SS 入門 (上級編)

佐藤　定夫[1]

2006 年 7 月

[1]東京電機大学　自然科学系列, 〒 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。

上級編

ネットワーク、構造体、などの上級機能について解説する。これらの多くは、ss 独自の拡張機能であり、標準の R5RS では定義されていない。楽しんでください。

2006 年 7 月,　佐藤　定夫

目 次

第 1 章　基本的データ、関数の補足

1.1　hash 表

hash table は entry (key,value の組) からなるデータベースである。連想リストと異なり、各 entry は key の hash 値をもとにして 1 種のベクターに格納されていく。このため、entry に対するアクセスはほぼ均等で高速になる。hash 値が衝突した場合、通常はその entry はリストとして格納される。衝突が多くなると、hash table を見てもすぐに目的の entry に到達できずに効率が落ちることになる。このため、ss-system は entry の個数が多くなると自動的に表を拡大する。

hash table は

#H(hash . (0 2 0 . #(() () () () () () () () ())))

のように印刷される。これは read 関数で読むことができる。

(**make-hash-table** func size) *procedure*

func は eq?,eqv?,equal?あるいは引数 2 個の関数である。これは、key の比較に用いられる。size は最小サイズで、table サイズは必要ならシステムが自動的に拡大する。

(**get-hash** table key [error-value]) *procedure*

key に対する値を返す。見つからないときは#f または error-value が与えられていればそれを返す。

(**set-hash** table key value) *procedure*

key と value を hash table に登録する。

(**hash-table?** x) *procedure*

(**rem-hash** table key) *procedure*

key の entry を削除する。

(**map-hash** function table) *procedure*

関数呼び出し (function key value) がすべての entry に対して実行される。返す値は table

(**clear-hash** table) *procedure*

すべての entry を消去する。返す値は table

(**hash-table-count** table) *procedure*

登録されている entry の個数を返す。

(**sxhash** obj) *procedure*

これは obj に対する hash-値 (整数) を返す。もし (equal? x y) が真であるなら、この関数は x,y に対し同じ値を返す。

例 1.1. *user* >(define hx (make-hash-table eq? 5))

hx

user >hx

#H(hash . (0 2 0 . #(() () () () () () () () ())))

user >(set-hash hx 'foo 123)

123

user >(set-hash hx 'woo 456)

456

user >(get-hash hx 'foo)

123

user >(get-hash hx 'piyo ())

()

user >(map-hash (lambda(x y) (format #t "~s ~s~%" x y)) hx)

woo 456

foo 123

#H(hash . (0 2 2 . #(() () () ((woo . 456) (foo . 123)) () () () () ())))

user >(sxhash 'woo)

1596

user >(sxhash 'foo)

1443

1.2 vector

(**vector?** obj) *procedure*

(**vector** obj ...) *procedure*

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

(**vector-size** vector) *procedure*

same as vector-length

(**make-vector** size [fill]) *procedure*

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

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

(**vector-fill!** vector fill) *procedure*

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

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

[SS] vector
(**vector-copy** vector [size] [fill]) *procedure*

vector を size の vector に copy する。size がもとより大きいとき fill が用いられる。この copy は、もとと同じ要素を新しい vector に置くだけであり、各要素が新しく copy されるわけではないことに注意せよ。

(**vector-rotate** vector shift) *procedure*

shift が正ならば vector の要素は右へ移動する。

1.3 [SS] matrix

2 次元配列について解説する。
vector は

#(1 2 3 4)

のようであったのに対し、配列は

#a((1 2)(3 4))

のように書く。 index は 0 から始める。
(**matrix?** obj) *procedure*

(**make-matrix** gyo retu [value]) *procedure*

user >(make-matrix 2 3 0)
#a((0 0 0)(0 0 0))

(**matrix** list-1 list-2 ....) *procedure*

user >(matrix '(1 2)'(2 4))
#a((1 2)(2 4))

(**matrix-size** matrix) *procedure*

matrix の size をドットペアで返す。*user* >(define y (matrix '(1 2)'(3 4)))

user >y
#a((1 2)(3 4))
user >(matrix-size y)
(2 . 2)

(**matrix-ref** matrix gyo retu) *procedure*

user >(matrix-ref y 0 1)
2

(**matrix-set!** matrix gyo retu value) *procedure*

user >(matrix-set! y 0 1 6)

user >y
#a((1 6)(3 4))

(**matrix->list** matrix) *procedure*

user >(set! y (matrix '(1 2)'(3 4)))

user >(matrix->list y)
((1 2) (3 4))

(**list->matrix** list) *procedure*

user >(list->matrix '((1 2) (3 4)))
#a((1 2)(3 4))

(**vector->matrix** vector) *procedure*

vector を 1 列の配列に変換する。

user >(vector->matrix #(1 2 3))
#a((1)(2)(3))

(**matrix-fill!** matrix value) *procedure*

user >(matrix-fill! y 3)

user >y
#a((3 3)(3 3))

(**matrix-copy** matrix) *procedure*

もとの matrix と同じ要素を持った, 同じ size の matrix を返す。要素は新しい matrix に単純に置かれるだけなのでその要素に対して破壊的操作をするともとの matrix の要素も変化するので注意。

(**matrix-map-1** matrix func gyo) *procedure*

行 gyo に func を適用した結果をその gyo に代入する。破壊的。

user >(matrix-map-1 y sqrt 0)

#a((1.73205080756888 1.73205080756888)(3 3))

(**matrix-map-1-t** matrix func retu) *procedure*

(**matrix-map-2** matrix func gyo1 gyo2) *procedure*

gyo1 と gyo2 の各要素に対し 2 変数関数 func を適用して、その結果を gyo2 に代入する。破壊的。

user >(set! y (matrix '(1 2)'(3 4)))

user >(matrix-map-2 y (lambda (x y) (+ x y y)) 0 1)

#a((1 2)(7 10))

(**matrix-map-2-t** matrix func retu1 retu2) *procedure*

(**matrix-swap** matrix i j) *procedure*

行の交換

(**matrix-swap-t** matrix i j) *procedure*

列の交換

(**matrix-transpose** matrix) *procedure*

転置行列を返す。

(**matrix-transpose!** matrix) *procedure*

転置行列にする。破壊的

(**matrix-resize** matrix gyo retu fill-value) *procedure*

サイズを変更した新しい matrix を返す。 大きくするときは、fill-value が用いられる。

(**matrix-mul** m1 m2) *procedure*

積 m1m2 を計算する。

(**matrix-mul-t** m1 m2) *procedure*

= (matrix-mul m1 (matrix-transpose m2))

(**matrix-mul-set!** m1 m2 m3) *procedure*

結果を m3 に代入する。m3 は m1.m2 と別の matrix でないといけない。

(**matrix-mul-t-set!** m1 m2 m3) *procedure*

結果を m3 に代入する。

(**matrix-add** m1 m2) *procedure*

和 m1+m2 を計算する。

(**matrix-add-set!** m1 m2 m3) *procedure*

結果を m3 に代入する。

(**matrix-base-1** matrix c gyo) *procedure*

行 gyo を c 倍する。

(**matrix-base-1-t** matrix c retu) *procedure*

(**matrix-base-2** matrix c i j) *procedure*

i 行の c 倍を j 行に加える

(**matrix-base-2-t** matrix c i j) *procedure*

(**matrix-det** matrix) *procedure*

行列式の計算

(**matrix-det!** matrix) *procedure*

破壊的

(**matrix-base** matrix) *procedure*

行基本変形を計算する。

(**matrix-inverse** matrix) *procedure*

逆行列を求める。

(**matrix-append** A B) *procedure*

行列 A,B を横に並べた行列を作る。

(**matrix-append-t** A B) *procedure*

縦に並べる。

(**matrix-divide** matrix n) *procedure*

行列を 2 個に分割した list を返す。(A1 A2) で A1 は、A の 0 行から n-1 行で、A2 は残りである。

(**matrix-divide-t** matrix n) *procedure*

(**matrix-rotate** matrix sh [start][end]) *procedure*

行に関して整数 sh だけ rotate する。sh=1 なら 0 行は 1 行に移動し、.... start,end で rotate の範囲が指定できる。end の省略値は、matrix の行数である。rotate は start 行から end-1 の行までの範囲で行なわれる。

(**matrix->vector-set!** matrix gyo vector) *procedure*

matrix の gyo を vector に書く。vector の size が列数と異なっているときは単に不足分を無視する。

(**matrix<-vector-set!** matrix gyo vector) *procedure*

逆に vector の内容を matrix の gyo に書き込む。

1.4 char

文字 object は 1 つ 1 つの文字を表現する object である。

#\a, #\A, #\\,#\”,#\ あ, #\ 文

は文字 a,A,\,”, あ, 文 を表す。特殊コードは名前で表現される。

#\space, #\newline, #\tab

などであり、名前の完全な表は以下である。

```
{"null",0},{"soh",1}{"stx",2},{"etx",3},{"eot",4},{"enq",5},
{"ack",6},{"bell",7},{"backspace",8},{"ht",9},{"tab",9},{"newline",10},
{"vt",11},{"page",12},{"return",13},{"so",14},{"si",15},
{"dle",16},{"dc1",17},{"dc2",18},{"dc3",19},{"dc4",20},
{"nak",21},{"syn",22},{"etb",23},{"can",24},{"em",25},
{"sub",26},{"escape",27},{"fs",28},{"gs",29},{"rs",30},{"us",31},
{"space", 32},
{"delete",127},
{"kspace", 41377},
```

文字は即値データでありクオートは不要である。文字は内部 register 値として表現されているため、eq?で比較可能 ([SS]) である。

(**char?** obj) *procedure*

(**char=?** char1 char2) *procedure*

(**char<?** char1 cahr2) *procedure*

(**char>?** char1 cahr2) *procedure*

(**char<=?** char1 cahr2) *procedure*

(**char>=?** char1 cahr2) *procedure*

これらは (char->integer char) の値の比較である。

(**char-ci=?** char1 char2) *procedure*

(**char-ci<?** char1 cahr2) *procedure*

(**char-ci>?** char1 cahr2) *procedure*

(**char-ci<=?** char1 cahr2) *procedure*

(**char-ci>=?** char1 cahr2) *procedure*

これらは英大文字と小文字の区別をしない。

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

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

(**char-whitespace?** char) *procedure*

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

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

(**char-alphanumeric?** char) *procedure*

alphabetic or numeric

(**char-word?** char) *procedure*

alphabetic, numeric or _

(**char-kanji?** char) *procedure*

char is EUC-char ?

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

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

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

ASCII コードを返す。日本語文字に対しては、euc コードである。

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

1.5 string

(**string?** obj) *procedure*

(**string** char ...) *procedure*

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

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

漢字モードに関係なく単に byte としての長さを返す。[ss]

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

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

(**substring** str start [inc] end [inc]) *procedure*

部分文字列を取りだす。start から end の手前までである。start,end は index であり 0 以上の整数だが、 :end を用いてもよい。 inc が指定されたときは、inc が加算される。inc は必ず負の整数であり。前には:end がないといけない。

例 1.2. *user* >(substring "test" 1 :end)

"est"

user >(substring "test" 1 :end -1)

"es"

user >(substring "test" :end -2 :end)

"st"

(**make-string** len [char]) *procedure*

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

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

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

漢字モードに関係なく単に 1byte 文字としてのリストを返す。

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

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

(**string-fill!** str char) *procedure*

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

(**symbol->string** symbol) *procedure*

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

(**string-ci=?** str1 str2) *procedure*

大文字、小文字を区別しないで比較する。 ci='case ignore'

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

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

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

(**string-ci>?** str1 str2) *procedure*

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

(**string-ci<=?** str1 str2) *procedure*

(**string>=?** str1 str2) *procedure*

(**string-ci>=?** str1 str2) *procedure*

(**number->string** number [radix]) *procedure*

(**string->number** str [radix]) *procedure*

[SS] string

(**keyword->string** keyword) *procedure*

user >(keyword->string :top)

":top"

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

user >(string->keyword ":top")

:top

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

bag に指定された文字を区切りとして str を分割した string のリストを返す。bag の default は '(#\space #\tab #\newline) である。bag に文字列を指定したときはその string->list が bag となる。

例 1.3. *user* >(string-split "sato foo piyo what")

("sato" "foo" "piyo" "what")

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

("usr" "local" "env")

(**string-index** str1 str2 [:nocase]) *procedure*

str1 を str2 の先頭から探しみつかればその index を返す。ないときは#f が返る。 :nocase が指定されたときは大文字、小文字の区別をしない。

(**string-rindex** str1 str2 [:nocase]) *procedure*

str1 を str2 の末尾から探しみつかればその index を返す。ないときは#f が返る。:nocase が指定されたときは大文字、小文字の区別をしない。これらは、単純なアルゴリズムを使用している。

例 1.4. *user* >(string-index "ss" "associate")

1

(**string-intern?** str) *procedure*

str を印字名とするシンボルがアクセス可能ならそのシンボルを返す。

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

item は文字列、文字、シンボル、数のどれかである。それらを文字列に変換して append した結果を返す。

(**tk-name** item ...) *procedure*

同様に作った文字列を印字名とするシンボルを返す。

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

小文字に変換した新しい文字列を返す。

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

大文字に変換した新しい文字列を返す。

(**string-capitalize** str) *procedure*

単語の先頭を大文字にした新しい文字列を返す。

(**string-search** str1 str2 [start2 end2 :repeat :nocase :last]) *procedure*

Boyer-Moore Algorithm を用いて、str1 を str2 から捜す。start2 と end2 は捜す範囲を制限する。:end または :end 負の数を使うことができる。

:ci は:nocase と同じである。.

:last が指定されたときは、最後に見つかった index が返される。

もし、 str1 が前にこの関数を呼んだときと同じなら, 前の BM-tables が用いられる。

例 1.5. *user* >(string-search "test" "test kTest TEST " :repeat :nocase)
(0 6 11)

(**string-prefix-length** s1 s2 [start1 end1 start2 end2]) *procedure*

これは s1 と s2 の最初の共通部分の長さを返す。start1 end1 start2 end2 は範囲の制限である。:end または :end 負の数を使うことができる。

(**string-prefix-length-ci** s1 s2 [start1 end1 start2 end2]) *procedure*

string-prefix-length に :ci または:nocase を与えるのと同等である。.

(**string-suffix-length** s1 s2 [start1 end1 start2 end2]) *procedure*

(**string-suffix-length-ci** s1 s2 [start1 end1 start2 end2]) *procedure*

string-suffix-length に :ci または:nocase を与えるのと同等である。

(**string-prefix?** s1 s2 [start1 end1 start2 end2]) *procedure*

(**string-prefix-ci?** s1 s2 [start1 end1 start2 end2]) *procedure*

(**string-suffix?** s1 s2 [start1 end1 start2 end2]) *procedure*

(**string-suffix-ci?** s1 s2 [start1 end1 start2 end2]) *procedure*

1.6 list

[R5RS] list

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

x が () なら#t を返す。

(**pair?** x) *procedure*

x が pair なら#t を返す () は pair ではない。。

(**list?**) *procedure*

x が真正リストのとき#t を返す。つまり x は () であるか最後の cdr が () であるリストのとき真を返す。

(**cons** x y) *procedure*

x,y の pair (x . y) を返す。x は car, y は cdr と呼ばれる。

(**car** x) *procedure*

pair の car を返す。() の car を取ることはエラーである。

(**cdr** x) *procedure*

(**length** <list>) *procedure*

list は真正リストでなければならない。

(**list-tail** <list> n) *procedure*

takes cdr n times.

(**drop** <list> n) *procedure*

takes cdr n times.

(**list-ref** <list> n) *procedure*

:= (car (list-tail list n)) list の弟 n 要素を返す。先頭は弟 0 要素である。

(**list** x ...) *procedure*

リスト (x ...) を返す。

(**list*** x1 ... xn <last>) *procedure*

(x1 ... xn . <last>) を返す。

(**append** <list> ...) *procedure*

各要素リストを構成する要素を続けて並べたリストを返す。最後の引数はリストでなくてもよい。その場合結果は非真正リストである。

(**reverse** <list>) *procedure*

(**set-car!** <pair> obj) *procedure*

(**set-cdr!** <pair> obj) *procedure*

(**caar** <pair>) *procedure*

(**cadr** <pair>) *procedure*

...
...
(**cdddar** <pair>) *procedure*

(**cddddr** <pair>) *procedure*

(**assq** x <list>) *procedure*

list の各要素は pair である。各 pair のうちその car が x と eq?であるよ最初の pair が返される。存在しなければ#f が返る

(**assv** x <list>) *procedure*

eqv?

(**assoc** x <list>) *procedure*

equal?

(**memq** x <list>) *procedure*

x と list の要素が eq?で#t となる最初の要素をさがして、そこからの部分リストを返す。存在しないときは#f を返す。

(**memv** x <list>) *procedure*

eqv?

(**member** <list>) *procedure*

equal?

(**map** proc list1 list2 ...) *procedure*

proc は list の個数と同じ個数の引数を受けつける手続である。2 個以上のリストが与えられるならばそれらの最小の長さの回数だけ繰りかえしが行なわれる。proc は各リストの先頭要素を並べた引数に対し実行され、順に実行した結果を 1 個のリストにして返す。

(**for-each** proc list1 list2 ...) *procedure*

map と同様だが、返す値は未規定である。

[SS|SRFI]

(**pair-map** proc list1 list2 ...) *procedure*

repeats for list1 ... and successive CDR's.

(**pair-for-each** proc list1 list2 ...) *procedure*

(**filter** pred list) *procedure*

(**partition** pred list) *procedure*

values (filter pred list) and for not-pred.

(**find-tail** pred list) *procedure*

(**append!** <list> ...) *procedure*

(**pop** <var>) *macro*

```
(pop x):= (let* ((temp (car x)))
              (set! x (cdr x))
              temp
          )
```

つまり変数 x の値 (リストである) の car をとりだし,x の値を cdr に変更する。
car が返される。

(**push** obj <var>) *macro*

(push s x):=(begin (set! x (cons s x)) x)

つまり、x の値 (リストである) の先頭に obj をつけくわえたものに x の値を
変更する。変化したリストが返される。

(**reverse!** <list>) *procedure*

destructive

(**reverse*** <list>) *procedure*

= (apply list* (reverse x))

(**rassq** x <list>) *procedure*

dot-pair の cdr が x と eq?である pair を返す。

(**rassv** x <list>) *procedure*

eqv?

(**rassoc** x <list>) *procedure*

equal?

(**remq** x <list>) *procedure*

list の要素のうち x と eq?であるものをすべて除いた新しいリストを返す。

(**remv** x <list>) *procedure*

eqv?

(**remove** x <list>) *procedure*

equal?

(**last-pair** obj) *procedure*

obj が (非真正)list のとき、最後の pair(cons cell) を返す。それ以外は#f を返す。

user >(last-pair '(3 4 5 6))
(6)
user >(last-pair '(3 4 . j))
(4 . j)
user >(last-pair ())
#f

(**last** obj) *procedure*

=(car (last-pair obj))

it errors If obj is not a list.

(**make-list** n [fill]) *procedure*

(**list-copy** <list>) *procedure*

(**circular-list?** x) *procedure*

(**dotted-list?** x) *procedure*

x is improper list?

(**not-pair?** x) *procedure*

=(not (pair? x))

(**list=** proc list1 ...) *procedure*

Determines list equality, given an element-equality procedure <proc>. list1 ... may be a simple circular list, which is a circular list but there exists no paths from any CAR's of it to the body of it(itself or CDR's). If <proc> is eq?, then don't mind this restriction.

(**length+** x) *procedure*

returns #f for the circular list and error for the improper list.

(**length++** x) *procedure*

returns length , length to the last-pair for improper list or the length for the first circular CDR of x.

(**take** <list> n) *procedure*

returns the first n elements of <list>.

(**take!** <list> n) *procedure*

(**split-at** <list> n) *procedure*

:= (values (take <list> n) (drop <list> n))

(**split-at!** <list> n) *procedure*

(**append-reverse** head tail) *procedure*

:= (append (reverse head) tail)

(**append-reverse!** head tail) *procedure*

1.7 関数

[R5RS+]

(**procedure?** f) *procedure*

f が関数 object, またはその名前なら真を返す。

[SS]

(**tk-procedure?** f) *procedure*

f が TCL/TK 関数なら真を返す。

(**special?** f) *procedure*

f が特殊形式またはその名前 (symbol) なら真を返す。

(**macro?** f) *procedure*

f が symbol で、syntax-macro のときは f 自身 (symbol) を返す。f が symbol で、classic-macro なら macro-object を返す。もし f が classic-macro-object であればそれを返す。これら以外は#f である。

(**procedure-info** f) *procedure*

f が special, classic-macro, 関数のときその情報を次のようなリストにして返す。

(type para rest func-env address)

type は

special

macro

tk　tcl/tk

system　ss 内部関数

lambda　define によって作られた関数

のどれかである。para は整数で必須パラメータの個数。rest は、追加パラメータが許されるとき 1 その他は 0 である。func-env は、special のときは自分自身でありその他は closure 環境である。address は内部アドレスを示す。

注 classic-macro は、syntax-macro に再定義されているかもしれない。

(**function-name** f)　*procedure*

f が関数で名前を持つときそれを返す。それ以外は#f を返す。tcl/tk,system および define で作られた関数 object はそのときの名前を保持している。

(**function** s)　*procedure*

s が symbol ならその値を再帰的に調べる。最終的に得られた関数 object を返す。そうでないときは error である。

(**funcall** x p1 ...)　*procedure*

これは

(apply (function x) (list p1 ...))

または

((function x) p1 ...)

と同じである。

例 1.6. *user* >(define x '+)

x

user >x

+

user >(x 3 4)

---> error!!!

user >((function x) 3 4)

7

user >(funcall x 3 4)

7

user >(function-name x)

+

1.8 [R5RS]eq? eqv? equal?

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

x,y がメモリー上で同じアドレスを指す object か同じ register 値であれば真を返す。[SS] においてはシンボル、キーワード、#t,#f,(), 文字、小整数 (FIXNUM) は eq?で判別できる。これは最も高速な判定である。

(**eqv?** x y) *procedure*

もし x,y が eq?で#t なら#t である。また x,y が同じタイプの数で等しいなら$t を返す。

(**equal?** x y) *procedure*

もし x,y が eqv?で#t なら#t である。また x,y が同じタイプのリスト、ベクタ、文字列であって、その要素がそれぞれ equal?で同じであるなら#t を返す。

1.9 [SS] math

ss では integer,rational,float,complex が使用できる。R5RS では、すべての数に exact,inexact の区別があるがこの system ではそれは採用していない。exact は単純に integer,rational を指すことにする。complex が exact であるのはその実部、虚部が共に exact の場合である。integer は、system 内部では小正数 (FIXNUM) と無限長整数 (BIGNUM) がある。FIXNUM は 30bit で表現できる整数でありこれは単に register 値である。FIXNUM の範囲は-536870912 から 536870911 である。vector の size などは FIXNUM で表現できる範囲でなければならない。FIXNUM の比較には eq?を用いることができる。他の数はすべて、メモリー上に置かれた 構造体であり、eqv?,equal?,=,< ... で比較を行なう。

(注) bignum,rational の演算には GMP(GNU MP library) を用いている。GMP の開発者に深く感謝する。float については将来採用できるかもしれない。

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

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

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

(**number?** x) *procedure*

これらは、包含関係になっている。number?はすべての数に対して#t を返す。一般に異なる型の演算結果は広いクラスの型で返される。integer と float の積は float になる integer と integer の除算は rational である。rational をサポートしない scheme のプログラムを移植するときは注意が必要である。

(**fixnum?** x) *procedure*

(**bignum?** x) *procedure*

(**ratio?** x) *procedure*

(**float?** x) *procedure*

(**complex?** x) *procedure*

これらは、個別の型を判定する。

以下において型の指定のない関数はすべての数に対して使用できる。

(**zero?** x) *procedure*

(**positive?** <real>) *procedure*

(**negative?** <real>) *procedure*

(+ x1 ...) *procedure*

(-x1 ...) *procedure*

(∗ x1 ...) *procedure*

(/ x1 ...) *procedure*

(**1+** x) *procedure*

(+ x 1)

(**1** -x) *procedure*

(- x 1)

(= x ...) *procedure*

(< <real> ...) *procedure*

(> <real> ...) *procedure*

(<= <real> ...) *procedure*

(>= <real> ...) *procedure*

(**max** <real> ...) *procedure*

(**min** <real> ...) *procedure*

(**abs** x) *procedure*

(**magnitude** x) *procedure*

= (abs x)

(**sign** <real>) *procedure*

(**even?** <integer>) *procedure*

(**odd?** <integer>) *procedure*

(**isqrt** <integer>) *procedure*

(**quotient** <integer1> <integer2>) *procedure*

整数型割り算

(**modulo** x y) *procedure*

x-(floor x/y)∗y

(**remainder** x y) *procedure*

x-(truncate x/y)∗y

(**gcd** <integer> ...) *procedure*

(**lcm** <integer> ...) *procedure*

(**sin** x) *procedure*

sin x

(**cos** x) *procedure*

cos x

(**tan** x) *procedure*

tan x

(**exp** x) *procedure*

exp x

(**expt** a p) *procedure*

a^p

(**sqrt** x) *procedure*

sqrt x

(**log** x) *procedure*

log_e x

(**log10** x) *procedure*

log_10 x

(**log2** x) *procedure*

log_2 x

(**cosh** x) *procedure*

cosh x

(**acosh** <real>) *procedure*

arccosh x

(**acos** <real>) *procedure*

arccos x

(**acosh** <real>) *procedure*

arccosh x

(**asin** <real>) *procedure*

arcsin

(**asinh** <real>) *procedure*

arcsinh x

(**atan** <real-x> [real-y]) *procedure*

arctan real-x or arctan real-x/real-y

(**atanh** <real>) *procedure*

arctanh x

(**floor** real-x [real-y]) *procedure*

(**ffloor** real-x [real-y]) *procedure*

(floor x y)->(floor (/ x y)) (floor x) は x を越えない最大の整数を返す。ffloor は引数を float に変換してから同様のことを行なう。返す値の型は float である。

(**round** real-x [real-y]) *procedure*

(**fround** real-x [real-y]) *procedure*

(round x) は x に最も近い整数を返す。

(**truncate** real-x [real-y]) *procedure*

(**ftruncate** real-x [real-y]) *procedure*

(truncate x) は絶対値が x の絶対値を越えない x に最も近い整数を返す。

(**ceiling** real-x [real-y]) *procedure*

(**fceiling** real-x [real-y]) *procedure*

(ceilling x) は x より小さくない最小の整数を返す。

(**cis** <real>) *procedure*

cos x+ i sin x

(**angle** x) *procedure*

arctan imag/real

(**make-polar** radius angle) *procedure*

(**denominator** <ratio>) *procedure*

分母

(**numerator** <ratio>) *procedure*

分子

(**rational** <real>) *procedure*

実数を rational に変換する。

(**rationalize** <real> error) *procedure*

実数を error の誤差の範囲で最も単純な rational に変換する。

(**imag-part** x) *procedure*

(**real-part** x) *procedure*

(**conjugate** x) *procedure*

(**complex** real imag) *procedure*

(**make-rectangular** real imag) *procedure*

複素数を返す。

(**float** <real>) *procedure*

x を float にして返す。

(**exact->inexact** x) *procedure*

x を float にして返す。x がは複素数ならば実部、虚部を float にして返す。

(**inexact->exact** x) *procedure*

x を rational にして返す。x は複素数でもよい。

(**exact?** x) *procedure*

(**inexact?** x) *procedure*

(**random**) *procedure*

0.0<x<1.0 の乱数を返す。

(**random** n) *procedure*

0 ... n-1 を返す。

(**random-seed** x) *procedure*

乱数を初期化する。

1.10　[SS] inc dec macro

(**inc** var [n]) *macro*

= (set! var (+ var n)) n=1 is default.

(**dec** var [n]) *macro*

= (set! var (- var n)) n=1 is default.

[SS] math constant

数学定数

```
PI          3.1415927
PI/2        1.5707963
IMAG        +i
LOG2        0.69314718  = (log 2)
LOG10       2.3025851   = (log 10)
```

1.11 [SS] math logical bit 演算

以下のような関数がある。以下で、x,x1 などは整数で無限長 bit を持つとして扱われる。たとえば

1=.....0000000001
-1=.....1111111111

である。

(**logand** x1 x2 x3 ...) *procedure*

and

(**logior** x1 x2 x3 ...) *procedure*

or

(**logxor** x1 x2 x3 ...) *procedure*

xor

(**lognot** x) *procedure*

すべての bit を反転する。

(**logcount** x) *procedure*

x が正ならば、その中の bit 1 の個数を返す。x が負のときは、(lognot x) の bit 1 の個数を返す。

(**integer-length** x) *procedure*

x が正ならば、最上位の 1 が何番目の bit かを返す。x が負のときは、(lognot x) のそれである。*user* >(integer-length 251)

8

(**ash** x shift) *procedure*

shift が正なら x を左にシフトする。shift 分の 0 が入る。shift が負なら x を右にシフトする

(**lowest-bit** x) *procedure*

(logand x (logxor n (- n 1))) と同じ。つまり、x の最下位の 1 のみを持つ数を返す。*user* >(lowest-bit 160)

32

(**highest-bit** x) *procedure*

x が正ならば、最上位の 1 のみを持つ数を返す。x が負のときは、(lognot x) のそれである。

1.12　after [SS]

TCL/TK の after の代わりに scheme 自体の after を使うことができる。従って、以下は sn でも使用できる。scheme では

(**after** msec [command]) *procedure*

msec=mili sec

msec 後に command を実行する。この関数は予約 id(整数) を返す。command は、引数なしの lambda 式または S 式である。after で予約できるのは最大 99 個まで。command が与えられなければ、msec だけ sleep して、undef を返す。

(**after-info**) *procedure*

情報を表示する。

(**after-cancel** id) *procedure*

予約を取り消す。id の予約がないときは#f を返す。

(**update**) *procedure*

timer の切れた event の処理をする。(read-line) や toplevel で、キー入力待ちのときは自動的に呼び出される。(ss では、同時に TCL/TK の update も呼び出される。)

(**tk:update**) *procedure*

TCL/TK のみの update をする。ss timer 処理をしたくないときはこれを使う。

例 1.7. *user* >(after 100000 '(write "test"))

0

user >(after-info)

interval 100000 time_rest 36400 past 63600 sec 18

sys_alarm 0

0:(write "test") 36400

<#undef#>

user >(after-cancel 0)

#t

1.13　[ss] eval env

(**eval** form [env]) *procedure*

form を与えられた環境で評価する。環境は、局所変数の連想リストである。env の省略値は () である。これは toplevel における評価を意味する。eval に define 式を与えることはエラーである。

(**interaction-environment**) *macro*

これは、局所変数の環境を返す。

(**scheme-report-environment** 5) *procedure*

(**null-environment** 5) *procedure*

これらはいずれも単純に () を返す。

1.14 [SS] alloc

alloc は、32bit のデータを 1 単位として、メモリーを確保する。alloc data 構造は lisp object であるが、その中味は任意である。特に、gbc はその中味には無関心である。また print 関数はそのアドレスと size のみを表示する。lisp object は、32bit のポインタなのでそれを格納してもよいし、他の何であってもよい。

一般的なプログラムには適さない。

(**alloc** size) *procedure*

(**alloc-ref** alloc position) *procedure*

(**alloc-set!** alloc position obj) *procedure*

(**alloc-length** alloc) *procedure*

[SS] obj-address

(**obj-address** obj) *procedure*

obj を指す内部ポインタを整数値の文字列として返す。これは一般に 32bit 整数である。

(**obj-print** integer) *procedure*

例 1.8. *user* >(obj-address #t)

”167044836”

user >(obj-print 167044836)

#t

<#undef#>

第２章　schemeの構文

2.1　特殊形式

ss における特殊型式と基本マクロは以下のとおりである。

(**define** var value) *special form*

(**define** (func . <bind-vars>) . body) *special form*

関数定義

(**set!** var form) *special form*

form は評価され変数 var に代入される。変数は、define または let∗ などであらかじめ利用可能になっていなければならない。

(**begin** form ...) *special form*

form ... を順に評価する。最後に評価した式の値が返される。

(**quote** form) *special form*

= 'form
この式を評価すると form そのものが返される。これはあるテータを関数にそのまま渡したいときに多用される。キーワード、数、文字、文字列、#t,#f, ベクタなどは即値データでありクオートする必要はない。

(**let**∗ ((var init) ...) . body) *special form*

(**let** ((var init) ...) . body) *macro*

let∗ を用いたマクロである。

(**let** name ((var init) ...) . body) *macro*

letrec を用いたマクロである。名前付き let と呼ばれ、loop を構成する。

(**letrec** ((var init) ...) . body) *special form*

(**let-syntax** rules . body) *macro*

(**letrec-syntax** rules . body) *macro*

(**call/cc** proc) *procedure*

(**if** <cond> <then> [<else>]) *special form*

(**lambda** <bind-vars> . body) *special form*

(**macro** <bind-vars> . body) *special form*

classic macro を作る。

system 用

(**LOAD-REG** <register>) *special form*

(**TAIL-CALL** [x]) *special form*

SS の compiler は末尾呼び出しをサポートしている。関数の実行の最後が他の関数の呼び出しのときそれを末尾呼び出しという。この場合、呼び出しは実際には Jump で行なわれる。[R5RS 3.5] tail recursive を見よ。

2.2 let let∗ letrec

(**let**∗ ((var init) ...) form ...) *special form*

局所変数を用いたブロック構造を作る。init が評価されて、局所変数 var に bind される。bind 評価は先頭から順に行なわれる。後続の bind では先行する bind を見ることができる。すべての bind のあと、その環境のもとで form が順に評価されて最後に評価した式の値を返す。

(**let** ((var init) ...) form ...) *macro*

let∗ と似ているが、初期値はすべてあらかじめ計算されてから変数に bind される。 [SS] では、これは let∗ のマクロである。

(**let** name ((var init) ...) form ...) *macro*

繰りかえし構造を作る。name は bind 部の各変数を引数にもち本体が form ... であるような関数に bind される。本体で (name ...) を再帰的に呼び出すことにより繰りかえしが実現できる。[SS] ではこれは letrec に展開される。

(**letrec** ((var init) ...) form) *special form*

すべての bind 変数は、最初にある未規定の値に bind される。そのあとすべての init が計算されて各 bind 変数に set される。このため任意の初期値は相互に他の局所変数を参照できる。しかし、初期値の計算が陽な bind 変数の値の参照や代入を伴うならそれは一般的にエラーである。変数が最初に持っている値は未規定だからである。このような参照が意味をもつのは初期値が lambda 式であり、その中で bind 変数が関数呼び出しとなる場合である。letrec は、再帰呼び出しにより繰りかえし構造を作る。[SS] ではすべての繰りかえし構造は letrec を使う macro である。

例 2.1.

```
  (define (fact n)
(letrec ( (sub (lambda(x y) (if (= y 0) x (sub (* x y) (- y 1) )))))
      (sub 1 n)
)
)
```

は階乗関数を定義する。sub は再帰的に呼び出される。もしこの定義で letrec の代わりに let または let∗ を使ったらどうなるだろう? 初期値の lambda 式を評価するとき変数 sub は存在しないので、その参照は toplevel における値の参照になってしまう。lambda 式を評価した結果はその環境を含めたクロージャだからである。実行しても sub は関数ではないというエラーに終るだろう。letrec では、最初に sub は local 変数として定義されて、その後で lambda 式の評価が行なわれるので再帰が可能になる。なお、ここで採用した定義は sub が末尾再帰呼び出しになっている。この呼び出しはスタックを浪費しない。

2.3 define set!

(**define** var form) *special form*

これが toplevel において現われるとき (toplevel における begin, if の top でもよい。) form を評価した値が初期値であり、変数 var の global 値として set する。define されていない変数に対する set!はエラーである。もし var が既に define されているならこの式は set!と同じである。

内部 define (R5RS 5.2.2)

define を lambda,let,let∗,letrec,let-syntax,letrec-syntax の本体の先頭部におくことが許される。これは局所変数の bind を発生するがそれは、全く letrec で等価な形で置き換えられる。つまり

```
(let ((x 0))
   (define foo ...)
   (define goo ...)
    ...
)
```

と

```
(let ((x 0))
   (letrec (( foo ...)
            ( goo ...))
    ...
   )
)
```

は等価である。従って、foo,goo の初期値部分に foo,goo の値の参照、代入を書くのはエラーである。このことは、内部定義は局所関数の定義のみに用いるのが安全ということである。

```
(let ((x 0))
   (define foo x)
   (define goo foo)
   (write goo)
)
```

は goo が foo を参照しているのでエラーである。define 式が本体の途中に現われるのはエラーである。

(define (name . <vars>) form ...)
これは
(define name (lambda <vars> form ...))
と等価である。

2.4 [R5RS] syntax macro

多くの実例と詳しい説明は R5RS および sslib/syntax.ss を見て欲しい。

(**define-syntax** <macro-key> *macro*

```
(syntax-rules (<literal> ...)
   ( <pattern> <template> ) ...
)
)
```

R5RS 仕様のマクロ (syntax-macro と呼ぶことにする。) を定義する。toplevel でのみ使用できる。このマクロ定義は macro-key で指定したシンボルの値にはならない。syntax-macro 定義は、その定義が行なわれた環境を保存した形で行なわれる。つまり定義中に表われるシンボルの持つ関数、マクロ、特殊形式としての性質は定義のときの性質をそのまま保存される。実際にはこれらは改名されてもとの名前と値の情報と共に保存される。従って、定義のあとで環境が変化してその新しい環境をマクロ定義に反映させたい時は、macro-update を用いなければならない。

(**let-syntax** rules . body) *macro*

```
rules:=( (key (syntax-rules (<literal> ...)
                    ( <pattern> <template> )
                      ...
                )
          )
         ...
       )
```

(**letrec-syntax** rules . body) *macro*

2.5 SS の古典マクロと拡張機能

(**define-macro** (name para ...) . body) *macro*

古典的なマクロ定義 (classic-macro) である。define-syntax の利用を勧める。

(**macro-expand-rec** form env) *procedure*

form を再帰的にマクロ展開する。form をマクロを含まない式に変換した結果が返される。

(**special-syntax** <macro-key> *macro*

```
(syntax-rules  (<literal> ...)
   ( <pattern> <template> ) ...
)
)
```

これは define-syntax と同じ構文をもち、toplevel でのみ使用できる。通常の syntax-macro は、パターンマッチが失敗するとエラーになるのに対し special-syntax の場合はエラーにならず、それが変換の必要のない最終形であると判断する。従って、通常この macro-key は特殊形式または関数でもある。あるいは単なるデータ変換に用いてもよい。

(**get-macro** key) *procedure*

key が syntax-macro のときその定義内容を返す。

(**set-macro** data) *procedure*

data は get-macro で得たものでなければいけない。

(**remove-macro** key) *procedure*

syntax-macro を取り消す。

(**macro-update** <macro-key> ...) *procedure*

すでに定義されている syntax-macro を現在の環境を反映した形に変換する。1 つの重要な例はマクロの相互参照を可能にすることである。たとえば、A,B の 2 つのマクロが互に相手を使っており、定義が A、B の順に行なわれるとしよう。A の定義のときは B はまだ定義されていないので、A の展開は B が未定義の関数であるという環境のもとで行なわれてしまう。このような場合は、A、B を定義したあとで (macro-update 'A) を実行しなければいけない。同じ効果は A,B の定義の後で再度 A を定義しても得られるが、あまり美しい方法ではないであろう。もっと、簡単な方法は参照したいマクロを literal に指定することである。

(**macro-symbols**) *procedure*

すべての syntax-macro のシンボルのリストを返す。

2.6 [R5RS] syntax-rules literal pattern template

syntax-rules を構成する literal, pattern, template について解説する。

(<literal> ...) *syntax*

これはシンボルのリストであり、ここで指定されたシンボルはpattern,template 中で変化せずそのまま現われるものと解釈される。以下、これを LIT で表す。

... *syntax*

シンボル"..."は省略符合と呼ばれ、直前に置かれたパターンが 0 回以上あることを示す。シンボル ... は syntax-rule においてこれ以外の意味で使用することはできない以下 DOT と呼ぶ。

pattern *syntax*

pattern はリストまたは vector であって、その先頭はマクロキーワード (以下 KEY と呼ぶ) である。pattern の構成要素は任意の S 式からなり、特にシンボルと... が重要である。... はリストまたは vector の構成要素として現われる。pattern に現われるシンボルで KEY,LIT, DOT 以外のものをパターン変数 (PAT) と呼ぶ。

たとえば cond の定義は

```
(define-syntax cond
    (syntax-rules (else)
      ((cond) #f)
      ((cond (else y ...)) (begin y ...))
      ((cond (test y ...)) (if test (begin y ...)))
      ((cond (test y ...) clause ...)
          (if test (begin y ...)
             (cond clause ...)
          )
       )
 ))
```

である。cond は KEY で else は LIT である。第 2 の pattern

(cond (else y ...))

において y は PAT である。弟 3 では test, y が弟 4 では test,y,clause が PAT である。ある S 式をマクロ展開する時、その式とパターンマッチングが行なわれる。それが KEY から始まるとき、定義のパターンと照合される。定義の順に照合を行ない最初に照合したパターンに対する template を用いて変換が決定される。もしすべての照合が失敗すればエラーである。照合は次のように行なわれる。PAT は任意の式 (!!!) と照合する。1 つの pattern の中に同じパターン変数が 2 回現われるのはエラーである。 DOT があるとき、それは直前の部分パターンの 0 回以上の繰りかえしと照合する。部分パターンがリストなら照合するものもリストである。パターンに現われる、シンボル、リスト、ベクタ以外のデータは対応する S 式の部分が equal?で等しいとき照合する。たとえば pattern

(foo (a ...) (b ...)... c ...)

と S 式

(foo (3 (x)) (4 5 6) () (h (y)) +)

は照合する。PAT 変数 a には、２個のデータ 3, (x) が対応している。変数 b は複雑で 2 重になっている。b は {4,5,6}, {},{h,(y)} のように照合している。(4 5 6), (), (h (y)) のどれもが (b ...) と照合するからである。c には+が照合している。+は (b ...) とは照合しない。もし template が

(list c ... a ... ((- b) ...)...)

であれば、変換の結果は

(list + 3 (x) ((- 4)(- 5)(- 6)) () ((- h) (- (y))))

のようになる。pattern において多重となっている PAT 変数は、template においてはそれと同じかより da も同じ多重をもたなければならない。syntax-macro では名前の衝突などの問題が発生しない、また定義時点での環境の保存が保証されている。すなわち、template 中にある KEY,LIT,PAT 以外のシンボルは実質的な改名と、もしあればその定義を伴って変換される。LIT で指定したシンボルはそれが、展開する S 式の中やその環境において別な束縛を持たない限り改名はされない。逆に別な束縛を持つときは、そもそも照合のときもそれは LIT とはみなされない。たとえば

```
(let* ((else #f))
  (cond (else 3)
        (#t 10)
   )
)
```

においては else は単なる test 変数であり、この式は値 10 を返す。詳細は [R5RS] を見よ。

2.7 [R5RS] call/cc call-with-current-continuation dynamic-wind

(**call-with-current-continuation** proc) *procedure*

(**call/cc** proc) *procedure*

proc は 1 引数の手続きであり、この式の出現の継続を行なう脱出手続がその引数に bind される。脱出手続には 1 個以上の引数を渡してよい。一般に継続は多値となる。

例 2.2. *user* >(define x (values->list (call/cc (lambda(x) x))))

x

user >x

(#[closure system 1 1 #[subraddress 807a588](#[cc] . #[jmp a19ef14])])

user >(apply (car x) '(3 4 5))

x

user >x

(3 4 5)

この例では、最初に call/cc を呼んでいるが、proc が (lambda(x) x) なのでそれは、単に脱出手続を返す。values->list は多値を 1 つのリストに変える関数 (ss 独自のもの) なので、結局 x には脱出手続 1 個からなるリストが入る。apply を用いてこれを引数 (3 4 5) に対して実行すると、system の実行環境はそれを作った時点の継続となる。従って、多値 3,4,5 が生成され values->list によってリスト (3 4 5) が作られて define により変数 x の値は (3 4 5) に変化する。apply の返り値が x なのは、それが define の返り値だからである。call/cc と脱出手続は極めて強力な手続であり、任意の場所、任意の時点で実行できる。

(**dynamic-wind** before thunk after) *macro*

before thunk after は引数なしの手続である。この式の呼び出しが返す値は thunk の返す (多) 値である。before,after は thunk の前後に呼び出される。もし、脱出手続により thunk の中に戻るなら、やはり before が呼ばれ、thunk における継続のあと after が呼ばれる。そして thunk における継続の値が返される。

2.8 多値

[R5RS] values

(**values** a ...) *procedure*

は

(call/cc (lambda(exit)(exit a ...)))

と同等である。

ss では、0 個の多値は許さない。

(**call-with-values** vproc cproc) *procedure*

```
(call-with-values
   (lambda() (values 1 2 3))
   (lambda( x y z) (+ x y z)))
```

[SS]

(**values->list** [<form>]) *procedure*

form が省略されたときは、その直前における多値を list にして返す。form が与えられたときは、その form を評価したときに得られる多値を list にして返す。

関数 values->list は

(begin (values 2 3 4)(values->list))
(values->list (values 2 3 4))

のどちらも値 (2 3 4) を返す。つまり

(begin <form> (values->list))
(values->list <form>)

は同等である

(**multiple-value-set!** var1 ... proc) *macro*

```
(let* ((x 0)(y 0)(z 0))
    (multiple-value-set! x y z (values 1 2 3))
    (+ x y z)
 )
```

[SRFI-8]
(**receive** formals expression body ...) *macro*

```
(receive (x y z)
    (values 1 2 3)
     (+ x y z)
  )
```

2.9 [R5RS 4.1.4] lambda

(**lambda** <vars> form ...) *special form*

<vars> を仮引数にもつ関数 object を作る。これは、関数クロージャとも呼ばれ評価した時点の環境を保持する。<vars> は次の形である

(1) シンボルのリスト
`(x y z)` 3 個の引数を指定する。
(2) 1 個のシンボル
これは任意個数の引数を受けつけてそのリストを bind する。

```
((lambda x x) 1 2 3 4)   => (1 2 3 4)
```

(3) シンボルの非真正リスト
(x1 ... xn . y) は n 個以上の引数をとる。n 個までは x1 ... xn に bind され残りのリストが y に bind される

```
((lambda (x . y) y) 1 2 3) => (2 3)
```

作製された 関数 object の 呼び出しにおいては、まず lambda 式を作成した時点の環境がとりだされる。そして仮引数に実引数が bind されて環境に付け加えられる。この環境において本体の式が順に評価され、最後に評価した値が返される。

2.10 [R5RS] cond and or

(**not** x) *procedure*

(**and** form ...) *macro*

(**or** form ...) *macro*

(**cond** (test form ...) *macro*

```
    ...
  [ (else form ...) ]
)
```

(**when** test expr ...) *macro*

(**case** key *macro*

```
  ((key1 ...) form ...)
   ...
  [ (else form ...) ]
)
```

2.11 [R5RS] do while

(**do** ((var init step ...) ...) *macro*

```
      (test expr ...)
      command
      ...)
```

(**while** test expr ...) *macro*

(**foreach** var lis expr ...) *macro*

```
=  (do ((var 0) (temp lis (cdr temp)))
      ((null? temp))
      (set! var (car temp))
      expr ...
    )
```

2.12 [R5RS-] quasiquote backquote

バッククオート構文はリストに対してのみ用いることができる。(R5RS では vector も可能になっている。)

‘(x ,y ,@z)

は

(quasiquote (x (unquote y) (unquote-splicing z)))

と同等である。最初の式は read-macro 機能により qq 構文に変換される。quasiquote はマクロであり、この式のマクロ展開は

(append (list (quote x)) (list y) z)

となる。シンボル unquote,unquote-splicing を qq 構文以外で使うのは一般にエラーとなる。これらは quasiquote によってのみ解釈される。

第3章　構造体

構造体 (structure) はいくつかの要素からなる 1 つの data type である。これは、vector に似ているが、それぞれの要素には名前がついていて、名前によってアクセスができる。

下の defstruct はある名前の構造体を定義する。それはいくつかの要素からなり、各要素には名前がついている。defstruct は、マクロであり、構造体を生成する make-<name>, 値を参照するための <name>-<slot>, 値を変更する set-<name> 判定する <name>?, コピーする copy-<name> のマクロを自動的につくりだす。このため、system 関数を書き変える危険性があるので structure の名前は大文字を使用するのが望ましい。defstruct は toplevel で実行しなければならない。

3.1　defstruct マクロ

(**structure?** x)　　*procedure*

x が structure ならその structure の名前を返す。それ以外は#f を返す。

(**defstruct** name slot-1 slot-2 ....)　　*macro*

name は symbol (structure の名前) かまたは

(name :include sub-struct-name ...)

である。この場合、sub-struct-name のすべての slot を含めた形で構造体 name は定義される。slot-1 などは、キーワードまたは

(key-word init-value)

の形である。init-value は make-<name> の呼出しのたびに評価される。

例 3.1. *user* >(defstruct Person :name (:age 0))

(((Person) (get-file:GETF)) make-Person make-s-Person set-Person ref-Person Person)

user >(make-Person :name 'sato)

#s(Person :name sato :age 0)

user >(defstruct (Student :include Person) (:class 1))

(((Student (Person . 1)) (Person) (get-file:GETF)) make-Student make-s-Student set-Student ref-Student Student)

user >(make-Student)

#s(Student :class 1 :name <#undef#> :age 0)

のようになる。各要素への参照アクセスは関数 <name>-<slot-name> でおこなう。たとえば

例 3.2. *user* >(define x (make-Student :name 'sato))

x

user >(Student-name x)

sato

値の変更は関数 set-<name> で行なう。

例 3.3. *user* >(set-Student x :age 51 :name 'koike)

<#undef#>

user >x

#s(Student :class 1 :name koike :age 51)

値の指定は、常にキーワードと値の組であり、複数個書いてもよい、また順番を気にする必要もない。student は person を:include しているので、これに対して person のアクセス関数を用いてもよい

例 3.4. *user* >(set-Person x :name 'sasaki)

<#undef#>

user >x

#s(Student :class 1 :name sasaki :age 51)

defstruct は、関数 <name>? , copy-<name> も作成する。

例 3.5. *user* >(Student? x)

#t

user >(copy-Student x)

#s(Student :class 1 :name sasaki :age 51)

この copy は structure-data を作り、そこにもとのデータを置くだけであり、各要素の copy が作られるわけではないことに注意してください。

Lisp reader は、#s(name ...) または #S(name ...) を受けつける。従って、ファイルに書いた structure をそのまま読むことができる。ただし、その structure は前もって defstruct しておかなければならない。

3.2　同じ名前の slot, 構造体の:include

ss では、同じ名前の slot や構造体の:include が許されている。このようなときは、アクセスに shift を指定する必要がある。ただし、shift の指定は整数値 0,1,2,... のみが許されていて、変数名で指定することはできない。もし、変数でアクセスする必要があるのなら、それは構造体を要素とする vector を用いるべきである。なお、include される構造体の slot は直接指定の slot より後に配置される。 また、include が多重になっていても良い。

例 3.6. *user* >(defstruct (Pair :include Person :include Person) (:name 'foo)
)

(((Pair (Person . 1) (Person . 3)) (Student (Person . 1)) (Person)
(get-file:GETF)) make-Pair make-s-Pair set-Pair ref-Pair Pair)

user >(define x (make-Pair))

x

ここで Pair は 2 個の person からなる構造体である。なにも指定がなければアクセス関数は最初の person に作用する。2 番目にアクセスするには shift に 1 を与えなければならない。たとえば次のようになる

例 3.7. *user* >(set-Pair x :age 35 (:name 1) 'tokuda (:name 2) 'sakata)

<#undef#>

user >x

#s(Pair :name foo (:name 1) tokuda :age 35 (:name 2) sakata
(:age 1) 0)

3.3 基本関数

(**subst-structure** struct dest [shift]) *procedure*

struct を dest に代入する。dest が struct と同じ名前の構造体なら dest 全体の値が変化する。もし、dest が struct を:include しているならば、その部分が書き変えられる。

(**copy-structure** struct [sub-struct [shift]]) *procedure*

struct の copy を返す。もし sub-struct(名前) が与えられていれば struct が include している sub-struct の copy が返される。

(**structure?** struct) *procedure*

名前または#f を返す。

(**structure-name** struct) *procedure*

名前を返す。 struct が structure でなければ error である。

第4章　Package

4.1　symbol

(**string->symbol** str)　　*procedure*

(**symbol->string** symbol)　　*procedure*

[SS] symbol
(**constant!** symbol)　　*procedure*

symbol を定数にする

(**unconstant!** symbol)　　*procedure*

(**constant?** symbol)　　*procedure*

例 4.1. *user* >PI

3.1415927

user >(constant? 'PI)

#t

(**symbol-home** symbol)　　*procedure*

(**symbol-bind?** symbol)　　*procedure*

symbol が値を持っていれば真

(**symbol-value** symbol)　　*procedure*

eval より効率的

(**symbol-value-set** symbol value)　　*procedure*

Danger! No error for undefined symbol

(**symbol-value-set!!!** symbol value)　　*procedure*

Danger!!! No check, simply ATOMVAL(symbol)=value

例 4.2. *user* >(symbol-home 'PI)

#[package ss]
user >(symbol-bind? 'foo)
#f

```
see package
       find-symbol
       find-export-symbols
       symbol-home
       find-all-symbols
       list-all-symbol
       unintern
see string
       tk-name
```

4.2 シンボルの完全形と package 機能

これは、名前（シンボル）空間を分離する機能である。Common Lisp の機能の一部 (に近いもの) を実装している。package は 1 つの型であり、実体は symbol の hash 表を含む構造体である。

ss の package は

sys system 用

ss ss の関数などこの中のシンボルはすべての package から省略形で参照できる。

user ss が start した後の通常の package

macro syntax-macro system 用

compiler system 用

alias system 用

tk TCL/TK 関数で、ss の関数名と衝突するものはここに入る。

が予約されている。現在使用されている package は、シンボル sys:*package* の保持する値である。 :の前の部分は package 名であり、後ろは印字名という。すべての、シンボルはこの完全形で参照できる。

現 package 内のシンボルと ss 内のシンボルは,package 部を省略した形で参照できる。これを、内部参照ということにする。

define,+,cons,car,....

などは、ss のシンボルなので普通に書けば良いのである。内部参照ができるのは、ss のシンボルと、現 package の中で登録されたシンボル（その symbol-home は現 package である。）および export,use-package で外部から輸入して登録されたシンボルである。ss および輸入された symbol を外的 (external), 内部で定義されたシンボルを内的 (internal) と呼ぶ。

完全形は一意ではないことに注意すべきである。たとえば ss:define , user:define は通常、同じ define を指している。完全形は、ある package を指定してその中でアクセス可能な指定した印字名を持つ symbol を要求する。１つの package の中で同じ印字名をもつ symbol はただ１つである。

また、ss の read 関数は、印字名のみのシンボルを読んだとき、もしそれが新しいものなら現 package に属すように自動的に登録する。(intern という。）もし、完全形なら、その package が存在していればそこに intern されるが、そうでなければ error である。

印字名が同じでも、異なる package に属す内的シンボルは違うものであり、これを利用して名前の衝突の可能性を低くできる。逆に、export,use-package を用いて重要なシンボルを他の package で内部参照可能にできる。

関数 load は、その前後において sys:∗package∗ の値を同一の値に保持する。

関数 display は、シンボルに対してその印字名だけを印刷する。write は、それが現 package で内部参照できないなら完全な形で印刷する。そのときの、package 部はその symbol の home-package である。

4.3 package 基本関数

以下で、name とあるのは文字列, または package である。package 引数の多くは単にその名前を使用できる。また、その省略値は現 package である。

sys:∗package∗ *variable*

(**package?** x) *procedure*

(**make-package** string) *procedure*

(**in-package** name) *procedure*

name の package が存在しないときは、make-package をして sys:∗package∗ の値を変更する。

(**find-package** string) *procedure*

string の名をもつ package を返す。

(**list-all-packages**) *procedure*

すべての package のリスト

(**package-name** package) *procedure*

名前を返す。

(**package-use-list** name) *procedure*

name の package が use している package のリストを返す。

(**package-used-by-list** name) *procedure*

name を使用している package のリストを返す。

(**string->symbol** string) *procedure*

string を印字名とするシンボルがすでに内部参照可能なら単にそれを返す。そうでなければ、現パッケージに新しくシンボルを作りそれを返す。

(**intern** string [package]) *procedure*

string を印字名とする内的なシンボルを登録する。すでに package で内的なら単にそのシンボルを返す。新しいシンボルが作られたときはそれは内的である。しかし、次の関数を使う方が良い。

(**shadowing-intern** string [package]) *procedure*

string を印字名とするシンボルを内的に intern する。すなわち登録されるシンボルの home は package である。もしその印字名が ss または export によって、すでに内部参照可能な場合そのシンボルは新しいシンボルによって shadow されることになる。 この新しいシンボルを unintern するともとのシンボルが再び参照可能になる。(ss のシンボルであるかまたは、その export,use-package の状態が変化していない限り。) shadow が起こる場合、新しいシンボルの持つ値は、それが shadow するシンボルの値と同じである。

(**find-symbol** string [name]) *procedure*

string を印字名とするシンボルがあればそれを返し、なければ#f を返す。

(**unintern** symbol [package]) *procedure*

symbol を表から削除する。 それが内的シンボルの場合 unintern されたシンボルは、home package を持たないシンボルになる。そのようなシンボルを write すると none:::ghost のように印字される。この形式は、read 関数で error になる。

(**export** symbol-or-list [package]) *procedure*

symbol を export する。 symbol は package において内部参照できなければならない。package をすでに use している package があれば、ただちにそこへ登録される。名前の衝突が起こるときは error が発生する。symbol は symbol（または文字列）のリストまたは 1 個の文字列でもよい。

(**unexport** symbol-or-list [name]) *procedure*

export を取り消す。name を use しているパッケージがあるときこの symbol がそこで export されていない限り、symbol は use していたパッケージから取り除かれる。

(**unexport-rec** symbol-or-list [name]) *procedure*

symbol の export が連鎖しているとき、再帰的に unexport する。

(**export?** symbol [name]) *procedure*

symbol が、name において export 宣言されたものであれば真を返す。

(**external?** symbol [name]) *procedure*

symbol が name において外的なら真を返す。

(**use-package** package [inpackage]) *procedure*

package の中で export されているシンボルをすべて inpackage に登録する。export, use-package において名前の衝突が起こるときは error が発生する。

(**unuse-package** package [inpackage]) *procedure*

use を取り消す。export されていたシンボルは inpackage の表から取り除かれる。inpackage が package を use していなかった場合は単に無視される。

(**find-all-symbols** string) *procedure*

string を印字名とするシンボルすべてのリストを返す。

(**map-package** f [name]) *procedure*

name の package 内部のすべてのシンボル x に対して関数 (f x) を実行する。

(**list-procedure** [name]) *procedure*

name の中のすべての関数のリストを返す。

(**list-tk-procedure**) *procedure*

すべての TCL/TK 関数のリストを返す。

(**list-package-symbol** [name] [test-func]) *procedure*

name の中のすべてのシンボルの (test-func が真になるもの) リストを返す。

(**list-export-symbol** [name]) *procedure*

name の中のすべての export シンボルのリストを返す。

(**list-all-symbol** [test-func]) *procedure*

すべての package のシンボルの (test-func が真になるもの) リストを返す。

(**symbol-home** symbol) *procedure*

symbol の home package を返す。

(**rename-package** name newname) *procedure*

注意。

ss package

これは、特別で、この中のすべての symbol は、他の package からは登録されているとみなす。(実際には、hash-table 登録はしない。) (use,export は無効である)。

sys package, alias package

逆に、これは use, export を禁止する。

tk

TCL/TK 関数で、ss の関数と名前の衝突するものはここに入っている。

tk:if,tk:info,tk:while,tk:glob

などである。これらのリストが、

tk:∗shadowing-tk∗

に入っている。 必要なら、これらは完全形で使用できる。

export はシンボルの属性ではない。

あるシンボルが export であるかどうかは、シンボルそのものの性質ではなく package に依存している。たとえば、A は B を、B は C を use しているとする。C において x を export すると x は B に現れるが、x は A には現れない。A から見えるようにするには、さらに B において、x を export する必要がある。これを自動化する、export-rec を定義することは可能だが一般には名前の衝突の危険を増すことになるであろう。これが、unexport-rec はあるのに export-rec を用意しない理由である。

package の階層化は定義されていないが、構築することは可能である。印字名の中に:を含ませることは error ではないが、推奨できない。package 名に:や. を含ませることも推奨できない。将来、. は階層化に使われるかもしれない。

Common Lisp との違い

import 機能はない。ss での export は、実際に表に登録するので CL の export と import を合わせたもののようになっている。

第5章　UNIX環境

ここでは、ss の Unix 環境に関連した機能を説明する。いくつかの基本関数は Unix の基本コマンドと同等である。

5.1　時刻、日付

(**times**)　*procedure*

ss が起動してからの秒数を実数で返す。

(**time**)　*procedure*

1970.1.1 からの秒数を整数で返す。これは、file-stat における時刻表示でも用いられている。

(**timer** <command>)　*macro*

command の実行時間を測る。macro なので、command はクオートしない。

(**date**)　*procedure*

現在時刻の文字列を返す。

5.2　process

(**system** str)　*procedure*

str は Unix コマンドの文字列であり、それを Unix shell により実行する。 終了コードが返される。

(**exec** str)　*procedure*

str を Unix shell により実行して、それが標準出力に送る結果を文字列にして返す。

(**run-process** cmd p1 ...)　*procedure*

cmd p1 ... は文字列で、cmd をパラメータ p1 ... で fork する。返り値は process-id と 終了状態のドット対である。引数に (位置はどこでも良い) キーワード :wait を置いた場合子プロセスの終了を待つ。正常に終了したときの返り値は undef になる。

(**process-exited?** pid)　*procedure*

pid は run-process の返した値である。pid のプロセスが終了していれば \#t (正常), 整数 (signal killed) 終了していなければ \#f を返す。pid の cdr は、この関数が最初に終了を知ったときに破壊的に変更される。

(**kill** pid [sig]) *procedure*

pid は run-process の返した値かまたは整数 (プロセス id) である。sig の default はSIGTERM = 15 である。SIGTERM,SIGHUP,SIGKILL はsslib/lib_base.ss で定義されているので、これらのシンボルを使ってもよい。

(**getpid**) *procedure*

ss 自身の process id を返す。

5.3 directory

(**startdir**) *procedure*

ss が起動したときの最初の directory を返す。

(**pwd**) *procedure*

(**getcwd**) *procedure*

この 2 つの関数は current working directory の文字列を返す。cygwin の場合、getcwd は”/cygdrive/c/cygwin/home/...” のように返す。pwd は”/home/...”” のようになる。

(**chdir** str) *procedure*

cd str

∗**PWD**∗ *variable*

current working directory の文字列を保持する。chdir が行なわれれば、値も更新される。

∗**HOME**∗ *variable*

ユーザーの home directory である。

5.4 環境変数

(**getenv** str) *procedure*

環境変数の値を取得する。

(**setenv** name value) *procedure*

5.5 file 操作

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

pat のどれかとマッチするファイル、dir のリストを返す。

例 5.1. *user* >(glob "k*.c" "q*.c")
("kanji.c" "key_init.c" "keyready.c" "qquote.c")

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

directory only

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

not directory file

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

symbolic link only

(**file-exists?** file) *procedure*

(**file-stat** file) *procedure*

これはリスト

(mode size time date)

を返す。mode は /usr/include/bits/stat.h で定義されている整数である。

(**file-is-directory?** file) *procedure*

(**file-is-regular?** file) *procedure*

(**file-is-executable?** file) *procedure*

(**file-is-slink?** file) *procedure*

symbolic link ?

(**read-slink** file) *procedure*

file がシンボリックリンクのとき、そのリンク先を返す。

(**file-size** file) *procedure*

(**file-time** file) *procedure*

1970.1.1 からの秒数

(**file-date** file) *procedure*

例 5.2. *user* >(file-stat "test.ss")
(33188 54 1084952713 "Wed May 19 16:45:13 2004")
user >(file-time "test.ss")
1084952713
user >(file-size "test.ss")
54
user >(file-date "test.ss")
"Wed May 19 16:45:13 2004"
user >(file-is-regular? "test.ss")
#t
user >(file-is-executable? "test.ss")
#f
user >(file-exists? "test.ss")
"test.ss"

(**pathname** file) *procedure*

これは、文字列 file を unix の full path 形式に変換して返す。~/, ~user/, ./, ../などを解釈する。また、それが存在する directory であるときは、最後に/のついた形で返される。file があらかじめ存在する必要はない。file が unix 形式の記述でないならこれは#f を返す。

例 5.3. *user* >(pathname "../../BSD")
"/home/sato/SSlisp1.96/BSD/"
user >(pathname "~sato/foo")
"/home/sato/foo"

(**find-source** file [flag]) *procedure*

これは load 関数の search-path の部分である。flag に #f が指定されていれば、.sr 以外を search する発見できないときは、#f を帰す。flag が#t のときは、現在 directory のみの.sr 以外をみる。

例 5.4. *user* >(find-source "lib")
"/usr/local/lib/sslib/lib.sr"

(**file-sepa-ext** file) *procedure*

file が拡張子をもつとき、分離した文字列の CONS を返す。

例 5.5. *user* >(file-sepa-ext "ver2.1/test.ss")
("ver2.1/test" . ".ss")
user >(file-sepa-ext "ver2.1/foo")

("ver2.1/foo" . "")

(**rename-file** file new) *procedure*

(**remove-file** file) *procedure*

(**copy-file** file1 file2) *procedure*

files2 は常に上書きされる。

(**mkdir** name) *procedure*

(**rmdir** name) *procedure*

(**file->string** file) *procedure*

file の内容を 1 個の文字列として返す。

(**make-temp-file**) *procedure*

これは実際に新しい適当なファイルを作る。そして、そのファイル名を返す。ファイルの作られる場所は *PWD*,(startdir),*HOME* の順にためされる。

5.6 ss のネットワーク機能

connection 型のネットワークをサポートする。データ型 netport は network 定義に用いる。socket は、socket-connect,socket-listen,socket-accept によって生成されるストリームである。network の入出力は通常の read,read-line, write,display,format などを使って行なうことができる。

HOSTNAME *variable*

環境変数 HOSTNAME の値が system によって代入されている。

(**netport** port [host]) *procedure*

netport データを作成する。port はポート番号 (整数) またはサービス名である。host はホスト名か IP アドレスの文字列である。省略されたときは *HOSTNAME* を使用する。これは単なる定義であり、まだ socket は作られていない。

例 5.6. *user* >(netport 50000)

#<netport 50000 192.168.100.108 >

user >(netport "ssh" "hoge")

#<netport 22 133.14.99.124 >

(**netport?** x) *procedure*

(**socket-connect** <netport>) *procedure*

netport に connect する。これはクライアントで用いられる。netport は、ある server によって待機状態でなければならい。成功すると、socket-port を返す。socket-port は双方向のストリームである。scheme の入出力関数を用いて読み書きできる。通常のストリームと同じく不用になったら (close-port <socket-port>) を使用しなければならない。

例 5.7. *user* >(define soc (socket-connect (netport "http" "www.dendai.ac.jp")))

soc

user >soc

#[network-port connect -1 -1(#<socket 4> . #<netport 80 133.20.16.60>)]

user >(format soc "GET ~%")

<#undef#>

user >(read-line soc)

"<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML 2.0//EN\">"

user >(close-port soc)

4

なお入出力はバッファー動作をするので、送信の場合は改行コードが送られるかバッファーがいっぱいにならないと実際の送信は行なわれない。従って、必要なら (flush <port>) を使う。また、受信の場合データが来るまで待ちになるが、これを避けるためには (char-ready? <port>) を使うのが良いだろう。

(**netport-used?** <netport>) *procedure*

netport が既に使用されているなら真を返す。

(**find-netport**) *procedure*

サーバーが利用可能な netport を返す。

(**socket-listen** <netport>) *procedure*

これはサーバープログラムで使う。netport にクライアントが接続できるようにする。ポート番号は、システムと競合しない大きな値が必要である。この関数の返す値もストリームだが通常の読み書きはできず、次の socket-accept のためにのみ用いる。char-ready?により接続があるか確認できる。終ったら close-port すべきである。ポート番号の 1023 以下で listen するには root 権限が必要である。private port の範囲は 49152 から 65535 である。前の netport-used?が真になる port に対する listen は error になる。

(**socket-accept** <socket-listen-port> [wait]) *procedure*

接続してきたクライアントとの通信路を確保する。返される値は双方向のストリームである。wait はミリ秒単位で省略値は 0. wait に 0 を指定すると接続があるまで待ち続ける。wait の値が正のとき、wait の間に (またはすでに) 接続がなければ#f が返る。複数の接続が可能になっている。

例 5.8. *user* >(define listen (socket-listen (netport 50000)))

listen

user >listen

#[network-port listen -1 -1(#<socket 4> . #<netport 50000 192.168.100.108$>)]

user >(define acc (socket-accept listen))

;;; waiting
;;; 他の terminal から socket-connect を実行する。
acc ;;;; 接続が起こった

user >(format acc "OK~%")

<#undef#>

user >(define acc2 (socket-accept listen))

acc2 ;;; 2 人目も接続された。

user >(netport-ip acc)

"192.168.100.199"

user >(netport-ip acc2)

"192.168.100.234"

(**netport-ip** <port>) *procedure*

<port> は netport または socket ストリームである。ip address の文字列を返す。

(**netport-number** <port>) *procedure*

port number を返す。

(**netport-hostname** <port>) *procedure*

hostname の文字列を返すが、もし見つからないときは#f を返す。<port> が不正なら error である。

(**socket-port?** x) *procedure*

x が socket-listen-port なら (), 双方向の socket ストリームなら#t, それ以外は#f を返す。

(**when-socket-ready** socket command) *procedure*

socket が ready になったとき実行する command を定義する。command は 0 引数の lambda 式または S 式である。command に#f を与えると、その socket の予約を取り消す。これは、実際には after を用いて、一定時間ごとに char-ready?と実行を繰りかえす。必要なら update を呼び出すこと。

5.7　電子会議

以下に、例として電子会議を示す。server program は、２人以上の接続を待ち、1 人からメッセージを受け取ると他の全員に送信する。参加者はいつ送信を行なってもよい。最初に

ss server

のようにして server を起動する。その後、client を起動する。参加者は quit を入力すれば会議から出る。

```
;;;  電子会議  client program
;;;  2004.7.16

(define (client)
  (let* ((msg 0)(name 0)(server 0) (net 0)(soc 0))
     (display "Server name? ")
     (flush)
     (set! server (read-line))
     (set! net (netport 50000 server))
     (set!  soc (socket-connect net))

     (display "Your name? ")
     (flush)
     (set! name (read-line))
     (format soc "~a~%" name)

      (while (not (equal? msg "quit"))
         (when  (char-ready? soc)
             (set! msg (read-line soc))
             (format #t "~a~%" msg)
         )
         (when  (char-ready?)
             (set! msg (read-line))
             (format soc "~a~%" msg )
         )
      )

    (close-port soc)
))

(client)


;;;  電子会議   multi-client server
;;;  2004.7.16
```

```
;;;;

(define net (netport 50000))
(define listen-soc (socket-listen net))

(define (server)
  (let* ((name 0)(acc 0)(n 0)(clients ()) (person 0)(msg 0))
     (format #t "waiting connection on ~s ~s ~%"
               (netport-number net) (netport-hostname net) )
     (set! acc (socket-accept listen-soc)) ;;; wait first client
     (inc n)
     (set! name (read-line acc))
     (format acc "Welcome ~a! you are the first member, Please Wait~%" name)
     (push (cons name acc) clients)
     (while (> n 0)
         (when  (char-ready? listen-soc)   ;;;;  New member coming?
             (set! acc (socket-accept listen-soc))
             (set! name (read-line acc))
             (send-msg  "Hello!" name clients) ;;; greeting
             (push (cons name acc) clients)
             (inc n)
             (format acc "Welcome you are ~s st member~%" n)
             (format acc "Now members are ~s~%" (map car clients))
         )
         (do ((all clients ))
             ((null? all))
             (set! person (pop all))
             (set! name (car person))
             (set! acc (cdr person))
             (when (char-ready? acc)      ;;; msg coming?
                 (set! msg (read-line acc))
                 (if (eof-object? msg)
                     (begin
                       (close-port acc)     ;;; this member is out!
                       (set! clients (remove person clients))
                       (dec n)
                     )
                     (begin
                      (send-msg msg name (remove person clients))
                      (format #t "recv: from ~a > ~a ~%" name msg)
                      ;;;(format #t "recv: ~s ~%" (map char->integer (string->list msg)))
                     )
                )
```

```
            )
         )
      )
    (close-port listen-soc) ;;;END!!!
))

(define (send-msg msg from-name others)
   (while (pair? others)
       (format (cdr (pop others)) "~a>> ~a~%" from-name  msg)
   )
)

(server)
```

この server プログラムは、when-socket-ready を用いて書くことにより、CPU 効率を向上してすっきりしたものになる。

```
;;;  電子会議   multi-client server
;;;  2004.7.16
;;;;  when-socket-ready  version

(define net (netport 50000))
(define listen-soc (socket-listen net))
(define *n* 0)
(define *clients* ())

(define (read-start name acc person)
  (when-socket-ready acc
    (lambda ()
       (let* ((msg 0))
           (set! msg (read-line acc))
              (if (eof-object? msg)
                (begin
                  (close-port acc)   ;;; this member is out!
                  (set! *clients* (remove person *clients*))
                  (dec *n*)
                  (when-socket-ready acc #f)  ;;;; cancel waiting
                 )
                (begin
                  (send-msg msg name (remove person *clients*)) ;;; send for others
                  (format #t "recv: from ~a > ~a ~%" name msg) ;;; display msg on server scr
                )
             )
       )
```

```
    )
  )
)

(define (send-msg msg name others)
   (while (pair? others)
       (format (cdr (pop others)) "~a>> ~a~%" name  msg)
   )
)
;;;; start accept !!
(when-socket-ready listen-soc
    (lambda ()
       (let* ((acc 0)(name 0))
         (set! acc (socket-accept listen-soc)) ;;; accept client
         (inc *n*)
         (set! name (read-line acc))
        (format acc "Welcome you are ~s st member~%" *n*)
         (if (= *n* 1)
            (format acc " Please Wait~%" name)  ;;; Only One
            (send-msg  "Hello!" name *clients*)  ;;; greeting for others
         )
         (push (cons name acc) *clients*)
         (format acc "Now members are ~s~%" (map car *clients*))
         (read-start name acc (cons name acc))
       )
    )
)
;;;;  you can stop server  by the following
(define (server-off)
    (when-socket-ready listen-soc #f)
    (map (lambda(x) (close-port (cdr x))) *clients*)
    (close-port listen-soc)
    (set! *clients* ())
)
```

第6章　SSとUNIX

ここでは、ssの起動やUnixとの連携に関連した機能を説明する。

6.1　ssの起動

ssを起動するときのコマンドの一般形は

ss file ... -p para1 ... -f file ...　　*unix command*

ssは、コマンドラインで指定されたfileをすべてloadする。-pが与えられると、そのあとのパラメータを変数

argv

に文字列のリストとして格納する。-fが与えられると、再びそのあとのfileをloadする。このあと端末は行編集のためにrawモードに設定され、対話モードが始まる。

パラメータは先頭から順に調べられることに注意してほしい。したがって、*argv*を利用するプログラムは

ss -p *.c -f program.ss

のように起動することになる。

その他のオプション

-s -sが与えられると、ssのsystem messageは出力されずエラーはstderrに送られる。従ってstdoutにはプログラムによる以外は出力されない。また端末に対する操作もされない。通常このoptionはパラメータで与えられるプログラムの実行のみをしたいときやcgiとして利用するときに用いられる。

-i init-file 最初に行なう初期化ファイルを指定する。この指定のないときはinitss.ssまたはinitsn.ssが用いられる。

-v 初期化のときのメッセージが詳しくなる。

-help, --help

(**silent-mode?**)　　*procedure*

-sで起動されたssの場合#tを返す。

6.2 [SS] 初期化ファイル, ∗auto-path∗

ss が起動したとき

initss.ss (または initsn.ss for sn)

をさがす。 search は一般に

./

の次に ∗auto-path∗(これは directory のリストである。) を見てその先頭から順に行なわれる。

∗**auto-path**∗ *variable*

ss のソースのサーチパスであり、初期値は
("/usr/local/lib/sslib/")
というリストである。必要ならこれを変更することが許されている。dir 名の最後に/が必要なことに注意してほしい。

∗**SSLIBDIR**∗ *variable*

sslib を install した directory の 文字列"/usr/local/lib/sslib/"が set されている。("/usr/lib/sslib/"かもしれない。)

変数 ∗SSLIBDIR∗ の値は ss の install に依存している。
initss.ss を load したあと

./start.sr (./start.ss)

があれば load されて、最後に起動コマンドのパラメータで渡された file が load されて start する
拡張子が省略されたときは, 拡張子なしの file を最初に見てから

.sr, .ss, .scm, .lsp,.stk

の順に search が行われる。.sr は compile された file である。/usr/local/lib/sslib/を変更するには、ss を make しなおす必要がある。
initss.ss を他の初期化ファイルに指定したいときは

ss -i hoge.ss

のように -i オプションを使う。

6.3 unix script

ss で開発したプログラムを unix のコマンドとして使用する方法について説明する。Unix の shell スクリプトを使うのが簡単である。
たとえば foo.ss が自分の HOME に完成しているとしよう。ファイル ssfoo を

```
#!/bin/bash
ss  -p $* -f ~/foo
```

の内容として作り

chmod +x ssfoo

で実行許可を与える。

./ssfoo p1 p2 ...

とすると ss が起動されて foo.ss を読みこむ。p1,p2,... は *argv* に文字列のリストとして渡されるので foo.ss の最後には

```
(when (pair? *argv*)
   (while (pair? *argv*)
      (foo-work (pop *argv*))
   )
 (bye)
 )
```

のように書いておくと、各パラメータに対して foo-work を実行したあとプログラムは終了する。すべてうまくいったら、

makesr foo

により foo.ss を compile しておく。さらに ssfoo をコマンド path の通った場所に移動すれば終了である。

もし GUI を使用しないのであれば、ss ではなく sn を使うのが良いだろう。また、すべての user が使用するのなら foo.sr を/usr/local/lib/sslib/ に copy しておく。そして ssfoo の 2 行目を

ss -p \$* -f foo

に変更する。

ss -s

-s オプションは、ss にメッセージを出力させない起動方法である。このときは、ss は初期化メッセージはもちろんプロンプトや評価した S 式の値も表示しない。標準出力に現われるのは明示的な出力命令によるものだけである。これは、ss または sn を cgi やスクリプト言語または filter として利用する事を想定している。もし、エラーが発生してもそれは stderr に出力される。line edit 機能は無効であり、通常の対話を行なうことはできない。-s オプションを用いた ss スクリプトは、ユーザーには scheme であることがわからないであろう。もし、あなたが自分自身のインタープリタ言語を開発したいのなら、最初に scheme 上で作ってみるのがおそらく早道である。実際、多くの prolog 言語は、Lisp 処理系の上にある。また、Emacs も lisp である。

6.4 net-run

C-program や unix コマンドの出力を見るだけなら、system,exec が使える。また、redirect を用いればさらに複雑な通信も可能である。ここでは、network port と redirect を用いたライブラリである net-run を紹介する。

(**net-run** cmd [para1 ...]) *procedure*

これは、cmd 以下の process を起動して、そのプロセスの pid とプロセスとの通信 (双方向) に用いる net-stream をドットペアにして返す。

(pid . stream)

この stream は、cmd の標準入出力にリダイレクトされているので、これを用いて process と交信できる。終了したら (kill pid) と (close-port stream) をする。なお、net-run は自動的に after を用いてプロセスの終了を監視しており、もし終了したらプロセス側の connection は close するようにしてある。

例 6.1. cat は標準入力をそのまま標準出力に送ることに注意して、これを net-run してみます。

user >(define ps (net-run "cat"))
port number 52000
ps
user >ps
((4960 . \#f) .
#[network-port connect -1 -1(#<socket 6> . #<netport 52000 192.168.100.101>)])
user >(define comm (cdr ps))
comm
user >(format comm "test of comm~%")
<#undef#>
user >(read-line comm)
"test of comm"
user >(close-port comm)
6
user >(kill (car ps))
#t

6.5 SRFI-22

SRFI-22 は scheme script の 1 つの仕様を定義している。これは、ss では /usr/local/bin/scheme-r5rs によって供給される。scheme-r5rs は、通常 make install のときに install されている。

例として、unix の cat コマンドを SRFI-22 のスクリプトで書いてみる。ss-cat.ss を以下の内容として作る。

```
(define (main argv)
  (for-each display-file (cdr argv))
  0
)

(define (display-file file)
  (call-with-input-file file
```

```
          (lambda(port)
             (let loop ()
                (let ((thing (read-char port)))
                   (if (not (eof-object? thing))
                       (begin
                          (write-char thing)
                           (loop)
                       )
                    )
                 )
              )
           )
        )
    )
```

ここで、main 関数が実行される関数であり、引数の argv は先頭が ss で残りが実行時に渡されるパラメータからなるリストである。なお、main 関数の最後が 0 になっているのは、正常終了を意味する。 これを compile する。

makesr ss-cat.ss

ここでできた ss-cat.sr を

```
mv ss-cat.sr ss-cat
chmod +x ss-cat
```

のように ss-cat に変更して実行属性をつける。さらに、ss-cat を vi のような editor で開き

```
#!/usr/bin/env scheme-r5rs
```

を先頭行として挿入する。

./ss-cat file1 ...

とすれば、file1 ... の内容を表示できる。この場合、GUI を使わないので ss より sn を使うのが適切である。そのためには、先頭行を

```
#!/usr/bin/env  sn-r5rs
```

に変えればよい。

6.6 Module

(**provide** module) *procedure*

module が load 済であることを宣言 (登録) する。

(**require** module [dir]) *procedure*

もし module が登録済でなければ、module を load する。成功すれば provide されたことになる。dir は最初に見る directory の指定である。既定値は"./" である。dir は文字列で最後に/が必要である。

(**remove-module** module) *procedure*

単に、登録を取り消す。module が消えるわけではない。

(**module-dir** module) *procedure*

module が見つかった directory を返す。ただし、module が provide 宣言されている場合は実際の値ではない。登録されていなければ#f が返される。

module 引数は文字列またはシンボルである。それは、拡張子を含んではいけない。 また、これらは toplevel に置くのが望ましい。特に、module がマクロや構造体の定義を含むときは、compiler はそれを知らなければいけない。require はソースファイルの先頭部分に置くべきである。

たとえば

(require "macro")

は、次のようなことをひきおこす。さがされるのは、macro.sr または macro.ss である。最初に現在 directory からさがす。見つかれば、それを load して登録する。 macro.sr,macro.ss がない場合 directory の macro∗ または MACRO∗ の中が捜される。このような directory が複数あるときは (glob-dir "macro∗" "MACRO∗")(つまり string<?) で最後の directory が対象になる。

これに失敗したときは、∗auto-path∗ で同じことを行なう。

compiler は、toplevel の require,provide,remove-module を実行する。require の場合それが含む定義がすべて load される。

require は、その探索において chdir をする。実行が終ればもとに戻る。従って、module は GUI などの実行をその中で伴うものは好ましくない。基本的に、module は変数、関数、マクロ、構造などの定義文のみからなるものである。実行は require の終了後に行なうのがよい。

require は、directory に access するのが特徴である。ただし.ss,.sr 以外は無視する。これに 対し、load は ∗autopath∗ の sub directory を捜すことはしない。

第7章　入出力

7.1　データの表記法 (read 関数)

シンボル

特殊な文字を含むときは, \ (single-escape) または | (multi-escape) を用います。

例. |Yes,I have.| , \(

数の表記

整数 (任意長)

100

100. （この形は常に 10 進整数を表す。)

#b1011 (2 進)

#o3257 (8 進)

#xffd0 (16 進)

∗read-base∗　　*variable*

2 以上 32 以下 (default 10) n 進数入力に固定ができます。

(set! ∗read-base∗ 12.)

とすると, 10 は 12(10 進), 10. は 10(10 進), ab は symbol ではなく整数 (131.) と解釈されます。従って, この機能は data file の read にのみ使うのが安全です。

(∗print-base∗ で n 進数出力ができます。format 関数も参照)

有理数 (任意長)

整数/整数

-1/2 ,22/7

など。約分は自動的に行なわれる。#b なども使える。

#b101/11 (=5/3)

#b などは、一時的に ∗read-base∗ を変える働きをする。

実数

1.0 ,3.14 , 6.02e23 (または 6.02d23)

指数部に使える文字は e,E,d,D 実数は ∗read-base∗ の値にかかわらず 10 進数で解釈される。

複素数

1+i,1-i,1/2+i,3.5+1.2i,4+i6

など。純虚数 i,-i は

+i, 1i,+1i,+i1,0+i
-i,-1i,-i1,0-i
のように書ける。i,i1,++i などは複素数ではなくシンボルとして解釈されることに注意して欲しい。#b なども前と同様である。
#b101+11i = 5+3i

日本語

(**kanji-mode** #t) *procedure*

read,format 関数 は漢字に対応します。(デフォルト)

大文字化

Mac-Lisp, Common Lisp では通常 read は入力された文字を大文字に変換します。また、多くの scheme は逆に小文字に変換する。しかし、本システムでは小文字と大文字を区別するのが通常となっている。

∗**auto-case**∗ *variable*

この値が#f ならば変換はしません。(デフォルト). もし#t なら, escape されていない限り英字は小文字に変換されます。#f,#t 以外の値のときは大文字に変換されます。ただし, 本 system では, これは#f がデフォルトで, 大文字小文字は区別されます。 しかも, system 関数は総て小文字なので, ファイルの変換などのときだけに使うのが安全でしょう。

文字

#\a,#\b,#\ あ,,... などは文字 a,b, あ... を表します。特殊 code は,
#\newline,#\tab,....
のように名前で記述できます。 これは, (integer->char n) で見て下さい。 名前は, すべて小文字です。

文字列

”echo test.” のように書きます。\ は” 自身の escape に用いる。また、\ 文字を入れたいときは、\\ としなければいけない。
例. ”this is \”test\” char #\\a”

vector

#(1 2 3) のように書きます。

配列

#a((1 2 3)(4 5 6)) のように書きます。

7.2 [SS] read macro,read table

この system では, CL の仕様をもとにした read macro 機能を持っています。従って, 任意の文字を macro 文字とすることができます。標準的には

```
括弧  (  )
クオート  '
逆引用符記法  `,@
#  非終端 dispatch マクロ
    #\name    char
    #(s1 s2 ...)     vector
    #a(.....)       array
    #b,#o,#x
 注釈     ;
 #| .... |#      注釈  (nest して良い)
 #! ...          改行まで註釈とみなされる。shell script の 1 行目を
                 skip するためにある。
 escape   \
 multi-escape   |
```

が利用可能です。

read 関数の動作は、ベクトル read-table で支配されており

sys:∗read-table∗ *variable*

read-tabel の初期値を保持する。user は変更してはならない。

∗read-table∗ *variable*

現在使っている、read-tabel の値を保持する。

によって参照できる。read-table を操作するための関数は sslib/read_table.ss に定義されている。read-table を操作することにより、lisp 以外の言語を read することも可能になる。

```
(load "read_table")
(in-package "read_table")
```

とすると以下の関数が使えるようになります。

(**copy-read-table** table) *procedure*

複製を作る。

(**get-syntax** char table) *procedure*

char(文字) の属性値と macro 関数の cons pair を返す。

(**set-syntax** char type function table) *procedure*

例. (get-syntax #\( ) ==>(16 . #[closure system 1 0 8061df0 ()])

数 16 は、左括弧 (が終端 macro を意味しています。
macro 関数は

(rm-quote path)

のような 1 引数の関数でなければいけません。read 関数は, macro 文字を read すると入力 stream を引き数として, マクロ関数をよびだして, その帰り値を結果とします。マクロ関数は, read 自身を再帰的に呼び出してもかまいません。 詳しい動作は Common Lisp 仕様書, 例は read_table.ss にあります。 以下で概略を説明します。属性値は

```
constituent                (通常文字)
w_space                     white space     #\space #\newline など
m_escape                    multiple escape     |
s_escape                    single -escape      \
t_macro                     終端macro  (terminating macro)    ()'` ", ;
non_t_macro                 非終端 macro (non-terminating macro)
                            標準の table にはない。
dispatch_macro              dispatch macro
                            標準の table にはない。
non_t_dispatch_macro        非終端 dispatch macro
                            (non-terminating dispatch macro)      #
illegal_char                標準の table にはない。
```

の 8 種類です。左の名前は、read_table.ss で使われているシンボルで、ある整数値（システム内部での表現）をもっています。たとえば

(set-syntax #\[illegal_char #f ∗read-table∗)

を実行して

'[

と入力すると、error が発生します。illegal_char を読むと read は単に error を起こします。s_escape,m_escape は 1 個または複数の文字を constituent として解釈するようにします。constituent の連続は 1 つの token を構成します。token の終わりは w_space または終端 (dispatch)macro の出現です。token は通常シンボルまたは数として解釈されます。他の、データ型は macro 呼び出しによって構成されます。さて、

```
(define (foo path)
    (list 'foo (read path))  )
(set-syntax  #\&   t_macro hoo  *read-table*)
```

としてみます。

'&x -> (foo x)
'(x&y) ->(x (hoo y))

となります。では、

(set! ∗read-table∗ (copy-read-table sys:∗read-table∗))

で、もとに戻して

(set-syntax #\& non_t_macro hoo *read-table*)

としてみます。

'&x -> (foo x)
'(x&y) -> (x&y)

つまり、非終端 macro が token(単語) の途中で現れたときは、read は macro とはみなしません。これに対し、終端 macro の場合は、escape されるか他の先行する macro で処理されない限り必ずその macro 関数が呼び出されて、その返す値が read 関数の得る値となります。もし、macro 関数が値 #<undef> (この値を read が返すことはありえません。) を返すなら、read は、単に空白を読み込んだのと同じ動作（従って、read は続行される）をします。dispatch macro は その直後に続く 1 文字（補助文字）によって動作が変化します。get-syntax の返す値は属性値 (dispatch-macro または non_t_dispatch_macro) と vector のドット対です。この、vector に関数を入れます。関数は

(rm-func path ch)

のような 2 引数の関数で、入力 stream と補助文字を引数として呼び出されます。標準の #は、非終端 dispatch macro です。

```
(make-dispatch #\& dispatch_macro *read-table*)
(define (and-e path c)
    (list 'expt (read path)(read path)) )
(set-dispatch #\& #\e and-e *read-table*)
```

は文字&を dipatch macro として、&e を定義してます。

'&ex 6 ->(expt x 6)
'&e x 6 ->(expt x 6)

のようになります。

7.3 [SS] tty

ss のキーボードからの入力
次の 2 つの機能が使えます。(1)**history 機能**

入力した式は vector
line-history
に蓄えられます。(default は 25 個）line edit 機能が使えない場合は
(his)
とすると、番号を選んで過去の入力を再実行できます。
(his n)
ただし 1<= n <25 でもよい。

(2)**line edit 機能**

今のところ rxvt, kterm, xterm 端末に対応しています。rxvt,kterm の場合について説明します。

PageUp, PageDown　（history 機能）
history の内容を読み込んでそれを現在の編集内容にします。

カーソルキー
移動します。
BS, delete
1 文字の消去。
CTRL-P
編集してる式の先頭または末尾にカーソルを移動
CTRL-O
S 式の check をします。
CTRL-D
カーソル位置で行を分割します。カーソルが行末なら新しい行を挿入。

CTRL-E
カーソル位置の行を削除します。カーソルが先頭行ならその
行の内容を消去します。

Return
編集内容が read 関数に渡されます。S 式が完全なら、実行されますが
不完全なら、さらに line editor が呼び出されます。
このとき、すでに送った内容については編集不可能です。
取り消したいときは、
CTRL-C
非常手段
CTRL-Q
Return と同じ。
CTRL-T
再表示を行います。

original の xterm の場合、history 機能は CTRL-U,CTRL-Y で使えます。Redhat-9 の xterm の場合は上と同じ。X-window における、通常の日本語入力、cut and paste が使えます。ss を通常に起動すると、このモードになりますがやめたいときは

(tty #f)

を入力します。復帰するときは

(tty) または (tty #t)

とします。

(transcript-on file)

を使うときは、(tty #f) でないといけません。

注意

対応してない端末を使用するときは、たとえば vt100.ss のようなソースが必要です。rxvt.ss, xterm.ss を参考にしてください。そこで、必要なのはキーボード入力と機能（カーソル移動など）の対応付け、および画面における

カーソル移動
色をつける（あるいは何か、通常文字と区別する）

のための sequence です。カーソル移動さえできれば、read check 以外は使えます。

現在、この機能はすべて scheme で書かれています sslib/KEY/

key.ss keycontrol.ss readchk.ss rxvt.ss

を見てください。

画面サイズの変更は、次の入力から有効になります。端末の行数を超える、行数の編集には対応してません。

7.4 [SS] format

Common Lisp 仕様の一部をサポートしている。
注。 現在：指示は指定できない。大文字小文字変換などもサポートしていない。
簡単な場合は、

~a,~s, ~%

だけでも十分である。

(**format** stream control-string arg1 arg2 .....) *procedure*

stream は

#t 標準出力
#f 文字列として出力する

でも良い。

以下の詳細は、 CL の仕様を参照。

~a

```
(format #t "this ~10,,,'*a ~%" 'is)
       this is********
```

10 は幅
'*は padding 文字、 これは is が左詰めで出力される。

~width,colinc,minpad,padcharA
出力が width にみたないとき、minpad 個の padchar が付加される。
さらに必要なら、colinc の整数倍の padchar が出力される。
既定値は、width=0,colinc=1,minpad=0,padchar=' （空白）
である。

```
(format #t "this ~10,,,'*@a ~%" 'is)
        this ********is
```

@は右詰めを指示する。

~s

```
(format #t "this ~10,,,'*@s ~%" "is")
       this ******"is"
```

~a と同様。read 可能な S 式として出力する。

~d

整数と有理数の１０進表示

```
(format #t "this ~15,'*@d ~%" 1234)
      this **********+1234
```

~d では、常に右詰め。@は、符号 +　の印字を指示する。

~b

```
(format #t "this ~15,'*@b ~%" 1234)
       this ***+10011010010
```

~b は２進数表示　　パラメータは~d と同じ。

~o

```
(format #t "this ~15,'*@o ~%" 1234)
       this **********+2322
```

~o　８進数

~x

```
(format #t "this ~15,'0x ~%" 1234)
      this 0000000000004d2
```

~x　　１６進数

~r

```
(format #t "this ~16r ~%" 1234)
      this 4d2
```

~nR は、n 進数

~p

```
(format #t "this ~p ~%" 1234)
     this s
```

~p は、引数が１でなければ s を印字する。

~c

```
(format #t "this ~c ~%" #\a)
     this a
(format #t "this ~@c ~%" #\a)
     this #\a
```

~c は文字の~a, ~@c は~s で印字する。

~f

```
(define (foo x)
  (format #t "~f:~6,2f:~6,2,1,'*f:~6,2,,'?f:~6f:~,2f ~%" x x x x x x)
)
(foo 3.14159)  ->   3.14159:  3.14: 31.42:  3.14:3.1416:3.14
(foo -3.14159) ->   -3.14159: -3.14:-31.42: -3.14:-3.142:-3.14
(foo 100.0)    ->   100.0:100.00:******:100.00: 100.0:100.00
(foo 1234.0)   ->   1234.0:1234.00:******:??????:1234.0:1234.00
(foo 0.006)    ->   0.006:  0.01:  0.06:  0.01: 0.006:0.01
```

~f は小数点表示をする。

~width,d,k,over_char,pad_char f

d は小数点の後の桁数

k はスケールファクタ

over_char が指定されていれば、印字が width を超えるときそれが印字される。

~e

```
(define (foo x)
    (format #t "~e:~9,2,1,,'*e:~10,3,2,2,'?,,'$e:~9,3,2,-2,'%@e:~9,2e~%" x x x x  x)
)
(foo 3.14159)    ->  3.14159e+0:  3.14e+0: 31.42$-01:+.003e+03:  3.14e+0
(foo -3.14159)   ->  -3.14159e+0: -3.14e+0:-31.42$-01:-.003e+03: -3.14e+0
(foo 1100.0)     ->  1.1e+3:  1.10e+3: 11.00$+02:+.001e+06:  1.10e+3
(foo 1.1e13)     ->  1.1e+13:*********: 11.00$+12:+.001e+16: 1.10e+13
(foo 1.1e120)    ->  1.1e+120:*********:??????????:%%%%%%%%:1.10e+120
```

~e は指数表示である。

~width,d,e,k,over_char,pad_char,exp_char e

e は指数部の桁数

~g

```
(define (foo x)
   (format #t "~g:~9,2,1,,'*g:~9,3,2,3,'?,,'$g:~9,3,2,0,'%@g:~9,2g~%" x x x x x)
 )
(foo 0.0314159) ->  0.0314159    : 0.031   :314.2$-04:+.314e-01:  3.14e-2
```

```
(foo 0.314159)  ->  0.314159    :  0.31   :314.2$-03:+.314e+00: 0.31
(foo 3.14159)   ->  3.14159    :   3.1   : 3.14    :+3.14    :  3.1
(foo 31.4159)   ->  31.4159    :   31.   : 31.4    :+31.4    :  31.
(foo 314.159)   ->  314.159    :  3.14e+2: 314.    :+314.    :  3.14e+2
(foo 3141.59)   ->  3141.59    :  3.14e+3:314.2$+01:+.314e+04:  3.14e+3
(foo 3.14e12)   ->  3.14e+12:*********:314.0$+10:+.314e+13: 3.14e+12
(foo 1.1e120)   ->  1.1e+120:*********:??????????:%%%%%%%%:1.10e+120
```

~g は、~f,~e を選択して印字する。
~w,d,e,k,over_char,pad_char,exp_char g

~% 改行

```
  (format #t "~s ~3% test~&" 'sato)
```

~n% はｎ個出力する。

~& 行の先頭以外では改行する。

```
 (format #t "~s ~&~&test~&" 'sato)
```

~n&は１個の&と n-1 個の%と同じ。

```
(format #t "~s ~3& test~&" 'sato)
```

~| 改ページ

~~ ~自身

~改行 ~の直後の改行は無視され、さらにその後の空白も無視される。
これは、長い出力制御文字列を書くときに使用する。

~t

```
    (format #t "~t~a~t~a~t~a~%" 'a 'b 'c)
 a b c
    (format #t "~&~a~8t~a~16t~a~%" 'a 'b 'c)

a       b       c
    (format #t "~&~a~0,8t~a~0,8t~a~%" 'a 'b 'c)
```

tabulate
~pos,inc t
これは、カーソルが pos の位置になるように、空白を出力する。
もし、すでにカーソルが pos 以上の位置にあるとき、inc が 0 でなければ、
カーソルが
pos+inc*最小の正整数　の位置になるようにする。pos,inc の省略値は 1 である。

~?

```
(format #t "~?   ~s"    "~s ~s" '(a b) 'c)
     a b   c
(format #t "~@?  ~s "   "~s ~s" 'a 'b 'c)
     a b  c
```

~?に対応する引数は、制御文字列である。そして、次の引数はリストであり、この制御文字列によって消費される。
~@?は，その位置に制御文字列が挿入される。

7.5 [R5RS] stream

(**input-port?** obj) *procedure*

(**output-port?** obj) *procedure*

(**current-input-port**) *procedure*

(**current-output-port**) *procedure*

(**close-input-port** port) *procedure*

(**close-output-port** port) *procedure*

(**open-input-file** filename) *procedure*

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

[R5RS,+] すでに filename が存在する場合それは上書きされます。

(**call-with-input-file** file proc) *procedure*

(**call-with-output-file** file proc) *procedure*

(**char-ready?** [port]) *procedure*

port が入力可能なら#t を返す。port が socket の場合は可能な入力バイト数または#t を返す。入力がなければ#f を返す。

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

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

漢字モード (EUC) のときは euc コードは 2byte を 1 個の文字として read する。

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

This is almost same as (unread-char(read-char port) port)

(**eof-object?** obj) *procedure*

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

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

(**newline** [port]) *procedure*

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

7.6 [SS]stream

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

改行が現われるまで読みこんだ文字列を返す。改行コードは文字列に含まれない。SJIS の改行 0d0a, MAC の改行 0a は 1 個の改行コードとして処理する。

(**read-byte-char** [port]) *procedure*

漢字モードに関係なく 1byte ずつ read する。

(**read-number** [path]) *procedure*

結果は常に実数または eof-obj である高速性が要求される場合に１０進実数の read のみに使う空白と tab, 改行は無視する。

[SS] port stream ss で扱えるストリームは、標準入出力、ファイル、socket, 文字列、関数型である。すべてのストリームに対して、read,write,format,load などの関数を使用できる。

(**close-port** port) *procedure*

(**closed-port?** obj) *procedure*

(**socket-port?** obj) *procedure*

(**current-error-output-port**) *procedure*

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

(**file-exists?** filename) *procedure*

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

追加書き込みで open される。

(**open-input-str** str) *procedure*

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

(**open-append-str** str) *procedure*

(**open-io-str** str) *procedure*

(**open-ia-str** str) *procedure*

これらは、文字列に対するポートを作る。書き込みの場合もとの文字列の大きさは自動的に拡大される。結果の文字列は、これを close したときの返り値である。open-io-str では str は input,output モードで open される。open-ia-str では str は input,append モードで open される。

(**reopen-str** str-port [:input|:ouput|:append|:io|:ia]) *procedure*

str-port はすでに close された文字列ポートで、それが作られたときと同じモードで open される。mode を指定してもよい。

(**open-input-func** funcion [p1 ...]) *procedure*

関数型の入力ポートを作る。関数は 2 個の引数を受けつけ、以下のように動作しなければならない。弟 2 引数で与えられるのは通常 eof-object である弟 1 引数の値が:open のときは初期化を行なう。これは、open 処理のときの呼びだしである。初期化に失敗したときは関数内部で error を起こすか、#f を返す。もし、unread-char をこの関数で独自に監理するときは:open に対する帰り値はキーワード :unread でなければならない。:open の場合の弟 2 引数は、p1 ... のリストである。弟 1 引数の値が:close のときは close 処理を行なう。弟 1 引数の値が:ready のときは 文字が入力可能か end-of-file なら#t そうでなければ#f を返す。。弟 1 引数の値が:input のときは 1 個の文字または eof-object を返す. 弟 1 引数の値が :unread のとき第 2 引数は文字であり、関数はその文字を値として返す。弟 1 引数の値が :reopen のとき弟 2 引数は reopen-func に与えられたすべての引数のリストである。 :reopen をサポートする場合、帰り値は真である。

```
[read-pos  (optional)]
    :readpos    see read-pos.

[set-read-pos (optional)]
    :set_readpos    see set-read-pos.

 [shift-read-pos (optional)]
    :shift_readpos    see shift-read-pos.
```

弟 1 引数がこれらでないときは#f を返す。これは将来の拡張のためである。(このときの弟 2 引数は eof-object でないかもしれない。)

unread-char の処理は関数が処理しないのならシステムが行なう。日本語処理は system が行なう。(関数において行なっても良い)

(**open-output-func** funcion [p1 ...]) *procedure*

関数型の出力ポートを作る関数は 2 個の引数を受けつけ、以下のように動作しなければならない。弟 2 引数で与えられるのは通常出力するべき文字である。弟 1 引数の:open,:close の意味は前と同様である。close-port 関数は、この関数が close 処理で返す値を返す。弟 1 引数の値が:flush のときは、関数は flush 動作を行なう。flush が返す値も、このときの関数が返す値である。弟 1 引数の値が :output のときは弟 2 引数で与えられた文字を出力する。

```
:unwrite    The second argument is eof-object. See unwrite
:getpos     See get-line-pos
:setpos     See set-line-pos
```

:unread,:unwrite,:getpos,:setpos,:readpos ,:set_readpos, :shift_readpos のいくつかを サポートするときは、 :open に対してそれらのキーワードのリストを返さなければならない。

(**open-io-func** funcion [p1 ...]) *procedure*

入力出力の両方可能な関数型のポートを作る。

(**reopen-func** func-port [p1 ...]) *procedure*

(**call-with-input-str** str proc) *procedure*

(**call-with-output-str** str proc) *procedure*

(**with-input-from-file** file thunk) *procedure*

[R5RS,+]

(**with-output-to-file** file thunk) *procedure*

[R5RS,+]

(**end-of-file?** port) *procedure*

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

入力 port に 1 文字戻す。これを連続して行なうことは通常 error である。 関数 port を見よ。

(**unwrite-char** port) *procedure*

port は str または関数型である。書き込み位置を 1 つ前に戻す。返される値は取り消された文字である。もし、位置がすでに先頭ならば、eof-object が返される。

(**inkey** [echo-flag] [wait-flag]) *procedure*

この関数を使用するためには scheme が起動したときの標準入力は端末でなければいけない。これは端末から全く buffering なしで 1 文字入力する。echo-flag,wait-flag の default は#t である。(read-char) と (inkey) は似ているがかなり動作は異なる。(read-char) は buffer 動作を行なうので。1 文字の入力だけでは終了しない。

(**copy-port** in out) *procedure*

入力ポート in の内容をすべて out に書き出す。

(**file->string** file) *procedure*

(**port->string** port) *procedure*

(**do-read** proc) *procedure*

proc は 1 引数の関数 object である。標準入力から read した object に対し proc を適用する。これを入力が eof になるまで行なう。

(**do-read-line** proc) *procedure*

(**port-filename** port) *procedure*

port が file に対するものなら、その file 名を返す。それ以外は#f.

(**port-string** port) *procedure*

(**port-function** port) *procedure*

(**fileno** port) *procedure*

port が Unix の fileno(整数値) を持つならそれを返す。そうでなければ#f を返す。

(**stdio-push!** integer port) *procedure*

integer は 0,1,2 のどれかである。integer=0 ならば (current-input-port) の値が port になるように変更される。これは stack として動作する。redirect が、fileno の操作を伴うのに対しこれは単に scheme の内部における標準 port を変更する。

(**stdio-pull!** integer) *procedure*

もとにもどす。close-port は別にする必要がある。

(**read-pos** port) *procedure*

(**set-read-pos** num port) *procedure*

(**shift-read-pos** num port) *procedure*

port must be a string or function port. These handle read position. Do not use these functions with unread-char or peek-char. It may cause confusion.

(**get-line-pos** [output-port]) *procedure*

行出力の位置 (行頭を 0 とする。) を整数値で返す。この関数は制御コードなどは認識しない。最後に改行が出力されてから何文字が出力されたかを示す。

(**set-line-pos** pos [output-port]) *procedure*

(**load** filename-or-port) *procedure*

[R5RS,+]

(**load-verbose** [read-echo value-echo search-echo]) *procedure*

load 関数の情報表示を制御する。3 個の引数の default は真である。read-echo は、ファイルから read した行を表示する。value-echo は、eval の結果を表示。search-echo は、load 関数が search する path 名を表示する。compile された ∗.sr ファイルに対しては read-echo,value-echo は無効である。引数のないときは、それらの値のリストを返す。

∗**stdnull**∗ *variable*

出力ポートだが、このポートに対するすべての出力は単に捨てられる。

7.7 [SS] 日本語処理

ss の日本語処理は基本的に euc code で行なわれます。文字列処理、入出力は euc に対応してます。TCL/TK は utf8 を使っていますが、その変換は ss が自動的に行ないます。他のコード系に対応するため、以下のシステム関数が定義されています。

(**euc->sjis** str) *procedure*

(**sjis->euc** str) *procedure*

(**jis->euc** str) *procedure*

(**euc->jis** str) *procedure*

(**euc->utf8** str) *procedure*

(**utf8->euc** str) *procedure*

これらの関数は変換に成功すれば新しい文字列を返しますが、失敗したときは#f を返します。jis は ISO-2022-JP であり。旧漢字 (78-JIS) や弟 3、4 水準の漢字をサポートしません。euc,utf8 は半角カタカナを使用できますが半角カナや機種依存文字は、使わないのが無難です。

[SS library]

(**sjis->jis** str) *procedure*

(**sjis->utf8** str) *procedure*

(**jis->sjis** str) *procedure*

(**jis->utf8** str) *procedure*

(**utf8->jis** str) *procedure*

(**utf8->sjis** str) *procedure*

(**kanji-code?** str) *procedure*

str のコード系を判定します。結果はキーワードで:ascii, :jis, :utf8, :euc, :sjis となります。binary に対しては#f が返されます。判定は上の順に行なわれるので、文字列が短いときなどは正しい結果とならない場合があります。特に、:euc または:utf8 と判定された場合:sjis である可能性があります。SJIS は最も多種のコードを使用するからです。

[SS library]

(**kconv** str code) *procedure*

str を code で指定した型に変換します。code は:jis, :euc.:utf8,:sjis のどれか。この関数は必ず文字列を返す。

(**kconv-func** from-code to-code) *procedure*

変換関数を返す。

(**file-code?** file) *procedure*

file のコード判定

(**kconv-file** file new code [from-code]) *procedure*

file を code に変換して new を作る。file と new は同じでもよい。返り値は new だが変換に失敗したときは、推定した元ファイルのコードと失敗した行数の cons を返す。file のコードがわかっているときは、それを from-code で与えることができる。new がすでに存在しても上書きされる。

(**jiscode->unicode** integer) *procedure*

(**unicode->jiscode** integer) *procedure*

(**jiscode->sjis** integer) *procedure*

(**sjis->jiscode** integer) *procedure*

(**utf8code->unicode** integer) *procedure*

(**unicode->utf8code** integer) *procedure*

(**euc->utf8code** integer) *procedure*

(**utf8code->euc** integer) *procedure*

これらは、整数値としてコード変換を行ないます。jis,unicode 変換では文字が存在しなければ 0 が返されます。sjis,jis 変換では文字の存在は check されません。

unicode は 16bit のみ、従って utf8 は 1byte ないし 3byte の表現のみをサポートします。なお euc は jis に#x8080 を加えたものであり、jis は区点コードに#x2020 を加えたものです。(半角カナを除く)

第8章　TK,Open/GL

8.1　[SS] TCL TK

ss は TCL/TK を内部に取り込んでいる。TCL/TK の使いかたは、stk を模範としているので stk で開発したプログラムはほとんど修正無しで動く。詳しい解説は「SS 入門」を見てほしい。また TCL/TK の入門書も参考にしてほしい。

tcl/tk のコマンドは自動的に ss の関数に登録されている。ただし、ss のシステム関数と同じ名前のものは”tk”パッケージに登録される。

(list-tk-procedure)

は tk-procedure のリストを返す。たとえば、ss は after 関数を持っているので TCL/TK の after に直接アクセスするには tk:after と書かないといけない。TCL/TK のコマンドのうち TCL の部分はほとんど必要がない。scheme の変数、関数、制御構造などが使えるからである。主に必要なのは TK 部分で widget を作成操作するコマンドである。たとえば TCL/TK における

```
button .hello -text Hello -command {puts stdout ”Hello, World!”}
pack .hello
```

の例は ss では

```
(button '.hello :text ”Hello” :command '(format #t ”Hello, World!~%”))
(pack .hello)
```

となる。TCL/TK の文法では scheme と違い、パラメータの評価 (実際は置換) は$で明示する。また {...} はリストを作る。TK の button は button widget を作る命令で.hello は button につける名前である。widget 名は必ず |.| (ルートウインドウ) から始まる。-text, -command のように-から始まるのはオプションの指定である。これらは ss ではキーワード :text,:command にする。このとき、TK の解説ではオプションに大文字を使っていることがあるが オプションはすべて小文字で記述しないといけない。これは特に、:textvariable のようなオプションに対して重要である。上の例で ss の button 文では、.hello はシンボルなのでクオートが必要である。これが実行されると、シンボル.hello には自動的にその widget を操作する tk-procedure がグローバルな値として set される。このため その後の pack の文では.hello はクオートされていない。

```
user> .hello
  #[TK 0 1 #[subraddress  804f918]((:tk-obj-name . .hello)
     (:command format #t "Hello, World!~%") (:text . "Hello")
     (:T . .hello) (:tk-obj-name . button) (:command format #t "Hello, World!~%")
     (:height . 0) (:width . 0) (:text . "Hello") (:tearoffcommand)
     (:postcommand) (:bind) (:vcmd) (:validatecommand) (:invcmd)
```

```
 (:invalidcommand) (:yscrollcommand) (:xscrollcommand) (:command)
(:T . button) (:T . tk-make-cmd))]
```

のように実際に値を持つことが確認できる。

user> (.hello 'configure :text "test")

とすると button の表示名が test に変わる。TCL/TK の教科書では

.hello configure -text test

と書いているであろう。 このようにして、TCL/TK の操作をすべて ss 上で行なうことができる。

widget 名や TCL/TK 変数などは常にグローバルであることに注意して欲しい。(もし ss を複数起動していてもそれらの間で widget 名の衝突はない) 1 つの ss で複数の同じ種類の widget を生成するとき名前の衝突をさけるには

(winfo 'exists name)

を使って存在を確かめるか、ライブラリ関数

(gen-toplevel name)

を使って、toplevel window を作り、すべての widget はその下に作るようにすると良い。

(**gen-toplevel** name) *procedure*

name の toplevel widget を作る。もし name がすでに存在していたら name のあとに数を付けた名前を試す。toplevel を呼び、作成した toplevel の名前を返す。

例 8.1. *user* >(define topw (gen-toplevel '.test))

topw

user >topw

.test

user >(define bt1 (tk-name topw '.bt1))

bt1

user >(button bt1)

.test.bt1

user >(pack bt1)

()

user >(funcall bt1 'configure :text "Hello")

()

最後の funcall に注意して欲しい。これを

(bt1 'configure :text "Hello")

と書くことはできない。bt1 を評価した値はシンボル.test.bt1 であってこれは関数 object でないからである。もう 1 度評価して実行できることになる。funcall は、第 1 引数を関数 object が得られるまで評価して残りをパラメータとして実行する。

(**tk:update**) *procedure*

[tk] ss の update を使う方が良いだろう。(update) は ss の update と (tk:update) の両方を行なう。しかし、画面の表示のみ更新したいときは良い。

(**tcl** command-string) *procedure*

[ss] tcl コマンドを直接実行する。debug などで必要なときのみ使用するのが良い。

(**tklink** var value) *procedure*

[ss] 変数 var を TCL/TK と共有する。var はグローバル変数になり、TCL/TK からも ss からもアクセス可能になる。変数の共有は、widget オプションによるものも存在する。

例 8.2. . *user* >(tklink 'sss #t)

<#TKLINK 1#>

user >sss

1

user >(tcl "puts $sss")

1 ()

user >(set! sss '(a b))

<#undef#>

user >sss

" a b"

user >(tcl "puts $sss")

a b ()

TCL/TK においては#t,#f は 1,0 であり、すべてのデータは本質的に文字列のためこのようになる。TCL/TK のリストは単に空白で区切られた文字列である。従って、1 個の要素だけからなるリストと 1 個の文字列 (または数) は、TCL/TK では区別されない。widget から get オプションなどでリストを取得するとき、この問題が発生するかもしれない。標準的なものに関しては ss がリストに変換して対応しているが、すべてではない。このような場合は ss のプログラムにおいて、リストであるかを check する必要がある。

(**tkunlink** var) *procedure*

[ss] tklink は解除され、var のグローバル値は初期化されるので, この後 var に set!するには define が必要になる。

(**tcl-echo** flag) *procedure*

flag が#t なら、TCL/TK の呼び出しの詳細を画面に表示する通常#f.

(**get-file-name** pat ...) *procedure*

[ss] file 撰択 dialog pat は "∗.ss" のように与える。選択が行なわれなければ #f を返す。

(**save-file-name** default-name [message]) *procedure*

[ss] file 名 入力 dialog file 名は、full path で返される。入力が無いか cancel ならば#f を返す default-name は””でもよい。 message は dialog に表示される。この dialog では名前の over write も check される。また新しい directory を作ることができる。この dialog を実行した後の working directory はもとの値である。

[TK] bind bind コマンドは、イベントに対する動作を定義する。たとえば

```
(label '.lab :text "Push")
(bind .lab "<1>" (lambda()(focus .lab)(format #t "Mouse-1 PUSHED!~%")))
(pack .lab)
```

とすると、このラベルはクリックできるようになる。この例のように、bind で与えるコマンドは必ず lambda 式で与えないといけない。 lambda 式は、引数を取ることもできるがその名前は TCL/TK でイベント キーワードとして定義されている。たとえば k はキーボードのコードを表す。

```
(bind .lab "<KeyPress>" (lambda(k)(format #t "Key ~s~%" k)))
```

としてクリックのあとでキーボードを押すとそのコードが標示される。TCL/TK の本ではベント キーワードは、%x,%y,%X,%Y,%k,%K のように記述されている。先頭の%を取った形で lambda 式の引数にする。このように bind では、引数の名前が重要なので lambda 式のみが許されている。すでにあるバインドを上書きせずに追加のバインドをしたいときは

```
(bind .lab "<1>" +(lambda(x y)(format #t "x~s y~s~%" x y)))
```

のように lambda 式の前に+を置く。クリックした位置も表示されるようになる。さらに細かい操作は bindtags コマンドでできる。

詳細は ss 入門を見てください。

8.2 OpenGL

OpenGL は、３Ｄグラフィックス用のＣ library である。 図形の表示、回転、拡大などが高速に行える。Linux では、Free の MesaGL が最初からインストールされており、大部分のグラフィックスカードが OpenGL に対応している。

以下がもしうまくいかないときは、Xの driver が OpnGL を含んでいるか/etc/X11/XF86Config で確認してほしい。

tkogl

通常 OpenGL は、C で書かれ、コンパイルして実行する。tkogl は、OpenGL を、より簡単に利用するために開発された。これは Tk(tool kit) の中に OpenGL の Tk widget を定義する形で Tk を拡張するので、Tcl/Tk に OpenGL が簡単に組み込める。そして、インタープリタ言語の TCl によってプログラムが可能になる。最初の開発者は Claudio Esperanca in 1997 である。詳細は

http://www.ece.ubc.ca/~hct/research/tkogl/

http://tcltk.free.fr/tkogl/

を参照。このシステムは Tcl/Tkogl と呼ばれる。

ss の拡張

ss には TK(ver. 8.4) が組み込まれている。従って、tkogl を入れることにより scheme から、Tk だけでなく OpenGL の関数が使えることになる。STk の作者 Erick Gallesio と tkogl の作者 Claudio Esperanca ならびにその協力者の人々に深く敬意を表する。

tkogl は現在 ver3.0(Windows 用) であるが、現在の ss 拡張には Claudio Esperanca の最初の version が用いられている。

変更したのは tkogl のソースの中の、 tkAppInit.c,tkogl.c,tkogl.h および

最初に ss を起動したら

```
ex.
user> (toplevel '.test)

user> (oglwin '.test.gl)
user> (pack  .test.gl)
user> (.test.gl 'main :clear 'colorbuffer
    :begin 'triangles
    :vertex 0 1 0
    :vertex -1 -1 0
    :vertex 1 -1 0
    :end
    )
```

で、白い３角形が表示されたら成功である。

OpenGL サンプル

いくつかのサンプルが、ss のソースの下の ogldemo の下に入っているので、これを見れば大体のことはわかるはずである。上の最初の例は、ここの simpledemo.ss なので

user> (load "simpledemo")

でも、同じ３角形が見えるはずである。

サンプルと OpenGL の解説書を比較するとよい。以下は概略である。また、tkogl のサンプルが

OGL/demo

の下にある。ogldemo のサンプルの多くはこれを ss に移植したものである。こちらも参考にすると良いだろう。

以下で [...] は省略可能である。

(**oglwin** 'name) *procedure*

name の OpenGL widget を定義する。name は必ず ドット . から始まる。

以下、この名前の widget を <ogl> で表す。

(pack <ogl>)

でウィンドウに <ogl> が配置される。

(<**ogl**> 'main :opt1 value1 .....) *procedure*

オプションで指定された OpenGL コマンドを main list とする。

(<**ogl**> **'newlist** [listnumber] :opt1 value1 .....) *procedure*

オプションで指定された OpenGL コマンドを新しい display list とする。もし、listnumber が与えられていれば、その番号の display list は書き換えられる。そうでなければ、新しい display list が作られその番号が返される。display list は:call listnumber で呼び出せる これは、コマンド キャッシュであり、高速にしかも何度も呼び出されるときに有効である。。

(<**ogl**> 'configure :opt1 value1 .....) *procedure*

widget のオプションを設定する。何もオプションを与えなければ、現在の設定を表示する。

(<**ogl**> 'deletelist number) *procedure*

number の display list を削除する。

(<**ogl**> 'eval :opt1 value1 .....) *procedure*

OpenGL コマンドを実行する。 ただし、その効果は redraw によって現れる。

(<**ogl**> 'redraw) *procedure*

main display list を call する。もし double-buffered ならば front と back を交換する。

OpenGL コマンド

詳細は OpenGL の解説書か、man コマンドで見ることができる。たとえば、man glAccum とする。

```
ss-option      arguments                                                OpenGL command
:accum         'operation value                                           glAccum
               operation={accum,load,add,mult,return}
:begin                                                                    glBegin
:end                                                                      glEnd
:clear          'mask                                                     glClear
        mask={colorbuffer,depthbuffer, accumbuffer,stencilbuffer}
:clearaccum     red,green,blue [,alpha]                                  glClearAccum
:clearcolor     red,green,blue [,alpha]                                  glClearColor
:cleardepth     red,green,blue [,alpha]                                  glClearDepth
:clearstencil   red,green,blue [,alpha]                                  glClearStencil
:color          red,green,blue [,alpha]                                  glColor
:colormask      red,green,blue [,alpha]                                  glColorMask
:colormaterial   'face   'mode                                           glColorMaterial
                face={front_and_back,front,back}
                mode={emission,ambient,diffuse,specular,ambient_and diffuse}
:call          listnumber                                                glCallList
:copypixels    x y width height                                          glCopyPixels
:newlist       listnumber  '{compile,compile_and_execute}                glNewList
```

このコマンドの後に続く、すべてのコマンド

`:endlist` `glEndlList`

第9章　正規表現

SS での正規表現 (regular expression) は現在 VER2.9 であり、TCL, egrep, Perl の仕様に準拠する。一部は拡張してある。

9.1　基本表現

pattern は文字列の一種であり、その中にメタ文字を置くことでいろいろな pattern を表現するのが、正規表現である。

pattern の一致は、最初に match する部分で最長のものが選ばれる。(最左/最長一致の原則という。)

`.`　改行以外の任意の 1 文字とマッチする。single モードでは、任意の 1 文字と match

`*`　直前のパターンが 0 個以上を示す。

`+`　直前のパターンが 1 個以上を示す。

`?`　直前のパターンが 0 個または 1 個を示す。

`{n,m}`　直前のパターンが n 個以上かつ m 個以下を示す。n=m なら{n}と書いても良い。

`[...]`　文字の集合のどれかとマッチ

```
^と\以外のメタ文字は、単に文字として扱われる。
0-9  a-z A-Z     範囲による指定
[^....]    は not
[を置くときはどこにおいてもよい。   [[123]
] は [の直後におく。                  []123]
-は   先頭または末尾におく          [-+]
これらは\[,\],\-を用いてもよい。
```

`^`　文字列の先頭。　multiple モードでは\n の直後とも match

`$`　行の終り (文字列の終りまたは改行の直前)

`(...)`　group　部分パターンを指定する。これと match した部分文字列はその group によって capture されたという。capture された文字列は \1,\2,... で後方参照ができる。つまり\number を書くと capture された文字列がそこにあるとみなされる。

`(?: ...)`　group　　capture しない。先頭部が 3 文字 (?:で残りが正規表現であることに注意。

`(?<name> ...)`　名前 付き group．複数のグループに同じ名前を使ってもよい。同じ名前は通常選択部分に対して行われる。そうでなければ混乱を招くだろう。

capture group，名前付き group はその左かっこ (の出現順に 1 から番号が振られる。

(?P=name)　名前による後方参照に使う。 この表現はグループではない。 name は英字から始まり英数字+-._のみが使える。

|　選択
たとえば #"what|where|how"は、what,where,how のどれかと match する。この場合もし what による match が見つかれば、残りの match は探索されない。また選択するパターンの範囲を指定するには group を用いる。
#"pc(100|80)" または#"pc(100)|80"は pc100 または pc80 に match する。group は表現の区切りも意味する。
#"a|(b)c|d"は ac,ad,bc,bd と match する。

正規表現を入力するときは

#".....”

の形式で行なうのが良い。これは、通常の scheme 入力形式ではなく ss 独自の read macro である。 ss の read は、この形式を読むと自動的に正規表現の中間実行形式にこれを変換する。このためのグローバル変数は

∗**auto-regexp**∗　*variable*

正規表現文字列を変換する関数を保持する。

である。この動作を止めるにはこの変数の値を#f にする。 この場合、#"...”は単に文字列を返す。#"...”の形式は、本質的に文字列を表すが、通常の文字列"...”と異なり、\"が”を表現する以外は \ の出現などをそのまま文字列とする。従って#"abc\n" は正規表現 abc\n を表す。これに対して通常の文字列表現では scheme の規定により \ は \\ と書かねばならない。つまり"abc\\n" と書くことになる。これが、特別な入力形式を採用した理由である。また、∗auto-regexp∗ によりあらかじめ解釈（中間的）が行なわれるので高速になる。

meta 文字を通常文字とするときは \ を使う。従って、#"2\∗3\"c" は文字列 2∗3"c と match する。\ 自身を 1 個の文字として正規表現に置きたいときは \\ と書くことになる。また

```
\n   改行文字 (10)
\r    return 文字 (13)
\t    tab
\e    escape
\f    改ページ
\a    alert (bell)
\v    vertical tab (\x0B)
\s    space,tab,\n,\f
\S    not space,tab,\n,\f
\d    0-9
\D    not 0-9
```

```
\w    0-9 a-z A-Z _
\W     not 0-9 a-z A-Z _
\k    2 バイト文字 (euc-code)
\K    1 バイト文字
\1,\2,\3, ...     指定した番号の group が最後に capture した文字列との
      match を行なう。 capture されていないときは match は不成功。後方
      参照という。
\<  単語の先頭 (英語のみ)
\>  単語の終り
\b  backspace    これは文字クラス [...] の中でのみ有効。
\b  単語の区切り    (\<または\>)
\B  単語の区切りでない。
\A   常に文字列の先頭を示す。
\z    常に文字列の最後
\Q    literal mode
       \E までの文字は, 単に文字自身を表す。
\E    end of literal mode
\cA,...,\cZ       ctrl-A, ...
\x0 ... \x3FFFFFFF     16 進数により文字を指定する。

\G    下記で解説。
\g

\#    match には関係せず無視される。単なる区切り文字として使用できる。
\!    Fail     これとの match は常に失敗する。
```

これらは特別な意味の 1 個の文字として利用できる。\ のあとがこれら以外の場合は \ は単に直後の文字 (meta とはみなされない) を escape していると解釈される。これらの特殊文字を [...] の中においても良い。

9.2 regexp 関数

(**regexp** string pattern [flag]) *procedure*

flag は

```
なし    一致した最初の部分文字列を返す。(default)
:repeat    一致する部分文字列または index のリストを返す。
           繰りかえしは、一致文字列の後の部分に対して行なわれる。
:index    一致する最初の部分の index をドット対 (begin . end) として
返す。
:nocase     文字の比較で大文字、小文字の区別をしない。
:ci           same as :nocase
:single          single mode
```

:multiple	multiple mode
:cpos	match に失敗しても pos を reset しない。
:debug	NFA の各ステップを表示する。

これらのオプションは適当に組み合わせて指定してよい。一致しないときは、#f が返される。pattern は正規表現、文字列または regexp->nfa によって 生成した NFA である。

例 9.1. *user* >(regexp "This is a pen." #"\w+" :repeat)
("This" "is" "a" "pen")

正規表現が空文字列に match する"x∗"のような場合に、repeat の指定は無限ループに落ちてしまうので注意してほしい。

9.3 NFA

(**regexp->nfa** str) *procedure*

str は文字列または#"..."による正規表現である。#"..."は文字列を解釈してある中間形に変換する。これはさらに NFA(非決定性オートマトン) に変換されて実行される。regexp->nfa はこの NFA をあらかじめ生成する。ループの中で同じ正規表現を使用するときは、外側でこれを実行して regexp に渡すと speed up になる。

例 9.2. *user* >#".∗"
((0 () (0 . #f) (#f . #f) #f) (:∗ ((string regexp:r-dot))))
user >(regexp->nfa #".∗")
((0 () (0 . #f) (#f . #f) #f) .
#(0 (7 . 2) (3 . 4) (regexp:back (2)) ((:or 5 0)) (6 . 2)
(string regexp:r-dot) (regexp:back-init (2))))

9.4 regexp-subst

(**regexp-subst** string pat replace [:repeat :nocase]) *procedure*

string の pat と match する部分を replace で置き換えた文字列を返す。:repeat が指定されたときはすべて置き換えられる。一致しないときは、#f が返される。replace には文字列でなく、1 引数の関数を与えることもできる。その場合、関数は一致部分文字列を引数として呼ばれその関数が返す文字列が代入されることになる。

例 9.3. *user* >(regexp-subst "This is a pen" #"\sis\s" " are ")
"This are a pen"

user >(regexp-subst "This is a pen is" #"\sis(\s|$)" " are " :repeat)
"This are a pen are "
user >(regexp-subst "aaa 111 222 ccc" #"\d+" (lambda (x) (string-append "(" x ")")) :repeat)
"aaa (111) (222) ccc"

9.5 get-group

(**get-group-string** n) *procedure*

(**get-group-index** n) *procedure*

この関数の呼び出しの前に regexp または regexp-subst の呼び出しがなければいけない。n は group 番号 (0 以上) または名前 (文字列) であり対応する文字列のリストまたは index が返される。もし、その group が後続の+,* により複数個の match を持っているなら、返されるのは 2 個以上の文字列のリストになるだろう。もし、照合したがその group に対してでないのなら#f が返される。regexp の呼び出しが:repeat によるものなら返されるのは各呼び出しの結果のリストである。

例 9.4. *user* >(regexp "abc def" #"(\w)+")
"abc"
user >(get-group-string 0)
"abc"
user >(get-group-string 1)
("a" "b" "c")

例 9.5. *user* >(regexp "-123+234-567" #"(\d+)|(\D+)" :repeat)
("-" "123" "+" "234" "-" "567")
user >(get-group-string 0)
("-" "123" "+" "234" "-" "567")
user >(get-group-string 1)
(#f ("123") #f ("234") #f ("567"))
user >(get-group-index 2)
(((0 . 1)) #f ((4 . 5)) #f ((8 . 9)) #f)

9.6　match

(**match-string** n) *procedure*

(**match-index** n) *procedure*

この関数の呼び出しの前に regexp または regexp-subst の呼び出しがなければいけない。 あるいは、regexp における関数呼び出しの中で使ってもよい。n は group 番号 (0 以上) または名前（シンボル) であり, その group が (その時点で）最後に capture した文字列が返される。capture されていなければ #f が返される。0 は、match 全体か、内部呼び出し時点での位置までの文字列となる。この関数は、たとえ regexp が失敗に終ってもその NFA の最終状態から値を返す。

例 9.6. ;;;; after the above
user >(match-string 0)
"567"
user >(match-string 1)
"567"
user >(match-string 2)
#f

9.7　mode 制御

```
(?i)     :nocase
(?-i)     case
(?s)     single mode.   . は任意の文字と match する。
(?m)     multiple mode.  ^ は文字列の先頭または改行の直後の位置と
        match する。
(?n)    normal mode &&  case (default)
```

これらは、正規表現の中におかれる。そして、match の流れにそって実行される。たとえば

例 9.7. *user* >(regexp "PC-30 hp-1 pc-10 HP-3" #"((?i)pc(?-i)|HP)-(\d+)" :repeat)
("PC-30" "hp-1" "pc-10" "HP-3")

は間違った例である。この場合 pc の match に失敗すると、その後の (?-i) は実行されないので、HP も:nocase で比較されてしまう。従って正しくは

例 9.8. *user* > (regexp "PC-30 hp-1 pc-10 HP-3" #"((?i)pc|(?-i)HP)-(\d+)" :repeat)
("PC-30" "pc-10" "HP-3")

のように書く。(?i) などはグループではないので選択 | の区切りにはならないことも注意して欲しい。

`(?x)` `free space & #comment mode`
これは、正規表現の記述を見やすくするために用意されている。
この `mode` では表現における、 空白、改行、`tab` 文字は無視され, さらに
`#`から行末までも無視される。

`(?X)` 通常にもどす。

例 9.9. *user* >(define x (regexp->nfa
#"(?x) (<A\s+[^>]+> \s*)? # <A > tag
<IMG\s+[^>]+> # <IMG> tag
(?(1)\s*</A>) # </A> if <A>
"))
user >(regexp "test <A b> <IMG test> </A>" x)
"<A b> <IMG test> </A>"

9.8 先読み戻り読み

`(?=...)` 先読み
`(?!...)` 否定の先読み
`(?<=...)` 戻り読み
`(?<!...)` 否定の戻り読み

これらは任意の正規表現を使用して良い。先読みは与えられた表現と match する文字列の先頭位置を示す。否定の先読みは、それが表現と match しない位置であることを 意味する。逆に戻り読みは表現と match する文字列の終りの位置を示す。これらは、その位置の正当性を check する。これらは一種の capture なしのグループである。もし capture が必要なら表現の中にグループを書けばよい。

例 9.10. *user* >(regexp-subst "if :kkk u:ss :sort" #"(?<=\B)(?=:s)" "skeyword")
"if :kkk u:ss skeyword:sort"

9.9 regexp-pos

regexp は、前回の match が成功で終ったときその終りの位置と文字列を記憶している。全く同じ文字列に対して regexp が行なわれるとき、\g は前回以後の位置に match する。これに対し \G は正確に前回の位置とだけ match する。もし、前回の match が不成功かまたは違う文字列に対する regexp の呼び出しのときは regexp は、記憶した位置を reset する。このとき、記憶位置は 0 なので \g は単に無視され、\G は先頭を示すことになる。もし、失敗しても前の記憶位置を保存したいときはオプション :cpos を指定する。この機能は繰りかえしをより細かく制御したいときに有用である。

例 9.11. *user* >(define str "a b cd ef")
str

user >(regexp str #"\g\w+")
"a"
user >(regexp str #"\g\w+")
"b"

:cpos

regexp option. 繰り返しにおいて、失敗したとき他の match を試したいときに有用である。

\g,\G は通常、表現の先頭に置かれるがそれ以外の位置においてもよい。

(**regexp-pos**) *procedure*

現在の記憶位置を返す。

(**regexp-pos** n) *procedure*

次に実行される regexp における、記憶位置をあらかじめ指定する。 このときは、対象文字列が変更されてもこの指定は有効である。

9.10　条件式

(?(n)then|else)

n はグループ番号または名前である。もしそれが capture されていれば then がためされ、そうでなければ else がためされる。else は省略できる。else が省略されたとき、条件が偽ならこの表現は単に無視されて探索は続行する。そのような場合失敗させたいなら else 部に \!を書くとよい。else 部分があるとき選択の | はトップレベルで表現を選択していなければならない。これも、1 種の capture なしグループである。

(?(?=...)then|else)

条件に (?!,(?<=,(?<! などの先読み戻り読みを使ってもよい。

(?f(proc p1 ...)then|else)

proc はグローバル scheme 関数名または lambda 式である。p1 ... は番号または名前なら capture された文字列または#f として proc にわたされる。それ以外のパラメータは評価されずにそのままわたされる。文字列をわたしたいときは表現の内部なので \"...\"のようにエスケープが必要であることに注意する。パラメータは無くてもよい。 関数の帰り値は真偽を示す。

例 9.12. *user* >(regexp "text is 2000 yen, is not 3000 yen" #"(?<w>\w+)|\d+(?(w)\s+is|\s+\yen)" :repeat)
("text is" "2000 yen" "3000 yen")

9.11 関数呼び出し

(?P(proc p1 ...))

proc の呼び出しが行なわれ、帰り値の (単純) 文字列との照合が行なわれる。""を返すようにすれば、正規表現の debug に用いることもできる。

例 9.13.
```
user >(define (pr x) (format #t "regex-print ~s~%" x) "" )
     pr
user >(regexp "Jeff read a book." #"(\w+\s+(?P(pr 0)))+")
     regex-print "Jeff "
     regex-print "Jeff read "
     regex-print "Jeff read a "
     "Jeff read a "
```

9.12 動的表現

(??{varname})

(??{(proc p1 ...)})

varname はグローバル変数名であり、その値は正規表現 (または文字列、NFA) である。これによって、正規表現を部品にしたり再帰的に定義できる。関数呼び出しのときは、それは正規表現を返さなければいけない。これらは必要なら NFA に変換されて、その時点でのモードで実行される。もし、その NFA によってモードの変更が行われても元のモードが復帰される。

例 9.14.
```
user >(define Rlist #"\([^()]*((?(?=\()(??{Rlist})|[^)])[^()]*)*\)")
     Rlist
user >(regexp "ss (abc(d(g)e(f)))" Rlist)
     "(abc(d(g)e(f)))"
```

例 9.15.
```
user >(define Ratom (regexp->nfa #"[^()\s]+"))
     Ratom
user >(define Rs (regexp->nfa
     #"(?x) (??{Ratom}) | # Rs is an atom or
     ( \( # (
     ( \s* (??{Rs}) \s*)* # Rs ...
     \) # )
     )
     "))

     Rs
user >(regexp "ss (abc(d(g)e(f)))" Rs)
     "ss"
user >(regexp " ( abc (d (g) e( f)))hhh" Rs)
     "( abc (d (g) e( f)))"
```

9.13　非欲張り演算子

```
*?
+?
??
{n,m}?
```

通常 *,+などは欲張りである。つまり、match する限り繰り返したあとで、失敗すればそこで初めて back-track により他の可能性を探索する。成功したらその時点で match は終了する。この意味で、本システムでは必ずしも最長 match は保証されていない。非欲張り演算子が指定されたときは、その match をしないで最終までいけばそこでやめてしまう。失敗したときに、match を増やしながら探索を続行する。

例 9.16. *user* >(regexp "where is an example." #"\w*.e")
"where"
user >(regexp "where is an example." #"\w*?.e")
"whe"

9.14　atomic

(?>...)

atomic なグループの指定である。atomic な場合、そこで選択や *,+, などで match が起きた場合、このグループの match を最初からやりなおす場合を除いて back-track しない。これは強欲と呼ばれる。つまり、この部分で match が起きてその後ろで失敗してもこの部分における他の可能性が試されることはない。さらに前に戻って、探索が行われることになる。

```
*+
++
?+
{n,m}+
```

これらは、強欲であり.*+と (?>.*) は同等である。

例 9.17. *user* >(regexp "where is an example." #"\w*+.e")
"an e"
この場合、最初の where は \w*+によって飲み込まれてしまい、それは譲ることが無いので先頭での match は失敗する。

9.15　regexp-split

(**regexp-split** str regex [option])　　*procedure*

regex で match した部分を区切りとして、str を分割した文字列のリストを返す。失敗すれば#f を返す。

例 9.18. *user* >(regexp-split "133.14.64.128" #"\.")
("133" "14" "64" "128")

第10章　compiler,debug

10.1　compiler

makesr *shell script*

compie 用のシェルスクリプト。

(**compile-file** filename) *procedure*

file を compile して、 ”.sr”の拡張子のついたファイルを作ります。compile によって load の速度があがり、実行速度も良くなります。ただし、この関数は通常の起動では読みこまれないので

(require ”compiler”)

を実行する。

(**compile** obj) *procedure*

内蔵された 1-pass compiler の出力を見る。

(**make-m** <list> <length of list>) *procedure*

10.2　trace

(**trace** f1 f2 ...) *procedure*

関数, 古典マクロ (built-in でも良い) を trace

(**untrace** [f1 f2 ...]) *procedure*

引数がなければ、すべて untrace する。

(**trace-var** symbol thunk) *procedure*

symbol の値 (global) が変更されたとき実行するコマンドを指定する。繰りかえし指定したときは、最後に登録されたコマンドから順に実行される。symbol は tklink であってもよい。

(**untrace-var** symbol) *procedure*

10.3 debug loop

Ver1.99 では error が起きたとき、error-gui.ss を使っている。しかし、今のところこれは無限 loop に入ることが多く使いにくい。1.99 以前の interface の方が好ければ, sslib/tkinit.ss において行 (load "error-gui") をコメントにして

makesr tkinit

を実行する。そして src directory において

make install

を実行すればよい。

10.4 sn または 1.99 以前の debug loop

ss において、なんらかの error が発生したとき system は error に対応したメッセージと関連する scheme object を印刷し、さらにそのときの環境と関数呼び出しのリストを印刷したあと debug 用の read-eval-print ループに入ります。このときのプロンプトは

debug>

となる。このループからの脱出は :q を入力する。

```
debug>:h
debug> :q
 :q
 jump here mctop  level=2

user>
```

関数呼び出しリストは

```
debug> 30 :h
 30
debug> :h
env: jmp to top of debug MC loop :h  entered n 30
 Func & Macro (:T ) history
his 1: pair?
his 2: macro-expand-rec
his 3: funcall
his 4: cdr
his 5: null?
his 6: cdr
.......
......
his 30: pair?
debug>
```

のように、

数 :h

と入力すれば再表示できます。数の最大値は 99. また単純なエラーの場合は

debug>:n

と:n を入力することにより、次の命令から実行を続けることができる場合があります。またエラー原因を解消したあと

debug>:c

でエラー時点から再度実行ができるかもしれません。たとえば

user> (do ((i 0)) ((= i 10)) (inc i) foo)

と実行すると、foo のところでエラーがおこります。i を入力すると変数 i の変化が確認できます。:n の入力によりループを継続できます。

10.5 catch

error を trap するには、catch-error,catch を使うことができます。これらは call/cc と *errorhook* を用いたマクロです。trace,trace-var の利用も有効でしょう。

(**help** word ...) *procedure*

word ... をすべて含む sslib/Doc/のファイルを撰択して表示する。ss では、GUI を使った help が実行されます。<F3> で keyword の位置を移動できます。

(**debug** [obj] [env]) *procedure*

obj を印刷して read-eval-print ループに入る。この呼び出しの起こった環境または env のもとで eval が行なわれる。ループからの脱出は :q を入力する。

(**mcprint** function [port]) *procedure*

define で定義した関数 object や mceval で作成した object の内容を表示する。表示は compile された中間コードである。

(**error** errno obj) *procedure*

error を発生する。255 以下の errno は sslib/sserror.ss で定義されている。256 以上は user 用である。system は error が発生したとき、通常 errno に対応したメッセージと obj を印刷して、debug 用の read-eval-print ループに入る。ループからの脱出は :q を入力する。

system-error-msg *variable*

要素が (number . "message") の形のリストである。このリストに追加 (push) することにより error 関数が指定したメッセージを出力するようにできる。

(**system-error** errno obj) *procedure*

error を発生する。これは、*errorhook* の影響を受けない。

(**print-error** errno obj [path]) *procedure*

errno に対応したメッセージと obj 印刷する。

∗**errorhook**∗ *variable*

この global 変数には、2 引数の関数を set できる。その引数は errno,obj である。それが定義されたとき system で発生した error は、通常の error 表示関数の呼び出しではなく ∗errorhook∗ に set された関数の呼び出しになる。∗errorhook∗ の値はこの関数の実行時に別の値に変更されるので、この関数は、その実行の最後に ∗errorhook∗ を set!するコードを含むのが普通である。(別の値に変更されるのは、関数内部でエラーが発生して無限ループになるのを防ぐためである。) ∗errorhook∗ の値を関数以外に set すれば、hook は無効になる。これは、call/cc と組み合わせて使うのが普通であろう。

∗**errorenv**∗ *variable*

error が発生したときの環境が set される。

(**catch-error** thunk error-proc) *macro*

thunk は引数なしの関数 object または実行可能な S 式である。error-proc は 2 引数の関数である帰り値は、thunk の実行が正常終了すればその返す値であり、もし error が発生したら error-proc の帰す値である。

(**catch** form1 ...) *macro*

form1 ... を順に実行する。もし途中でエラーが発生したらそこで実行をやめて#t を返す。正常終了なら#f が返される。

(**call-history**) *procedure*

関数呼び出しのリストを返す。 call-history 自身が先頭にあるリストが返される。

以下は通常のプログラミングには使用できない。

(**mcompile** form env) *procedure*

form を compile して実行可能な中間コードからなる object を作る。

(**mc-eval** obj env) *procedure*

mcompile で作成した obj を評価する。

(**meval** form env) *procedure*

第11章　SSLIB

11.1　[SS] graph

無向グラフを扱う関数群である。グラフは、((a . b) (b . c) (a . d)) のように、頂点を結ぶドット対のリストとする。ただし、(a . b) は同時に (b . a) も意味するものとする。

(**find-edge** v graph)　*procedure*

user >(find-edge 1 '((0 . 1) (2 . 0) (1 . 3)))
(0 . 1)

(**connect?** v1 v2 graph)　*procedure*

user >(connect? 1 0 '((0 . 1)))
(0 . 1)

(**nearest** v graph)　*procedure*

user >(nearest 0 '((0 . 1) (2 . 0) (1 . 3)))
(1 2)

(**remove-vertex** v graph)　*procedure*

user >(remove-vertex 0 '((0 . 1) (2 . 0) (1 . 3)))
((1 . 3))

(**remove-edge** edge graph)　*procedure*

user >(remove-edge '(0 . 2) '((0 . 1) (2 . 0) (1 . 3)))
((0 . 1) (1 . 3))

(**co-vertex** v edge)　*procedure*

user >(co-vertex 2 '(1 . 2))
1

(**move-vertex** v1 v2 graph)　*procedure*

v1 を v2 にする。*user* >(move-vertex 0 1 '((0 . 1) (2 . 0) (1 . 3)))
((1 . 1) (2 . 1) (1 . 3))

(**reduce-graph** graph)　*procedure*

user >(reduce-graph ' ((1 . 1) (2 . 1) (1 . 3)(1 . 2)))
((2 . 1) (1 . 3))

(**edge-assoc** edge assoc-list-for-edges)　*procedure*

user >(edge-assoc '(1 . 2) '(((3 . 4) . a) ((2 . 1). b)))
((2 . 1). b)

11.2 [SS] set 集合演算

(**list->set** list) *procedure*

user >(list->set '(2 4 4 8))
(2 4 8)

(**add-element** x set) *procedure*

user >(add-element 2 '(3 5))
(2 3 5)

(**set-union** s1 s2 ...) *procedure*

user >(set-union '(1 3 2) '(2 4 5 3))
(5 4 1 3 2)

(**set-intersection** s1 s2 ...) *procedure*

user >(set-intersection '(1 3 2) '(2 4 5 3))
(2 3)

(**set-difference** s1 s2 ...) *procedure*

s1-s2- ... *user* >(set-difference '(1 3 2) '(2 4 5 3))
(1)

(**set-intersection?** s1 s2 ...) *procedure*

return #t if not empty intersection *user* >(set-intersection? '(1 3 2) '(2 4 5 3))
#t

(**subset?** A B) *procedure*

user >(subset? '(1 3 2) '(2 4 5 3 1))
#t

(**superset?** A B) *procedure*

user >(superset? '(1 3 2 5 4) '(2 4 5 3 1))
#t

(**set-member?** x set) *procedure*

user >(set-member? '(1 2) '(3 4 (2 1)))
#t

(**set-remove** x set) *procedure*

user >(set-remove '(2 (1)) '(3 ((1) 2) 4))
(3 4)

(**set-equal?** A B) *procedure*

return #t if (equal? A B) or A=B as sets! *user* >(set-equal? '(1 3 2 5 4) '(2 4 5 3 1))
#t

(**forall-element** set test) *procedure*

returns #t if test suceeds for all element of set *user* >(forall-element '(2 3 4 #t) number?)

#f

(**exists-element** set test) *procedure*

returns the element if test suceeds for an element of set else return #f *user* >(exists-element '(2 3 4 #t) number?)

2

11.3 [SS] ssed

(**ssed** [file ...]) *procedure*

起動 scheme の内部から起動するときは

(require "ssed")

の後

(ssed)

または

(ssed file1 file2 ...)

Linux のコマンドラインから使用するときは

ssed

または

ssed file1 ...

とする。

操作

[**ファイル**] **open** 新規なら，file を選択して読み込むすでに，読み込んだ file があるときは，New window 挿入（カーソル位置に）か，append（文書の最後に追加）を選ぶ

save 忘れずにして下さい。上書きされます。名前を変えるときは 1 番上にある入力欄で指定

new 新しい sed を起動

Home

[**編集**] マウス左で範囲を指定してから

Copy <CTRL-C>

Cut <CTRL-W>

Paste <CTRL-Y>

このメニューは、マウスの right クリックでも現れる

[**Tool**] **SSload** 編集中の内容を ss に読ませま

Check 入力カーソルの手前までを check する直前にあるリストは brown すべての文字列 green 注釈 pink

Tag clear 設定されている tag を消去

Tag jump tag に jump. <F7> と同じ

Jump line 行番号を指定して jump

＊＊ 自動 check ＊＊”と）が入力されたとき自動で Check する

Find 入力すると自動で捜す。return キーでテキストに戻る <F3> で，見つけた word を移動できる再検索のときは，このボタンを押すか，入力欄で return または <F3>.

replace find で見つけた，word を置換するには，この横に入力して return 問い合わせの dialog が出るのでキー y (置換）,n (no), a (残りのすべてを置換）,c (残りの置換をキャンセル) のどれかを押す。置換した文字列は，<F5> で，捜せるこのボタンは，再置換や，単に word の消去のとき使

キーバインド

CTRL-c コピー

CTRL-w カット

CTRL-y 貼り付

<**F3**> jump to found

<**F5**> jump to replaced

<**F7**> jump to tag (marks)

CTRL-Q tag set

<**Button-3**> popup-menu of Edit

11.4 [ssftp] usage

From a unix terminal window , you can exec the command

ssftp

then sesseion window will appear. Fill host and user (user can be omitted) and push 'SAVE'. Saved session can be loaded by Double-click next time.

Then push 'OK'. You will be asked password in the terminal. After the success of ssh connection , main window will appear.

You can select files by using Mouse, SHIFT-Mouse and CTRL-Mouse , which is the usual selection method for the listbox in TK.

Then copy by >> or << , move by -->> or --<<.

Directory can be copied , moved or removed recursively.

Caution:

```
symbolic link is copied when it links to a regular file.
And the result is a regular file not symbolic link.
If the link is to a directory, it is not copied!!!
In this case, some errors will be reported in the terminal window.

So, if you copied a directory and it includes some symbolic links,
then it is not the same to the original.
In such a case, you should use other programs: scopy, rsync -e ssh,
tar, etc.
```

This program is running 'sftp' in the background. This program only handles GUI interface for sftp. The source can be found in $prefix/lib/sslib/SSFTPxxx.

索 引