクロ本体の評価で最後に得られた S 式を評価して終わるのである。このようにマクロを使うと特殊形式と同じような手続きを作れることになる。(define x 値) と同じ動作をするマクロは

```
(define-macro (mydefine x s) (list 'define x s))
```

となる。マクロがうまく作れたかどうかは

user >(macro-expand-rec '(mydefine y 'set) ())
(define y (quote set))

のように **macro-expand-rec** 関数を使ってテストできる。この関数の第 2 引数は環境であり上の例では、簡単に () としている。

基本的にはこれでどんなマクロも書けるが，複雑なマクロを容易にかくための read-マクロがある。これはバッククォート ‘から始るので，**バッククォートマクロ**という。

user >(define x 'y)

user >(define lis '(1 2 4))

user >‘(hoge ,x = ,lis ,@lis)
(hoge y = (1 2 4) 1 2 4)

実際に打ちこんでためしてもらいたい。‘の次はリストであり，その中でコンマ”，” とコンマアットマーク”,@”が使用できる。コンマのない部分は，そのままなのでバッククォートはクォートに似ている。ただし，コンマのあとの S 式は評価されたものが入る。コンマアットマークの次の S 式の評価は必ずリストでなければならない。そしてこのリストは上のように周りと append される。内部的には，これらのマクロはシステム内部で適当な list,append ,quote,eval などを使った S 式に変換されるのである。(ただし，ss ではそのようすは隠されている。)

さて，上の 2 つの例をバッククォートで書いてみよう。

(define-macro (def0 x) ‘(define ,x 0))

(define-macro (mydefine x s) ‘(define ,x ,s))

とすっきりする。push マクロは

```
(define-macro (push x list)
  `(begin
     (set! ,list (cons ,x ,list))
     ,list
   )
 )
```

と書ける。

user >(define x '(2 3))

user >(push 4 x)

user >x
(4 2 3)

のように使える。pop は

```
(define-macro (pop list)
  (let ((var (gensym)))
  `(let ((,var ()))
      (set! ,var (car ,list))
      (set! ,list (cdr ,list))
      ,var
    )
 ))
```

と定義できる。ここで **gensym** 関数は新しいシンボルを作るための関数でそれを (car list) を一時的に保存するために使っている。こうしないで var に固定した名前を使うと呼び出しのときに与えられる名前とたまたま衝突する可能性があるからである。上の例に続けて

user >(pop x)

user >x
(2 3)

となることを確認して欲しい。また pop を macro-expand-rec で展開してみると良い。

古典マクロは強力ではあるが、変数名の衝突の問題がつねにある。また、意味がわかりずらいという欠点もある。R5RS では、パターンマッチングを用いた syntax-macro が定義されていてこれには変数の衝突がないことが保証されている。syntax-macro は上級編で述べよう。

練習

(1) 変数 x,y の値を入れ換えるマクロ (swap x y) を書け。

(2) when マクロを書いてください。

1.31 静的クロージャ

ちょっと特殊なテクニックを紹介しよう。

```
(define (make-counter)
  (define x 0)
  (lambda() (set! x (+ x 1)) x)
)
```

と定義して,

user >(define c1 (make-counter))
c1
user >(define c2 (make-counter))
c2
user >(c1)
1

user >(c1
) 2
user >(c2)
1

おわかりだろうか？関数 make-counter の返す値は lambda 式

(lambda() (set! x (+ x 1)) x)

を eval したものである。この式の中の変数 x は，make-counter の中で define されている。この x は，make-counter のレベルにあり，外からは見えない。lambda 式を eval すると，単にその関数定義だけではなく，そのレベルにおける関数評価の環境も付け加えられたものができる。これを**静的クロージャ**と呼ぶ。今の場合，make-counter 内の局所変数 x も閉じ込められたクロージャが c1 にバインドされる。(c1) が呼び出されるたびに，この閉じ込められた x の値は 1 増加して x の値が (c1) の返り値となる。(define c2 ...) の呼び出しで，再び define 文により変数 x のバインド（0 である。）が作られ，これが c2 に閉じ込められる。このため，カウンタ c1,c2 は独立して動くことになる。通常，局所変数は関数の呼び出しが終われば消滅するが，クロージャの中ではずっと生き続けることができる。変数の有効な文脈的な範囲をスコープと言い，参照の発生する時間をエクステントと言う。クロージャに閉じ込められた x は静的なスコープと無限エクステントを持っている。カウンタを作る最も簡単な手段はグローバル変数を使うことであるが，このテクニックを使うと，たくさんのグローバル変数を定義する必要がなくなる。上の例では，そのかわりグローバル関数が増えることになるが，これも 1 個のカウンター関数を作って, (count 1),(count 2) のように呼び出すように定義することもできる。

```
(define (create-counter n)
   (define cts ())
     (do ()
      ((= n 0))
      (set! cts (cons (make-counter) cts))
      (set! n(- n 1))
     )
   (lambda (k) ((list-ref cts k)))
 )
 (define counter (create-counter 20))
```

(counter 0) から (counter 19) まで 2 0 個の counter が使用できることになる。

これを loop の中などに置いて，実行回数を計測できる。

練習

1.(counter -1) とするとすべてのカウンターが 0 に reset されるようにしなさい。

1.32 Lisp の内部

Lisp は文法が簡単なので，C 言語などを用いて自分用の Lisp を作ることも難しくない。やや単純化したモデルで説明する。コンスセルは，64bit のポインタを 2 つ入れられるメモリー上にある箱である。セルに型を付けるために 32bit の TAG をつけて 160 ビットをセルと呼ぼう。

TAG	CAR	CDR

空リスト（）はアクセス不可能なアドレス（たとえば 0) または特別なセルにする。セルは適当な構造体で定義し，CAR,CDR はそのアクセスマクロとする。リストを構成するコン

スセルは TAG を 0 とし，整数は TAG=1 で 64bit がその値とする。シンボル，文字列なども適当な TAG 値を設定して 64bit に格納することにする。最初にシステムは十分大きなメモリーを確保して，セルの freelist を作る。先頭のセルのポインタをグローバル変数 freelist に入れて，先頭のセルの CDR は次のセルのポインタとする。以下同様にして，最後のセルの CDR は () にする。このとき関数 cons(x,y) は

```
f=freelist;
freelist=CDR(f);
CAR(f)=x;
CDR(f)=y;
return f;
```

と書ける。もし，キーボードから整数が入力されたら，まず cons を呼び，そして TAG を 1 に変更する。シンボルは別のテーブルで管理し，同じ名前のものは 1 つだけであるようにする。シンボルのバインドは 1 個の association リスト (env という変数に先頭アドレスを入れよう。）にする。従って，シンボル s に値 v をバインドする bind(s,v) は

```
env=cons(cons(s,v),env);
```

と書ける。さて，freelist が () になってしまったらどうするか？システムはシンボルに登録されているセル，env にあるセル，それらの CAR,CDR が指しているセルに再帰的にマークをつける。TAG にはマーク用の 1bit を確保しておく。そして，すべてのセルの空間を調べてマークのないセルをリストにする。つまり，最初に見つかったマークのないセルのポインタを，freelist に入れ，このセルの CDR に次の free なセルのポインタを入れるを繰り返す。これをゴミ集め，**ガベージコレクション**と呼ぶ。終了したらすべてのセルのマークをクリアする。

システムは起動したとき，特殊形式やシステム関数の登録を行う。たとえば cons だと，シンボル cons を作ってテーブルに登録する。次に cons を呼んで，セルを 1 つ確保して，その CDR に内部関数 cons のポインタを入れ，CAR には引数が 2 個などの情報をいれる。TAG をシステム関数の値に変更してからそれをシンボル cons にバインドする。このあとシステムは，read-eval-write の無限ループを開始する。

1.33 letrec

letrec の構文は let と同じである。

(**letrec** 局所変数のバインド form1 ...) *special form*

違いは次の 2 つの点である。

1. 局所変数の参照は，letrec の構文内（初期値の部分を含めて）すべて有効である。

2. すべての初期値の eval において，変数の参照，代入があってはならない。

第 2 の制限のため，初期値に与えるのは通常 lambda 式である。lambda 式の eval では関数クロージャを作るだけなので， 2 が自動的に満たされる。第 1 の条件により，この lambda 式には，letrec の任意の変数を書くことができる。これは，複数の lambda 式が相互に呼び出す再帰を定義するのに便利な機能となる。

```
(macro-expand-rec '(do ((n 0 (+ n 1))(s 0 (+ s n))) ((= n 10) s) ) ())
```

とすると

```
(letrec ((G238
           (lambda (n s)
                (if (= n 10) (begin s)
                   (begin (G238 (+ n 1) (+ s n)))
                )
           )
         )
        )
  (G238 0 0) ;;; body
)
```

のように返ってくる。（見やすいように改行してある。）do がマクロで，実際には letrec を使って実行されているわけである。G238 は gensym により生成されたシンボルである。これに (評価された）lambda 式がバインドされて本体の

(G238 0 0)

により，実行されている。n<10 のときは，lambda 式の中で G238 が再帰呼び出しされてループができているのがわかる。0 から 9 までの和が計算される。

Chapter 2

TKを使う

2.1 TKとは

TK は tool kit の略であり，window,button などのＧＵＩ部品を集めたものである。これにスクリプト言語 Tcl を組み合わせて良く使われているのが **Tcl/Tk** である。ss はこの Tk を内部に組み込み，Tcl からの呼び出し形式とほぼ同じ形で schme 言語からＴｋを呼べるようになっている。このため，多くの Tcl/Tk で書かれたプログラムを移植しやすくなっている。いろいろな処理には scheme 言語が使えるのでＴｃｌより柔軟なプログラミングができる。なお Tk は Perl,Ruby,Python などでも利用できる。

2.2 ハローワールド

最初に次のプログラムを見よう。

```
(toplevel '.hello)
(wm 'geometry .hello "200x100+100+50")
```

1 行目を実行すると，hello という名前の window が現れる。２行目をやるとその大きさと位置が変わる。きわめて簡単に window が作れる。Tk ではウインドウや**ウイジェット**（ボタンなどの部品のこと widget）の名前は

.hello.bt

のようにドット. から始まる階層形であらわされる。これは Unix のディリクトリの

/home/sato

のようなものと思えばよい。/の代わりが. である。. は**ルートウィンドー**とも呼ばれる。２行目の **wm** はウインドーマネージャーの呼び出し関数である。'geometry で.hello の大きさと位置を指定することを示している。データ

”200x100+100+50”

は大きさが 200x100 で位置が x 方向 100, y 方向 50 を表している。ただし，グラフィックの絶対座標は画面の左上隅が (0,0) で，x は右方向 y は下方向になる。単位は**ピクセル**（１画素）である。

２行目で.hello にクォートがついていないのはなぜだろうか。１行目で.hello が作られたとき，自動的に.hello にはＴｋオブジェクトを示すデータがバインドされる。このため.hello を eval した結果は.hello の実体そのものを指すことになるのでクォートは不要になるのである。このウインドーは右上のＸボタンを押せば終了する。もう 1 つの方法は scheme から **destroy** 関数を使って

(destroy .hello)

としてもよい。もし，.hello の下に階層があればそれらもすべて消滅する。

2.3 ラベルとボタン

ウインドーの中に文字列を表示するラベルを置いてみよう。

```
(toplevel '.hello)
(wm 'geometry .hello "200x300+100+50")
(label '.hello.lab1 :text "hello!world" :relief 'solid)
(pack .hello.lab1)
```

.hello の中に作るので名前が 1 段階進んで.hello.lab1 という名前のラベルを作っている。このようなものを.hello の子ウイジェットという。.hello はその親である。

:text のようにコロン:で始るシンボルは特別でキーワードシンボルと呼ばれる。これは即値データで変数としては使えない。そのかわり，例にあるようにクォートは不要である。

(**label** 名前 option ...) *procedure*

label ウイジェットを定義する。:text キーワードはそのラベルに置かれる文字列の指定を示す。

T k の多くの関数は

:key1 value1 :key2 value2 ···

のような引数（option) を許す。キーワードがあるおかけでこれらの指定の順番は変えてもよいし省略されてもよい。:relief はラベルの枠の形の指定のために使われる。どんな指定が可能かは Tcl/Tk の本をみればわかる。ss では Tk の指定方をそのまま組み込んであるので一定の規則で読み変えればよいのである。

pack は pack geometry マネージャーとよばれ，ウイジェットの配置を行うものである。pack をするとラベルが見えるようになる。ここでは.hello.lab1 を配置する。前の label 文で.hello.lab1 にはその実体がバインドされるのでクォートが不要になっている。次にボタンを表示しよう。上につづけて

(**button** '.hello.bt1 :text "start")

(pack .hello.bt1)

とすると start という表示のボタンが現れる。マウスで左クリックしてみる。何もおこらないが押せることはわかるはずである。

(destroy .hello.bt1)

でボタンを消して，今度は

(button '.hello.bt-exit :text "exit" :command "exit")

(pack .hello.bt-exit)

とする。exit ボタンを押すとすべて消えてしまう。**:command** キーワードは，ボタンが押されたとき何をするかを指定する。

"exit"

はT k コマンドの exit を実行することを意味するのである。では，scheme のある関数を実行するにはどうするか。ss を起動して

```
(toplevel '.hello)
(wm 'geometry .hello "200x300+100+50")
(label '.hello.lab1 :text "hello!world" :relief 'solid)
(pack .hello.lab1)
(button '.hello.bt1 :text "start" :command '(change))
(pack .hello.bt1)
(define (change)
   (.hello.lab1 'configure :text "change-ok")
 )
```

として，start ボタンを押す。hello!world のラベルの文字列が

change-ok

と変わる。今度は start ボタンを押すと ss の関数 (change) が実行されるのである。さて，関数 change の中を見よう。

(.hello.lab1 'configure :text "change-ok")

.hello.lab1 が先頭にある。scheme の規則により，.hello.lab1 は関数である。つまり，label コマンドによってラベルが定義され，その操作関数がシンボル.hello.lab1 にバインドされたのである。configure は widget の属性（表示文字，色，枠，など）を変更することを意味する。つまり

(<widget 名 > 操作コマンド option ...)

のような形で widget 操作ができる。操作コマンドはキーワードでは無いのでクォートが必要になる。

よびだされる schme 関数には制限はない。ss の関数を自由に使ってよいのである。多くの場合なんらかの計算をしたあと，widget や window の状態を変更することになる。

練習

(1) :bg カラーは background color の属性である。button を押す度にその色が red,blue,green と巡回的に変化するようにしなさい。

(2) 1 から 3 の数字を書いたボタンと 1 個のラベルを作り、ボタンを押すたびにそれまでの合計がラベルに表示されるようにしなさい。

2.4 いくつかの関数とマクロ

push マクロ，pop マクロは 1 章で説明した。**tk-name** 関数は

(**tk-name** s1 s2 ...) *procedure*

これは引数に与えられたすべてのシンボル，数，文字，文字列を連結した名前のシンボルを返す。

(tk-name ".hello.bt" 3)

.hello.bt3

のようになる。関数 **tk-string** も同様で，これは連結した文字列を返す。これらは、TCL/TK において widget の名前を作ったりするのに特に便利なので tk-のついた名前になっているが、普通のプログラミングに用いても良い。また、 1 章で解説した funcall は、widget を引数とする関数を書きたいときや、その名前を使って呼び出したいときに便利でこれもよく使う。これらは、すべて sslib の中で定義されていて ss の起動時に自動的に読み込まれる。

(**winfo** word win) *procedure*

これは win で指定された widget の word に関する情報を返す。

指定できる word は、いろいろとあるが

(winfo 'exists ".hello.bt")

のように exists を指定すれば、.hello.bt が存在すれば#t を返す。width,height,parent などが使える。

練習

(1) 関数 (greet str) を呼ぶと、1 個の window(名前は .greet) を作りその中に与えられた文字列 str を text とする button を作るようにしなさい。

(2) 前の (1) を修正して、.greet がすでに存在するときは、.greet1 の形の名前（それも既にあれば、さらに...) で存在しないものを見つけて作るようにしなさい。

2.5 複数のボタン

3 個のボタンを押して，どれが押されたかを表示してみよう。

```
(toplevel '.hello)
(wm 'geometry .hello "200x300+100+50")
(label '.hello.lab1 :text "hello!world" :relief 'solid :width 20)
(pack .hello.lab1)
(button '.hello.bt1 :text "button 1" :command '(change 1))
(pack .hello.bt1)
(button '.hello.bt2 :text "button 2" :command '(change 2))
(pack .hello.bt2)
(button '.hello.bt3 :text "button 3" :command '(change 3))
(pack .hello.bt3)

 (define (change n)
   (.hello.lab1 'configure :text (tk-string "button is " n))
 )
```

もっとたくさんのボタンだと, ループを使うのがよい。

```
(define (start)
   (toplevel '.hello)
 ;  (wm 'geometry .hello "200x600+100+50")
   (label '.hello.lab1 :text "hello!world" :relief 'solid :width 10)
   (pack .hello.lab1)
   (do ((n 1 (+ n 1)) (name 0))
       ((> n 12) )
       (set! name (tk-name ".hello.bt" n))
       (button name :text (tk-string "BT " n)
             :command  `(change ,n))
       (pack name )
   )
 )
(define (change n)
   (.hello.lab1 'configure :text (tk-string "button is " n))
 )

  (start)
```

12 個のボタンが表示されたはずである。ここでバッククォートが使われているが

(list 'change n)

としても同じである。良くわからない人はこれでもよいが，バッククォートの方がスマートでしょう。表示部分を関数 start にして，最後に start を呼んでいることにも注意されたい。この方が，画面再表示が簡単になり，なにかと都合がよい。

練習

1 から 3 のボタンとラベルを表示して、例えばボタンを 1,2,3 と押すとラベルの表示が 1,12,123 と変化していくようにしなさい。

2.6 pack と frame

前の例ではボタンがたて 1 列に表示されるだけで面白くない。pack にはいろいろな機能があり，配置を工夫できる。前のプログラムで，

(pack name)

を

(pack name :side 'left)

に変えてみる。今度は横 1 列になる。:side は pack の方向を決めるキーワードで left なら左から，top なら上から順につめる。初期値は top なわけである。長方形にしたかったらどうするか？これには **frame** という widget を使うのが簡単である。frame は目に見えない枠でこの中にボタンを pack して，さらに frame を pack する。4x5 の長方形を作ってみよう。4 個の frame に 5 個ずつ入れて各 frame は top から pack してみる。

```
(define (start)
   (toplevel '.hello)
   (label '.hello.lab1 :text "hello!world" :relief 'solid :width 20)
   (pack .hello.lab1)
   ;;; frame
   (do ((n 1 (+ n 1))(num 1) (fname 0))
       ((> n 4) )
        (set! fname (tk-name ".hello.fr" n))
        (frame fname )
        (do ((n 1 (+ n 1)) (name 0))
            ((> n 5) )
             (set! name (tk-name fname #\. "bt" num))
             (button name :text (tk-string "button " num) :width 12
                :command  `(change ,num))
             (pack name :side 'left)
             (set! num (+ num 1))
        )
        (pack fname)
    )
 )
(define (change n)
   (.hello.lab1 'configure :text (tk-string "button is " n))
 )

  (start)
```

ここで

(**frame** 名前 option ...) *procedure*

は，もちろん frame を作る T k コマンドである。各ボタンが frame の子供として定義されその中に左詰めで pack されている。frame を top から pack して完成である。ボタンの大きさが不揃いにならないように:width で幅を指定している。

より複雑な配置をするときは，frame の中にさらに frame を配置するなどして対応できる。

配置マネージヤには，pack の他に grid,place がある。T k の本を参考にしてもらいたい。以上を理解した人は，どう書換えたらよいかわかるはずである。T k の解説書では，たとえば

button .bt -text ”hello”

pack .bt -side left

と書いてある。TCL/TK の option は -名前の形だが，これは ss では:キーワードに変更する。その値が名前（シンボル）のときは，クォートをつける。指示コマンド (シンボルである) もクォートする。widget を最初に定義するときはクォートする。以上である。

練習

0 から 9 のボタンと+、＝のボタンと表示用のラベルを用意して足し算電卓を作れ。

2.7 update

簡単なゲームを作ってみよう。

```
;;;  slot game
;;;

  (define (start)
    (toplevel '.slot)
    (wm 'title .slot "SLOT GAME")
    (button '.slot.b1 :text "0" :width 4 :border 5)
    (button '.slot.b2 :text "0" :width 4 :border 5)
    (button '.slot.b3 :text "0" :width 4 :border 5)

    (button '.slot.start  :text "START" :command '(slot))
    (button '.slot.stop :text "STOP" :command '(end))
    (button '.slot.exit :text "End" :command "exit")
    (pack .slot.b1 .slot.b2  .slot.b3
        .slot.start .slot.stop .slot.exit  :side 'left)
  )
;;;;
 (define *owari* #f)

 (define (end)
     (set! *owari* #t)
     (format #t "owari~s~%" *owari*)
 )

 (define (slot)
    (set! *owari* #f)
   (do  ()
      (*owari*)
      (change)
      (after 100 )
      (update)
     ;;(format #t "change ~s" *owari*)
    )
 )

 (define (change)
  (let
```

```
    (( x1 ( random 10))
     ( x2 ( random 10))
     ( x3 ( random 10))
    )
   (.slot.b1 'configure :text (tk-string x1))
   (.slot.b2 'configure :text (tk-string x2))
   (.slot.b3 'configure :text (tk-string x3))
))

(start)
```

start ボタンを押すと，数字が変化し stop ボタンを押すと止まる。関数 slot をよく見てみよう。最初にグローバル変数 *owari* が#f にセットされて do ループに入る。終了条件は *owari* が真になったときである。stop ボタンを押すと (end) が呼ばれ *owari* が真になる。ところで，キーボードやマウスを押すなどなどをすると，**イベント**が発生したという。関数 slot は start ボタンが押されたイベントを処理する関数である。T k ではあるイベントを処理している間に発生した他のイベントは前の処理が終わるまで処理されないことになっている。これだと stop を押しても，slot 関数は終了しない。そこで

(**update**)

の登場である。これは，それまでに発生したイベントを強制的に処理するようにする。こうして (slot) は停止する。また

(**after** 100)

は，100 ミリ秒待つ命令で変化のスピードを押さえるために入っている。

練習

このゲームの表示を 2 段にして上段の 3 個のボタンはやや小さくして直前の結果を表示するように変更せよ。

2.8 画像を使う

文字列だけだとつまらないので，画像を表示してみよう。ss では，.xbm と.gif のタイプの画像が扱える。.xbm（X Bitmap) は Unix で使われる白黒のみの画像である。.gif はカラー画像で，Windows などでも広く使われている。最初に.xbm 画像からやろう。.xbm 画像は Linux の画像処理プログラム (xv,xpaint, など）で作れるのであらかじめ用意しておく。また，Windows 用のプログラムでも作れるものがあるのでそれでも良い。ラベルやボタン関連 widget は bitmap 属性を持っているので，:text と同様に

(button '.hello.bt1 **:bitmap** '@hello.xbm :border 5)

のようにすればよい。アットマークの後に画像ファイル名を書けばよい。クォートを忘れずに。これを使って最初の hello world を実行してみてほしい。データの無い人は

(button '.hello.bt1 :bitmap 'question :border 5)

を試してみよう。アットマークのつかない画像は T k の組み込み画像である。

gif を使うには，**image create** を使う必要がある。最初に

(image 'create 'photo 'maru :file "maru.gif")

のようにする。photo までは決まり文句であり，次の maru はこの image に対する名前である。シンボル maru には，画像の処理関数がバインドされる。そこで

(button '.hello.bt2 :image maru :border 5)

(pack .hello.bt2)

で表示できる。次に

(maru 'blank)

とすると，この画像は透明になる。画像を他のものに取りかえるには，たとえばすでに batu が image create されているとして

(.hello.bt2 'configure :image batu)

と configure を使えばよい。

練習

前セクションの SLOT GAME の表示を画像にせよ。

2.9 ラジオボタン

ラジオボタンは，いくつかの選択肢から 1 つだけ選ばせるための widget である。似たものに**チェックボタン**（複数選択可) がある。

```
(define *n* 0)
(define (radio)
   (toplevel '.sample)
   (wm 'geometry .sample  "300x100+80+350")
   (radio-init)
)
(define (radio-init)
  (let ( (n-list '(5 7 9 13 19) )
                (n-list-label '("5" "7" "9" "13" "19")))
       (frame '.sample.radio)
    (do ((i 0(+ 1 i))(name 0))
        ((null? n-list))
        (set! name (tk-name ".sample.radio" #\.  i))
        (radiobutton name  :variable '*n*
               :value (pop n-list )
            :text (pop n-list-label ) )
        (pack name  :in .sample.radio  :side 'left)
    )
      (button '.sample.ok :text "OK" :command  '(make-ban))
      (button '.sample.exit :text "exit" :command  '(owari))
      (pack .sample.radio .sample.ok .sample.exit)

   (toplevel '.ban)
   (image 'create 'photo 'shiro :file "shiro.gif")
   (image 'create 'photo 'kuro :file "kuro.gif")

))

(define (make-ban)
    (destroy .ban)
    (toplevel  '.ban)
  (do ((i *n*  (- i 1)) (bt 0))
      ((= i 0))
      (set! bt (tk-name ".ban.bt" i))
```

```
      (button bt :image shiro :command  `(change ,i))
      (pack bt :in .ban  :side 'right)
   )
)

(define (change i)
   (let ((image 0) (bt 0))
     (set! bt (tk-name ".ban.bt" i))
     (set! image (funcall bt 'cget :image))
     ;;;;         (set! color (bt 'cget :image)) はエラー
     (format #t "~s ~s ~%" bt image)
     (if (equal? image "shiro")
            (funcall bt 'configure :image kuro)
            (funcall bt 'configure :image shiro)
     )
 ))
(define (owari)
   (destroy .ban)
   (destroy .sample)
   (bye)
 )

  (radio)
```

radiobutton 関数の中の

:variable '*n*

:value (pop n-list)

に注目しよう。ボタンのどれが選択されたかを伝えるためのグローバル変数を *n* にしている。押されたときに，*n* に入る値が:value で指定されている。OK ボタンを押すと，*n* の値に応じて大きさの違う盤が現れる。同じグローバル変数を共有する，radiobutton が 1 組になる。

ここでは，toplevel ウィンドウを 2 つ作っている。radio ボタンを表示してＯＫと exit ボタンを表示する.sample が全体のメインで，ＯＫを押すたびに盤は変更されている。

関数 change では，ボタンに現在表示されている画像が何かを知るために

(widget 'cget :image)

が使用されている。**cget** は widget の持っている属性を取り出すための指示コマンドで，ここでは:image 属性をとりだしている。これにより，画像が shiro か kuro かを見て逆転するようになっている。このように widget の情報を widget 自身から取り出すことができる。

この種の情報は，scheme 側で管理してもよい。たとえば適当な配列に現在の状態を保持するようにして，ボタンが押されたらそれを見て状態を変更する。ゲームなどのプログラミングでは，盤の状態からいろいろなことを計算，判定しなければならないので scheme 側で情報管理するほうが普通である。 1 つの例として見てもらいたい。

もう 1 点注意する。pack の中で

:in

という option がある。これは，pack するときどこの中にするかを指示するためにある。ラジオボタンはフレーム.sample.radio の中に pack されている。ただし，これは本当は必要ない。上のプログラムではラジオボタンは.sample.radio.0 のような形で.sample.radio の子として定義されているからである。親子関係のとおりに pack する場合は，この option は必要な

い。frame の中に frame を置くなどして，完全な名前が面倒なときや，後から frame を追加してその中に配置したいときなどに使用すればよい。

練習

checkbutton について調べて、食事のメニュー選択と合計表示ができるプログラムを書け。

2.10 エントリーとバインド

```
;;;;   load  file GUI
(toplevel '.load)
(wm 'geometry .load  "200x100+100+50")
(label  '.load.lab  :text "Input filename")
(entry '.load.file )
(pack .load.lab .load.file )
(bind .load.file "<Return>" (lambda ()(load-file)))
(label '.load.mesg :text "message")
(pack .load.mesg)

(define (load-file)
   (let ((file  (.load.file 'get)) (mesg ""))
     (if (file-exists? file)
          (begin
             (load file)
             (set! mesg "Load OK!")
          )
         (set! mesg "Not exist")
     )
     (.load.mesg 'configure :text mesg)
))
```

このプログラムは，load を実行するためのＧＵＩである。実行すると入力らん (entry widget) が現れるので

hello.ss

などと入力して，return キーを押すと load できる。間違うと，Not exist と表示される。**entry** は，このように 1 行だけのテキスト入力を扱う widget である。

入力の終了には別のボタン（Ｏｋのような）を用意してそれを押すようにしてもよい。ここでは，return キーを押すことで発生するキーボードイベントを利用する。Ｔｋではイベントが発生したときそれを処理する関数をそのイベントに**バインド**することによってＧＵＩが動く。ボタンのときマウスの左クリックに対するバインドは:command オプションで指定した。いろいろなイベントを処理するためのバインドを定義できる汎用関数が **bind** である。

(**bind** Tk-obj イベント名 処理関数実体) *procedure*

が一般形である。Tk-obj は window または widget またはクラス名である。処理関数実体は:command のときと違って，単なる呼び出し形ではなく必ず **lambda 式**（つまり関数そのもの）でなければならない。

(lambda()(load-file))

を評価すると 0 個の引き数を持つ関数そのものになることを思いだしてほしい。かわりに

load-file

と書いてもよい。load-file を eval すると load-file の実体になるからである。しかし

'(load-file)

と書くと間違いである。この式の評価は単なるＳ式で関数実体ではない。とにかく lambda 式を使うと覚えておこう。イベント名はいろいろある。

"<Return>"

は明らかだろう。キーボード関連のイベント名としては

<KeyPress>,<Key-a>,<Key-b>,<Escape>,<BackSpace>,

<Up>,<Down>,<Left>,<Right>,

<Control-Key-c>,<Shift-Key-c>

などで，これらの名前は Unix の場合キーシム (keysym) と呼ばれている。キーシムを表示するＧＵＩを書いてみよう。

```
;;;  keysym
;;;
(toplevel '.key)
(wm 'geometry .key "600x100+80+80")
(wm 'title  .key  "KeySym")
(label '.key.la :text "Push any key")
(pack '.key.la)

(label '.key.w :text
    "Window-pass x-position y-position Keysym char-display" )
(label '.key.result  :width 70)
(pack .key.w .key.result)

(bind  .key "<KeyPress>"
          (lambda(|W| x y |K| |A| )
            (.key.result 'configure :text
             (format #f "~S  x=~S  y=~S keysym=~S  display=~S "
                      |W| x y |K| |A|  ))))
```

ここで，<KeyPress> の処理関数が５個の名前の引数を用いて書かれている。|W| はＴｋの内部変数 W で，これはイベントの発生した window のパス名が入る。たて棒は，このシンボルが大文字のためである。普通の scheme では小文字が標準でキーボードからの入力は文字列などを除いて必ず小文字に変換される。これを避けるために，大文字を含むシンボル名は縦棒で囲むことになっている。ss では、変換が行われないので縦棒は不要だが間違いではないので付けてある。同様に |K|, |A| も大文字のシンボルでそれぞれキーシム名と画面表示文字が入る。x,y はイベントの発生した相対座標（そのウインドウの左上隅が原点）である。これらの名前はＴｋで決まっているので，他の情報が必要ないなら上のバインド文を

```
(bind .key  "<KeyPress>"
          (lambda( |K| )
            (.key.result 'configure :text
             (format #f " keysym  ~S"
                       |K|  ))))
```

のように必要なパラメータのみで書いてよい。

このように、bind では仮引数の名前そのものが情報として必要になっている。これが、bind 文で必ず lambda 式そのものを書かなければならない理由である。

マウス関連のイベント名は

<Button-1>,<Double-1>

などである。<Button-1> は左クリック，<Double-1> は左ダブルクリックである。

(**after** n [コマンド]) *procedure*

は n ミリ秒後にコマンドを実行することを予約する。そしてコントロールはすぐにもどる。次に

(**tkwait** 'variable 変数名) *procedure*

は大域変数の値が変化するまで待つ関数である。

tkwait は他に

(tkwait 'visibility window) ;;; window が可視状態になるのを待つ。

(tkwait 'window window) ;;;window が削除されるのを待つ

の形で使える。

練習

(1) カーソルキーで画像が移動するプログラムを書け。

(2) マウスを左クリックしたとき、その位置を表示して、そこに Click というラベルを表示しなさい。このために place という geometry manager を調べなさい。

2.11 リストボックス

リストボックスは編集はできないが，スクロール可能な複数の行を表示する。リストボックスの使い方とスクロールバーの使い方をみよう。以下の **get-file-name** は、ss2lib にあるので、ss が起動すればすぐ使うことができる。

```
;;; name  get-file-name.ss
;;;
;;; usage:  (get-file-name)
;;;         (get-file-name "*.sgf")
;;;         (get-file-name "*.ss" "*.scm" "*.lsp")
;;;  if no-file then #f
;;;  if no selection or cancel,   ""  is returned
;;;      2002.5.20
;;;
(define *file-kakutei* "")

(define (get-file-name . ext)
  (let ((n-files 0) (dir-list ()) )
    (toplevel '.dir)
     (set!  *file-kakutei* "")
    (if (null? ext)
         (set! ext '("*") ) )

    (wm  'title .dir "File Select window")
    (wm 'geometry .dir "300x300+80+80")
     (listbox  '.dir.lb
        :yscrollcommand
                  (lambda l (apply  .dir.yscroll 'set l))
```

```
       :width 25 :height 8)
     (scrollbar  '.dir.yscroll  :orient 'vertical
                 :command
                    (lambda l (apply  .dir.lb  'yview l))
      )
     (grid  .dir.lb   .dir.yscroll  :sticky 'nws)
     (label '.dir.name :text "file name" :bg 'snow :width 25  :height 2)
     (grid .dir.name)
     (button '.dir.ok :text "OK" :command '(destroy .dir))
     (button '.dir.cancel :text "CANCEL" :command
                       '(begin (set! *file-kakutei* "")(destroy .dir))  )
     (grid .dir.ok .dir.cancel :sticky 'ew )
     (set!  dir-list (apply glob ext))

     (do ((x ()) (i 0) (dirs dir-list))
          ((null? dirs) (set! n-files i))
         (set! x (pop dirs))
          (.dir.lb 'insert 'end  x)
          ; (format #t "~a~%" x)
          (set! i (+ i 1))
       )
     (if (= n-files 0)
       (begin
         (.dir.name 'configure :text "No files")
             (after 2000)
         (destroy .dir)
         #f
        )
       (begin
          (bind '.dir.lb   "<Button-1>" (lambda (y)
                          (get-file-ypos  y dir-list)
                          )
           )
         (bind  '.dir "<Return>" (lambda () (destroy .dir)))
         (tkwait 'window .dir )
         *file-kakutei*
       )
    )
))
(define (get-file-ypos  y dir)
   (let ((i  (.dir.lb  'nearest y  ))
           (file "")
            )
     ; (format #t "~s ~s ~%" i y)
     (set! file  (list-ref dir i))
     (.dir.name 'configure :text file)
     (set! *file-kakutei* file)
))
```

.dir.lb というリストボックスを作り，y 方向のスクロールバーを定義してます。配置マ

ネージャーに pack ではなく **grid** を使ってます。grid は pack のように frame を使わなくても，長方形のような配置ができます。frame を作ってそれを，grid することもできるので，pack よりむしろ簡単（高機能）です。:sticky は，widget の大きさを引き伸ばす方向の指定で，'news（北東西南）と指定すると全方向に可能なだけ引き伸ばします。また:row 行，:column 列で配置位置を指定できます。詳しくはＴｋの参考書を見てください。

(**glob** pat1 pat2 ...) *procedure*

は，Tk の関数で pat1,pat2 ... のどれかにマッチするファイル名のリストを返します。パターンにはワイルドカード (∗,?) などが使えます。

(listbox-widget 'insert index 文字列)

は listbox の index の位置に 1 行挿入します。index は 0 から始る整数や'end で'end の場合は最後に追加されます。

(listbox-widget 'nearest ypos)

は listbox の y 座標に最も近い index を返します。このプログラムでは，マウスをクリックした位置から指定のファイルを見つけるのに使っています。マウスの bind の所で

(lambda (y) (get-file-ypos y dir-list))

の指定があります。ここで dir-list は get-file-name の局所変数で，この lambda 式から見ると上位にある変数です。このような場合 lambda 式は変数評価の環境（今の場合 dir-list に対するバインド）をその中に保持した関数実体を構成します。これを静的クロージャと呼びます。このため，この関数呼び出しはうまくいきます。わかりにくければ，dir-list を ∗dir-list∗ にしてグローバル変数としたほうが良いかもしれません。

練習

リストボックスを左右 2 つ作り、左で選択したものを右へ移し、逆に右からは左に移動できるようにしなさい。

2.12 テキストウィジェットと簡易エディタ

テキストウィジェットは複数行の編集が可能な高機能の widget です。簡単なエディタを作ってみよう。

```
(load "get-file-name")    ;;;; 前節を見よ。

(define (myed)
    (toplevel '.text)
    (wm 'geometry .text "900x650+100+100")
    (wm 'title .text "My editor")
    (label '.text.lb :text "File Name" :bg 'snow)
    (entry '.text.file  :bg 'snow :width 20)
    (button '.text.open :text "Open" :command '(my-open))
    (button '.text.save :text "Save" :command '(my-save))
    (button '.text.exec :text "Exec(ss)" :command '(my-load))
    (button '.text.clear :text "Clear" :command '(my-clear))

    (grid  .text.lb .text.file :sticky 'n)
    (grid  .text.open .text.save .text.exec .text.clear)
    (text '.text.tx :relief 'solid :font '(courier 12)
           :yscrollcommand (lambda l (apply .text.yscroll 'set l))  )
```

```
    (scrollbar '.text.yscroll :orient 'vertical
          :command (lambda l (apply .text.tx 'yview l))  )
    (grid  .text.tx :sticky 'news :columnspan 10 )
    (grid .text.yscroll :sticky 'ns
          :column 10 :row (list-ref (grid 'info .text.tx ) 5))
)
(define (my-open)
  (let ((s 0) (fname 0))
    (set! s (get-file-name "*"))
    (if (or (equal? s "") (eq? s #f))
        ()
      (begin
         (set! fname (.text.file 'get))
         (if (equal? fname "")
               (.text.file 'insert 'end s) )
          ; (.text.file 'configure :text s)
         (set! s (open-input-file s))
         (do ((x (read-line s)(read-line s)))
           ((eof-object? x)(close-input-port  s))
           (.text.tx 'insert 'end x)
           (.text.tx 'insert 'end "\n")
         )
       )
    )
 ))

(define (my-save)
  (let ((s 0) (text ""))
     (set! s (.text.file 'get ))
     (if (equal? s "")
        (error "file name?")
        (begin
            (set! s (open-output-file s))
            (set! text (.text.tx 'get "1.0" 'end))
            (display text s)
            (close-output-port  s)
            (.text.lb 'configure :text "Saved!" :bg 'red)
            (after 3000
               '(and (winfo 'exists '.text.lb)
                     (.text.lb 'configure :text "File Name" :bg 'snow))
            )
        )
     )

 ))
(define (my-load)
  (let ((s 0) (text "") (port 0))
            (set! text (.text.tx 'get "1.0" 'end))
            (set! port (open-input-string  text ))
              (do ((x (read port)(read port)))
```

```
                ((eof-object? x)(close-input-port  port))
                (write (eval x))
                (newline)
            )
          (close-port  port)

))
(define (my-clear)
  (.text.tx 'delete "1.0" 'end)   ;;; 1.0 means  line1, column 0
  (.text.file 'delete 0 'end)
)
```

テキスト widget では，あらかじめたくさんの bind が設定されています。普通の編集機能が使えるのはすぐわかります。さらに便利な機能として

マウスのドラッグで選択範囲を指定。

CTRL-x カット

CRTL-c コピー

CTRL-v 貼り付け

があるので，このままでもかなりのことができます。

関数 text で，

:font '(courier 12)

とあるのはフォントを courier 12 ポイントと指定しています。スクロールバーは前と同様。grid のところで

(grid 'info .text.tx)

とあるのは，.text.tx の grid 属性をリストにして返すコマンドです。(myed) を起動した状態で，ss のプロンプトから直接これをやってみてください。属性リストの 5 番目が.text.tx の置かれている grid 行を示すとわかります。これを使って，スクロールバーを正しい位置に置いています。

:columnspan 10

の指定は，.text.tx が 1 0 列を占めることを表します。my-save 関数の中の

(.text.tx 'get "1.0" 'end)

は，.text.tx の内容を文字列として得るコマンドです。範囲を index で指定しますが index は行（1 から始る）と行の先頭から何文字目（0 から始る）を組み合わせて指定するので"1.0" は先頭を表します。'end で文書の最後までとなるわけです。

(**winfo** 'exists '.text.lb)

は.tex.lb が存在するとき真を返す。3 秒後に.text.lb を元に戻すように after で指定しているのですが，このときエディタがすでに終了しているとエラーが発生するので，先に存在を確かめてから変更しています。

これに，検索や置換の機能を追加すればさらに実用的になる。ssed はその 1 つである。

2.13 canvas

canvas はいりろな図形を描いたり，画像を配置したりできる widget である。関数

$$y = \sin(\frac{1}{x})$$

を描くプログラムを示す。

```
(define *xmin* -1)
(define *xmax* 1)
(define *ymin* -1)
 (define *ymax* 1)
(define (start)
  (toplevel '.can)
  (canvas '.can.c0 :width 800 :height 800)
  (pack '.can.c0)
  (button '.can.print :text "Print"
     :borderWidth  4 :activeBackground 'blue  :command '(ps-print))
  (.can.c0 'create 'window 750 10 :window .can.print)
  (view -0.3  0.3  -0.3 0.3 )
  (fview (lambda (x) (* x (sin (/ x)))))
 )

(define (view xmin xmax ymin ymax)
   (set! *xmin* xmin)
   (set! *xmax* xmax)
   (set! *ymin* ymin)
   (set! *ymax* ymax)
   (line-x-y)
 )

(define (pos x y)
  (list
     (floor (* 800  (/ (-  x  *xmin*) (- *xmax* *xmin*))))
     (floor (* 800  (/ (-  *ymax*  y) (- *ymax* *ymin*))))
  )
)
(define (line pos1 pos2 color)
  (.can.c0 'create 'line (car pos1) (cadr pos1)
        (car pos2) (cadr pos2) :fill color)
)
(define (line-x-y)
   (line (pos *xmin* 0) (pos *xmax* 0) 'blue)
   (line (pos 0 *ymin*) (pos 0 *ymax* ) 'blue)
)

(define (fview fun)
  (let ( (x *xmin*) (h (-  *xmax* *xmin*)) (pos1 ()) (pos2 ()) )
    (set! h (* h 0.0025))
    (set! pos1 (pos x (funcall fun x)))
```

```
    (do ((i 1 (+ i 1)))
        ((= i 400))
        (set! x (+ x  h))
        (set! pos2 (pos x (funcall fun x)))
        (line pos1 pos2 'black)
        (set! pos1 pos2)
    )
))
(define (ps-print)
  (.can.c0  'postscript :file "ps-temp-999.ps")
 )

 (start)
```

canvas には，表示した絵をポストスクリプトで出力する機能がある。上の関数 ps-print を見てもらいたい。Print ボタンを押せば，”ps-temp-999.ps”ができるのでそれを ps2pdf のようなコマンドで pdf に変換できる。

練習

canvas に 1 個の小さい画像を置き、それを canvas 内でドラッグできるようにしなさい。

2.14 参考書など

Tcl/Tk については

Tcl/Tk 入門第 2 版 ウェルチ著 プレンティスホール出版

をみれば，ほとんどすべての解説がある。

Scheme の仕様は

R5RS (http://www.scheme.org/から DL 可能)

が標準である。ss の付属文書 (ソースの展開の中にある）

Doc/∗

は，ss で利用できるほとんどの関数や機能が解説されている。

user >(help)

で同じものが見れる。ssdemo/∗, ogldemo/∗ や sslib そのものも参考になるだろう。

Index