Scheme transcripts

Scheme is a language having a number of implementation-dependent features. All Scheme transcripts that are included in this Reader are generated using the Texas Instruments PC Scheme 3.03. Therefore, if executed in a different environment, the presented programs might sometimes generate results that are not identical to the presented results.

```
-----------------------------------------------
PROGRAM :  Sample Scheme Session
AUTHOR  :  Jozo J. Dujmovic
DATE    :  May 1, 2003
-----------------------------------------------

PC Scheme 3.03  07 June 88
(C) Copyright 1988 by Texas Instruments
        All Rights Reserved.

[PCS-DEBUG-MODE is OFF]

[1] (transcript-on "arith.log")      ; Definition of the Scheme
                                     ; Session Transcript File

OK
[2] ; Comments start with the semicolon
    ; and can have multiple lines.

    ; Blank lines can also be included.

    ; Comments cannot be the only contents after the prompt.
    ; Prompt for each command is provided by Scheme in the form

    ;              [integer]

    ; where integer denotes the number of the next expression to
    ; be entered.
    ; Comments can be added after each Scheme command


    ;---------------------------
    ;   C o n s t a n t s
    ;---------------------------

1   ; Constants evaluate to themselves:
1

[3] 0.12345678901234567890  ;  Accuracy of real numbers
0.123456789012346
```

```
[4] -1e33                          ; Exponential form of real constants
-1.e33

[5] -1e99
-1.e99

[6] -1e-123
-1.00000000000001e-123

[7] -1234.56789
-1234.56789


[8] ;-------------------------------------------------------------
    ; Arithmetic operators use the paranthesized prefix notation:
    ; ( <operator> <operand1> <operand2> ... <operand n> )
    ;-------------------------------------------------------------

    ; The number of operands can be zero:
(+)
0

[9] (-)
0

[10] (*)
1

[11] (/)
1

[12] ; The case with a single operand:
     ; -----------------------------
(+ 3)
3

[13] (- 3)
-3

[14] (* 3)
3

[15] (/ 3)
0.333333333333333

[16] ; Two operands:
     ; ------------
(+ 1 2)
3

[17] (- 1 2)
-1
```

```
[18] (* 1 2)
2

[19] (/ 1 2)
0.5

[20] ; Multiple operands:
   ; ------------------
(+ 1 2 3 4)       ;  Equivalent to (+ (+ (+ 1 2) 3) 4), i.e. 1+2+3+4
10

[21] (- 1 2 3 4) ; Equivalent to (- (- (- 1 2) 3) 4), i.e. 1-2-3-4
-8

[22] (* 1 2 3 4) ; Equivalent to 4!=1*2*3*4
24

[23] (/ 1 2 3 4) ; Equivalent to 1/2/3/4 = 1/24
0.0416666666666667

[24] (/ 1e77 1e75)
100.

[25] (/ 777.777 7)
111.111

[26] 12345678901234567890123456789O ; integers can have arbitrary
   ; precision because they are internally stored as list of
   ; digits, rather as binary numbers
12345678901234567890123456789O

[27] (exit)       ; End of Session (the transcript file will be
                  ; automatically closed)


[2]
   ;-------------------------------------------------------------
   ; Atoms:  indivisible data objects
   ;         Numbers, Strings, and Symbols
   ;-------------------------------------------------------------

   123     ;  integer: numerical constants evaluate to themselves:
123

[3] 1.23e-17  ;  real
1.23e-17

[4] "Character String"   ;  character strings evaluate to themselves:
"Character String"

[5] "123.456"            ;  character string, not a real constant
"123.456"
```

```
[6] ""                    ;  empty string
""

[7] true                  ;  symbol having a predefined value
#T

[8] #T                    ;  an equivalent notation for true
#T

[9] #t
#T

[10] false                ;  symbol having a predefined value #F or ()
()

[11] #F
()

[12] ()
()

[13] t                    ;  t and nil are additional predefined symbols
#T

[14] nil
()
[15] Variable             ;  undefined symbol

[VM ERROR encountered!] Variable not defined in current environment
VARIABLE

[Inspect] Quit
[16] (exit)
```

INPUT FILE START.S

```
(define (start prog log)  ;  Definition of function start
  (transcript-on log)     ;  Open the output transcript file
  (display "=================================================")
  (newline)
  (display "PROGRAM :  ")
  (display prog)          ;  prog is an input string (program name)
  (newline)
  (display "AUTHOR  :  Jozo J. Dujmovic")
  (newline)
  (display "DATE    :  SEP 6, 2004 ")
  (newline)
  (display "=================================================")
  (newline)
)
```

```
                                                          PCS DIALOG
PC Scheme 3.03  07 June 88
(C) Copyright 1988 by Texas Instruments
          All Rights Reserved.

[PCS-DEBUG-MODE is OFF]

[1] (load "start.s")      ;  Load the input source file start.s
OK
[2] (start "Addition demo program"  "add.log")      ; Exec. start fun.
================================================
PROGRAM :  Addition demo program
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 6, 1994
================================================
[3] (+)
0
[4] (+ 1)
1
[5] (+ 1 2)
3
[6] (+ 1 2 3)
6
[7] (exit)
```

```
                                  OUTPUT TRANSCRIPT FILE ADD.LOG
================================================
PROGRAM :  Addition demo program
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 6, 1994
================================================
[3] (+)
0
[4] (+ 1)
1
[5] (+ 1 2)
3
[6] (+ 1 2 3)
6
[7] (exit)
```

```
[1] ;-------------------------------------------------------------
    ;   Symbols and recognition of objects of the type symbol
    ;-------------------------------------------------------------
    x     ; x is an undefined variable

[VM ERROR encountered!] Variable not defined in current environment
X

[Inspect] Quit

[2] (+ x  2)  ; arithmetic expression with an undefined variable

[VM ERROR encountered!] Variable not defined in current environment
X

[Inspect] Quit

[3] t     ; true (logical constant - PCS only!)
#T

[4] true ; same as t
#T

[5] false
()

[6] nil  ; False and nil are interpreted as an empty list ()
()

[7] f     ; f is not an abbreviation for false; it is an undefined variable

[VM ERROR encountered!] Variable not defined in current environment
F

[Inspect] Quit

[8] #F                    ; constant false
()

[9] #T                    ; constant true
#T

[10] (quote x)            ; definition of symbol x
X
[11] 'y                   ; short version for (quote y)
Y

[12] 'y
Y

[13] 'abcde               ; symbol abcde
ABCDE
```

```
[14] '(+ 1 2)              ; this is a list, not a symbol
(+ 1 2)

[15] (symbol? 'x)          ;  symbol? is a recognizer for symbols
#T

[16] (symbol? '7)          ;  '7 is not a symbol
()

[17] (symbol?  7)          ;  constant 7 is not a symbol
()

[18] (symbol? '(+ 1 2)) ; the list is not a symbol
()

[19] (symbol? 'abcde)    ; symbol can have more than one character
#T

[20] (symbol? 'ab)
#T

[21] (symbol? 'a)
#T

[22] (exit)


[2] ;-----------------------------------------------------
    ;  DEFINE :  definition of variables and functions
    ;-----------------------------------------------------

x   ;  undefined variable
[VM ERROR encountered!] Variable not defined in current environment
X
[Inspect] Quit

[3] (define x 123)   ;  x := 123
X

[4] x
123

[5] (define y 321)   ;  y := 321
Y

[6] (+ x y)    ;  expression with previously defined variables
444

[7] '(+ x y)   ;  quote prevents the evaluation of this expression
(+ X Y)

[8] (define (double x) (* 2 x)) ; def of function double(x) = 2*x
DOUBLE
```

```
[9] (double 123)     ;  evaluation of double(123)
246

[10] '(double x)     ;  evaluation prevented
(DOUBLE X)

[11] x
123

[12] (double x)      ;  evaluation of double(x) where x=123
246

[13] (double (- 1 2)) ; x is replaced by -1
-2

[14]
(define (square n) (* n n))  ; definition of square(n) = n*n
                             ; where n is a local variable of
                             ; function square (a formal argument)
SQUARE

[15]
(define square_2 (lambda (n) (*n n))) ; lambda is a general form
           ; of an anonymous function: lambda(n) = n*n .
           ; This function is named square_2.
           ; Can you see an error in this definition?
SQUARE_2

[16] (square_2 10)

[VM ERROR encountered!] Variable not defined in lexical environment
*N

[Inspect] Quit

[17] (define square_2 (lambda (n) (* n n)))   ;  correct the error
SQUARE_2

[18] (square_2 10)
100

[19]
(define (square_3 n) (square n)) ; square_3 uses previously
                                 ; defined square
SQUARE_3

[20] (square_2 15)
225

[21] ; the function definition can include both the definition
     ; and the use of another locally defined function:
```

```
(define (square_4 n) (define (sqr n) (* n n)) (sqr n))
SQUARE_4

[22] (square_4 9)
81

[23] (exit)


[2] ;-----------------------------------------------------------------
    ;  LATENT TYPING:  Scheme does not use static data types
    ;  that are defined at the beginning of the program and
    ;  that last during the whole execution of the program.
    ;  Data types of variables and objects passed to a function
    ;  are defined and checked at run time, not during compile
    ;  time ("run-time type checking").
    ;-----------------------------------------------------------------

(define x 1234567)          ;   x is an integer
X

[3] x
1234567

[4] (define x 123.4567)     ;   x is now redefined as a real,
X

[5] x
123.4567

[6] (define x "String")     ;   string
X

[7] x
"String"

[8] (define x '(1 2 3 4))   ;   list
X

[9] x
(1 2 3 4)

[10] (define x '(+ a b))
X

[11] x
(+ A B)

[12]; Each new definition assigns a new value to a variable.
   ; The process is destructive for previous values
   ; ("destructive binding"). Similar effect can be achieved
   ; using the assignment operator set! :
```

```
(define x)
X

[13] x
#!UNASSIGNED

[14] ; x is here defined but "unbound", it is a general
     ; variable without a specific type and any specific value.
     ; The type and value can be explicitly assigned using set! :
(set! x 3)
3

[15] (set! y 4)     ;  set! cannot be used for defining variables
[VM ERROR encountered!] SET! of an unbound variable
Y
[Inspect] Quit

[16] (define y 4)
Y

[17] (define op)
OP

[18] (set! op +)
#<PROCEDURE +>

[19] (op x y)
7

[20] (define op *)
OP

[21] (op x y)
12

[22] ; definitions and redefinitions of variables and
     ; functions yield a very high level of flexibility
     ; in program design
(exit)

-----------------------------------------------
PROGRAM :  DEFINE-READ (reading from keyboard)
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 25, 1994
-----------------------------------------------
[7] (define x (read)) ; read x as next value from the keyboard
123
X

[8] x
123
```

```
[9] (set! x (read))
321
321

[10] x
321

[11] (define x (read))
1 2 3
X

[13] x          ; Only the first value
1

[17] (define p '(12 . 34))
P

[18] p
(12 . 34)

[19] (begin (define a (read)) (define b (read))) ; Sequence
1 2
B

[20] a
1

[21] b
2

[23] (begin (set! a (read))
            (set! b (read))
            (display a)
            (display " ")
            (display b)
            *the-non-printing-object*)
alpha beta
ALPHA BETA

[24] a
ALPHA

[25] b
BETA

[28] (let ((a (read))) (display "a = ") (display a))
alpha
a = ALPHA


[29] (let (( z (read))) (display z))
"string"
```

```
string

[30] (let (( z (read))) (display z))
(1 . 2)
(1 . 2)

[31] (car z)

[VM ERROR encountered!] Variable not defined in current environment
Z

[Inspect] Quit

[33] (define z (read))
(1 . 2)
Z

[34] z
(1 . 2)

[35] (car z)       ! Head
1

[36] (cdr z)       ! Tail
2

[37] (define z (read))
(1 2)
Z

[38] z
(1 2)

[39] (begin (display (car z))(display " ")(display (cdr z)))
1 (2)

[40] (let ((z (read))) (display "z = ") (display z)) ! local scope
( a (b c) d )
z = (A (B C) D)

[41] z      !  This is previous scope
(1 2)

[42] x
1

[43] (let ((z (read))) (set! x z))
( (a b c) . (1 2 3) )
((A B C) 1 2 3)

[44] x
((A B C) 1 2 3)
[45] (exit)
```

```
-----------------------------------------------
PROGRAM :  Elementary functions
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 15, 1994
-----------------------------------------------

[3] (abs -3.14)
3.14

[4] (quotient -7 2)
-3

[5] (remainder -7 2)
-1

[6] (modulo -7 2)
1

[7] (modulo 7 3)
1

[8] (modulo 7 2)
1

[9] (round 7.5)
8

[10] (truncate 7.999)
7

[11] (floor 3.14)
3

[12] (ceiling 3.14)
4

[13] (floor -3.14)
-4

[14] (ceiling -3.14)
-3

[15] (log (exp 1))
1.

[16] (sqrt 1024)
32.

[18] (gcd 11 33)
11

[19] (lcm 11 33)
33
```

```
[20] (expt 2 3)
8

[21] (expt 2.01 3.01)
8.17749209238799

[22] pi
3.14159265358979

[23] (sin (/ pi 4))
0.707106781186548

[24] (cos (/ pi 4))
0.707106781186548

[25] (tan (/ pi 4))
1.

[26] (- (/ pi 4) (atan 1))
0.

[27] (* 2 (asin 1))
3.14159265358979

[28] (acos (cos 1.23456789))
1.23456789

[29] (max 1.1 2.2 3.3 2.2 1.1)
3.3

[30] (min 4.4 3.3 2.2 1.1)
1.1

[26] (gcd 11 13)
1

[27] (gcd 11 33)
11

[28] (lcm 4 3)
12

[29] (lcm 40 30)
120

[30] (lcm 45 30)
90

[31] (gcd -11 -33)
11
[32] (gcd 11 -33)
11
```

```
;;;;;;;;;;;;;
;; INSPECT ;;
;;;;;;;;;;;;;

[52] (inspect)

[Inspect] ?
   ?     -- display this command summary
   !     -- reinitialize INSPECT
 ctrl-A -- display All environment frame bindings
 ctrl-B -- display procedure call Backtrace
 ctrl-C -- display Current environment frame bindings
 ctrl-D -- move Down to callee's stack frame
 ctrl-E -- Edit variable binding
 ctrl-G -- Go  (resume execution)
 ctrl-I -- evaluate one expression and Inspect the result
 ctrl-L -- List current procedure
 ctrl-M -- repeat the breakpoint Message
 ctrl-P -- move to Parent environment's frame
 ctrl-Q -- Quit  (RESET to top level)
 ctrl-R -- Return from BREAK with a value
 ctrl-S -- move to Son environment's frame
 ctrl-U -- move Up to caller's stack frame
 ctrl-V -- eValuate one expression in current environment
 ctrl-W -- (Where) Display current stack frame
To enter `ctrl-A', press both `CTRL' and `A'.


[Inspect] All   ;;;;;;; Ctrl-A
Environment frame bindings at level 0  (USER-INITIAL-ENVIRONMENT)
    A                    #!UNASSIGNED
    X                    123
    Y                    "string"
    Z                    123


[Inspect] Backtrace calls    ;;;;; Ctrl-B
Stack frame for ()
  called from   ()
  called from   ()
  called from   #<PROCEDURE EVAL>
  called from   #<PROCEDURE ==SCHEME-RESET==>


[Inspect] Current environment frame   ;;;;;; Ctrl-C
Environment frame bindings at level 0  (USER-INITIAL-ENVIRONMENT)
    A                    #!UNASSIGNED
    X                    123
    Y                    "string"
    Z                    123


[Inspect] `Parent' environment frame   ;;;;;  Ctrl-P
```

```
Environment frame bindings at level 1  (USER-GLOBAL-ENVIRONMENT)
    BREAKPOINT-PROCEDURE  #<PROCEDURE BREAKPOINT-PROCEDURE>
    QUOTIENT              #<PROCEDURE QUOTIENT>
    PCS-GC-RESET          ()
    OPEN-INPUT-STRING     #<PROCEDURE OPEN-INPUT-STRING>
    FAST-LOAD             #<PROCEDURE FAST-LOAD>
    PCS-KILL-ENGINE       #<PROCEDURE PCS-KILL-ENGINE>
    CAR                   #<PROCEDURE CAR>
    ABS                   #<PROCEDURE ABS>
    PCS-BINARY-OUTPUT     #T
    EOF                   #!EOF
    USER-GLOBAL-ENVIRONMENT
                          #<ENVIRONMENT>
    PCS-PRIMOP-APPEND*    #<PROCEDURE PCS-PRIMOP-APPEND*>
    LCM                   #<PROCEDURE LCM>
    PCS-STATUS-WINDOW     #<PORT>
    COMPILE-FILE          #<PROCEDURE COMPILE-FILE>
    SET-FILE-POSITION!    #<PROCEDURE SET-FILE-POSITION!>
    MAP                   #<PROCEDURE MAP>
    ENVIRONMENT-BINDINGS  #<PROCEDURE ENVIRONMENT-BINDINGS>
    MAPCAR                #<PROCEDURE MAP>
    EXACT?                #<PROCEDURE EXACT?>
    LOG                   #<PROCEDURE LOG>
    NIL                   ()
    TAN                   #<PROCEDURE TAN>
    PPEEP=                ()
    COS                   #<PROCEDURE COS>
    EXPAND-MACRO          #<PROCEDURE EXPAND-MACRO>
    %PCS-STL-DEBUG-FLAG   ()
    CLOSE-OUTPUT-PORT     #<PROCEDURE CLOSE-OUTPUT-PORT>
    AUTOLOAD-FROM-FILE    #<PROCEDURE AUTOLOAD-FROM-FILE>
    STRING<=?             #<PROCEDURE STRING<=?>
    %ERROR-INVALID-OPERAND-LIST
                          #<PROCEDURE %ERROR-INVALID-OPERAND-LIST>
    SUBSTRING-FILL!       #<PROCEDURE SUBSTRING-FILL!>
    SIN                   #<PROCEDURE SIN>
    STRING>=?             #<PROCEDURE STRING>=?>
    OPTIMIZE!             #<PROCEDURE OPTIMIZE!>
    EXPAND                #<PROCEDURE EXPAND>
    UNBIND                #<PROCEDURE UNBIND>
    EXP                   #<PROCEDURE EXP>
    PCS-PRINCODE          #<PROCEDURE PCS-PRINCODE>
    RANDOM                #<PROCEDURE RANDOM>
    OPEN-INPUT-FILE       #<PROCEDURE OPEN-INPUT-FILE>
    REMAINDER             #<PROCEDURE REMAINDER>
    FREESP                #<PROCEDURE FREESP>
    PSIMP=                ()
    FILE-EXISTS?          #<PROCEDURE FILE-EXISTS?>
    LIST->STREAM          #<PROCEDURE LIST->STREAM>
    STREAM->LIST          #<PROCEDURE STREAM->LIST>
    PCS-CHK-ID            #<PROCEDURE PCS-CHK-ID>
    PCS-ENGINE-TIMEOUT    #<PROCEDURE PCS-ENGINE-TIMEOUT>
    CALL-WITH-OUTPUT-FILE #<PROCEDURE CALL-WITH-OUTPUT-FILE>
```

```
SUBSTRING-MOVE-RIGHT! #<PROCEDURE SUBSTRING-MOVE-RIGHT!>
DOUBLE-SLASHIFY       #<PROCEDURE DOUBLE-SLASHIFY>
GENSYM                #<PROCEDURE GENSYM>
%XLI-DEBUG            #<PROCEDURE %XLI-DEBUG>
DELETE!               #<PROCEDURE DELETE!>
LIST->VECTOR          #<PROCEDURE LIST->VECTOR>
VECTOR->LIST          #<PROCEDURE VECTOR->LIST>
PCS-FAIL-K            ()
RANDOMIZE             #<PROCEDURE RANDOMIZE>
*THE-NON-PRINTING-OBJECT*
                      #!UNPRINTABLE
SET-LINE-LENGTH!      #<PROCEDURE SET-LINE-LENGTH!>
STRING-NULL?          #<PROCEDURE STRING-NULL?>
STRING->LIST          #<PROCEDURE STRING->LIST>
LIST->STRING          #<PROCEDURE LIST->STRING>
PCS-SIMPLIFY          #<PROCEDURE PCS-SIMPLIFY>
PCS-CHK-LENGTH>=      #<PROCEDURE PCS-CHK-LENGTH>=>
PCS-PRIMOP-LIST       #<PROCEDURE PCS-PRIMOP-LIST>
PCS-INITIAL-ARGUMENTS ()
HEAD                  #<PROCEDURE HEAD>
%PRETTY-PRINTER       #<PROCEDURE %PRETTY-PRINTER>
ERROR-PROCEDURE       #<PROCEDURE ERROR-PROCEDURE>
CREATE-SCHEME-MACRO   #<PROCEDURE CREATE-SCHEME-MACRO>
PCG=                  ()
CADR                  #<PROCEDURE CADR>
*IRRITANT*            -- list --
PCS-CHK-PAIRS         #<PROCEDURE PCS-CHK-PAIRS>
READ                  #<PROCEDURE READ>
PME=                  ()
LOAD                  #<PROCEDURE LOAD>
PCS-NULL-K            #<PROCEDURE PCS-NULL-K>
MAPC                  #<PROCEDURE FOR-EACH>
STANDARD-OUTPUT       CONSOLE
PCS-VERBOSE-FLAG      ()
ATAN                  #<PROCEDURE ATAN>
SYNTAX-ERROR          #<PROCEDURE SYNTAX-ERROR>
PCS-MACRO-EXPAND      #<PROCEDURE PCS-MACRO-EXPAND>
ACOS                  #<PROCEDURE ACOS>
T                     #T
%ERROR-INVALID-OPERAND
                      #<PROCEDURE %ERROR-INVALID-OPERAND>
FRESH-LINE            #<PROCEDURE FRESH-LINE>
EVAL                  #<PROCEDURE EVAL>
OPEN-EXTEND-FILE      #<PROCEDURE OPEN-EXTEND-FILE>
CEILING               #<PROCEDURE CEILING>
OUTPUT-PORT?          #<PROCEDURE OUTPUT-PORT?>
%COMPILE-TIMINGS      ()
PCS-INTEGRATE-PRIMITIVES
                      #T
TAIL                  #<PROCEDURE TAIL>
PCS-MACHINE-TYPE      252
PCS-LOCAL-VAR-COUNT   0
PCS-SUCCESS-K         #<PROCEDURE PCS-NULL-K>
```

```
ASIN                        #<PROCEDURE ASIN>
ASCII->SYMBOL               #<PROCEDURE ASCII->SYMBOL>
SYMBOL->ASCII               #<PROCEDURE SYMBOL->ASCII>
PCS-PRIMOP-LIST*            #<PROCEDURE PCS-PRIMOP-LIST*>
USER-INITIAL-ENVIRONMENT
                            #<ENVIRONMENT>
THAW                        #<PROCEDURE THAW>
COMPILE                     #<PROCEDURE COMPILE>
%INSPECTOR                  #<PROCEDURE %INSPECTOR>
IMPLODE                     #<PROCEDURE IMPLODE>
PCS-ASSEMBLER               #<PROCEDURE PCS-ASSEMBLER>
STREAM?                     #<PROCEDURE STREAM?>
RESET-SCHEME-TOP-LEVEL
                            #<PROCEDURE RESET-SCHEME-TOP-LEVEL>
PROCEDURE-ENVIRONMENT       #<PROCEDURE PROCEDURE-ENVIRONMENT>
EXIT                        #<PROCEDURE EXIT>
%PCS-EDIT-BINDING           #<PROCEDURE %PCS-EDIT-BINDING>
%EXPAND-SYNTAX-FORM         #<PROCEDURE %EXPAND-SYNTAX-FORM>
COPY                        #<PROCEDURE COPY>
%C                          #<PROCEDURE %C>
OPEN-BINARY-INPUT-FILE
                            #<PROCEDURE OPEN-BINARY-INPUT-FILE>
%D                          #<PROCEDURE %D>
%PCS-STL-HISTORY            -- list --
EXPLODE                     #<PROCEDURE EXPLODE>
PCS-CLEAR-REGISTERS         #<PROCEDURE PCS-CLEAR-REGISTERS>
TRUE                        #T
EXPT                        #<PROCEDURE EXPT>
PCS-PRIMOP-MAKE-VECTOR
                            #<PROCEDURE PCS-PRIMOP-MAKE-VECTOR>
STRING-APPEND               #<PROCEDURE STRING-APPEND>
EXPAND-MACRO-1              #<PROCEDURE EXPAND-MACRO-1>
WINDOW?                     #<PROCEDURE WINDOW?>
PCS-MAKE-LABEL              #<PROCEDURE PCS-MAKE-LABEL>
DELQ!                       #<PROCEDURE DELQ!>
%HASH                       #<PROCEDURE %HASH>
ENGINE-RETURN               #<PROCEDURE ENGINE-RETURN>
THE-EMPTY-STREAM            -- vector --
SQRT                        #<PROCEDURE SQRT>
PCS-SYSDIR                  "C:\\lang\\SCHEMEti"
PCS-PERMIT-PEEP-1           #T
PCS-INTEGRATE-INTEGRABLES                  #T
PCS-PERMIT-PEEP-2           #T
RUNTIME                     #<PROCEDURE RUNTIME>
OPEN-OUTPUT-FILE            #<PROCEDURE OPEN-OUTPUT-FILE>
FOR-EACH                    #<PROCEDURE FOR-EACH>
WRITELN                     #<PROCEDURE WRITELN>
PCS-DEBUG-MODE              ()
PCS-CHK-BVL                 #<PROCEDURE PCS-CHK-BVL>
PCS-INTEGRATE-T-AND-NIL                    #T
PCS-GC-MESSAGE              ()
*USER-ERROR-HANDLER*        ()
EOF-OBJECT?                 #<PROCEDURE EOF-OBJECT?>
```

```
WITH-OUTPUT-TO-FILE   #<PROCEDURE WITH-OUTPUT-TO-FILE>
%SYSTEM-FILE-NAME     #<PROCEDURE %SYSTEM-FILE-NAME>
CLOSE-INPUT-PORT      #<PROCEDURE CLOSE-INPUT-PORT>
PCS-DEFINE-OPCODE     #<PROCEDURE PCS-DEFINE-OPCODE>
PCS-COMPILE-TO-AL     #<PROCEDURE PCS-COMPILE-TO-AL>
%COMPILE              #<PROCEDURE %COMPILE>
MAKE-ENGINE           #<PROCEDURE MAKE-ENGINE>
TRANSCRIPT-ON         #<PROCEDURE TRANSCRIPT-ON>
ASSERT-PROCEDURE      #<PROCEDURE ASSERT-PROCEDURE>
GC                    #<PROCEDURE GC>
PCS-GENCODE           #<PROCEDURE PCS-GENCODE>
GET-GC-COMPACT-COUNT  #<PROCEDURE GET-GC-COMPACT-COUNT>
PCS-PRIMOP-STD-N2     #<PROCEDURE PCS-PRIMOP-STD-N2>
%EXECUTE              #<PROCEDURE %EXECUTE>
STRING-CI<?           #<PROCEDURE STRING-COMPARISON>
STRING-CI=?           #<PROCEDURE STRING-COMPARISON>
PCS-EXECUTE-AL        #<PROCEDURE PCS-EXECUTE-AL>
%INSPECT              #<PROCEDURE %INSPECT>
FALSE                 ()
PCS-ERROR-FLAG        ()
BOOLEAN?              #<PROCEDURE BOOLEAN?>
PI                    3.14159265358979
PASM=                 ()
PCS-INTEGRATE-DEFINE  #T
FORCE                 #<PROCEDURE FORCE>
CALL-WITH-INPUT-FILE  #<PROCEDURE CALL-WITH-INPUT-FILE>
LINE-LENGTH           #<PROCEDURE LINE-LENGTH>
LIST-REF              #<PROCEDURE LIST-REF>
PP                    #<PROCEDURE PP>
PCS-AUTOLOAD-BINDING  #<PROCEDURE PCS-AUTOLOAD-BINDING>
PCS-DEFINE-PRIMOP     #<PROCEDURE PCS-DEFINE-PRIMOP>
EDWIN                 #<PROCEDURE INITIATE-EDWIN>
INEXACT?              #<PROCEDURE INEXACT?>
SUBSTRING-CI<?        #<PROCEDURE MAKE-SUBSTRING<>
SUBSTRING-CI=?        #<PROCEDURE MAKE-SUBSTRING=>
SUBSTRING-MOVE-LEFT!  #<PROCEDURE SUBSTRING-MOVE-LEFT!>
STRING<?              #<PROCEDURE STRING<?>
*ERROR-CODE*          1
STRING=?              #<PROCEDURE STRING=?>
STRING>?              #<PROCEDURE STRING>?>
INITIATE-EDWIN        #<PROCEDURE INITIATE-EDWIN>
FLOOR                 #<PROCEDURE FLOOR>
GET-FILE-POSITION     #<PROCEDURE GET-FILE-POSITION>
%PP-ME                #<PROCEDURE %PP-ME>
WITH-INPUT-FROM-FILE  #<PROCEDURE WITH-INPUT-FROM-FILE>
ROUND                 #<PROCEDURE ROUND>
PCS-POSTGEN           #<PROCEDURE PCS-POSTGEN>
REMOVE-AUTOLOAD-INFO  #<PROCEDURE REMOVE-AUTOLOAD-INFO>
SET-GC-COMPACT-COUNT! #<PROCEDURE SET-GC-COMPACT-COUNT!>
*ERROR-HANDLER*       #<PROCEDURE *ERROR-HANDLER*>
TRUNCATE              #<PROCEDURE TRUNCATE>
%DELAY                #<PROCEDURE %DELAY>
PCS-PRIMOP-IO-1       #<PROCEDURE PCS-PRIMOP-IO-1>
```

```
    STANDARD-INPUT          CONSOLE
    PCS-PRIMOP-IO-2         #<PROCEDURE PCS-PRIMOP-IO-2>
    SUBSTRING<?             #<PROCEDURE MAKE-SUBSTRING<>
    *ERROR-MESSAGE*
"Attempt to call a non-procedural object with  1 argument(s) as follows:"
    SUBSTRING=?             #<PROCEDURE MAKE-SUBSTRING=>
    FLUSH-INPUT             #<PROCEDURE FLUSH-INPUT>
    TRANSCRIPT-OFF          #<PROCEDURE TRANSCRIPT-OFF>
    STRING-COPY             #<PROCEDURE STRING-COPY>
    PCS-CHK-BVAR            #<PROCEDURE PCS-CHK-BVAR>
    INPUT-PORT?             #<PROCEDURE INPUT-PORT?>
    PCS-PRIMOP-*            #<PROCEDURE PCS-PRIMOP-*>
    PCS-CHK-LENGTH=         #<PROCEDURE PCS-CHK-LENGTH=>
    PCS-PRIMOP-+            #<PROCEDURE PCS-PRIMOP-+>
    OPEN-BINARY-OUTPUT-FILE
                            #<PROCEDURE OPEN-BINARY-OUTPUT-FILE>
    PCS-PRIMOP--            #<PROCEDURE PCS-PRIMOP-->
    PCS-PRIMOP-VECTOR       #<PROCEDURE PCS-PRIMOP-VECTOR>
    PCS-DISPLAY-WARNINGS    #T
    PCS-PRIMOP-/            #<PROCEDURE PCS-PRIMOP-/>
    GCD                     #<PROCEDURE GCD>
    PCS-CLOSURE-ANALYSIS    #<PROCEDURE PCS-CLOSURE-ANALYSIS>
    DELAYED-OBJECT?         #<PROCEDURE DELAYED-OBJECT?>
    CURRENT-COLUMN          #<PROCEDURE CURRENT-COLUMN>
[Inspect] Quit
[36] (exit)

;-----------------------------------------------
;  Hyperbolic functions (file hyper.s)
;-----------------------------------------------

; In PC Scheme pretty-printing is associated with
; debugging. To enable pp we must first enable the
; debug mode:

(set! pcs-debug-mode #t)

; Now we can insert function definitions:

   (define (ch x)  (* 0.5 (+ (exp x) (exp (- x)))))
   (define (sh x)  (* 0.5 (- (exp x) (exp (- x)))))
   (define (th x)  (/ (sh x) (ch x)))
   (define (cth x) (/ (ch x) (sh x)))

(set! pcs-debug-mode #f)  ; debug mode is no longer needed

(pp ch)  (newline)        ; pretty-print
(pp sh)  (newline)        ; each function
(pp th)  (newline)        ; during the
(pp cth) (newline)        ; load process
```

```
------------------------------------------------
PROGRAM :  Pretty-Print
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 13, 1994
------------------------------------------------
[3] (load "hyper.s")
#<PROCEDURE CH> =
(LAMBDA (X)
  (* 0.5 (+ (EXP X) (EXP (- X)))))
#<PROCEDURE SH> =
(LAMBDA (X)
  (* 0.5 (- (EXP X) (EXP (- X)))))
#<PROCEDURE TH> =
(LAMBDA (X)
  (/ (SH X) (CH X)))
#<PROCEDURE CTH> =
(LAMBDA (X)
  (/ (CH X) (SH X)))
OK

[4] (ch 0)
1.

[5] (sh 0)
0.

[6] (th 0)
0.

[7] (cth 0)

[VM ERROR encountered!] Divide by zero
(/ 1. 0.)


[Inspect] ?
   ?     -- display this command summary
   !     -- reinitialize INSPECT
 ctrl-A -- display All environment frame bindings
 ctrl-B -- display procedure call Backtrace
 ctrl-C -- display Current environment frame bindings
 ctrl-D -- move Down to callee's stack frame
 ctrl-E -- Edit variable binding
 ctrl-G -- Go  (resume execution)
 ctrl-I -- evaluate one expression and Inspect the result
 ctrl-L -- List current procedure
 ctrl-M -- repeat the breakpoint Message
 ctrl-P -- move to Parent environment's frame
 ctrl-Q -- Quit  (RESET to top level)
 ctrl-R -- Return from BREAK with a value
 ctrl-S -- move to Son environment's frame
 ctrl-U -- move Up to caller's stack frame
 ctrl-V -- eValuate one expression in current environment
```

```
 ctrl-W -- (Where) Display current stack frame
To enter `ctrl-A', press both `CTRL' and `A'.

[Inspect] Quit          ;;;;;;;  Ctrl-Q
[8] (exit)
```

EQUIVALENCE PREDICATES

(eq? <obj1> <obj2>) — True if <obj1> is pointer-identical to <obj2>, i.e. bound to the same memory location (this test of physical sameness can be implementation-dependent)

(equal? <obj1> <obj2>) — True if <obj1> and <obj2> have the same type and value (this test causes evaluation of expr.)

(eqv? <obj1> <obj2>) — True if <obj1> and <obj2> are equal numeric values (same or dif. type), strings, and other atoms

```
------------------------------------------------------------
PROGRAM :  Equivalence Predicates (=  eq?  equal?  eqv?)
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 19, 1994
------------------------------------------------------------

[5] (define one 1)
ONE

[6] (define two 2)
TWO

[7] (define lst '(1 2 3))
LST

[8] (define pair (cons 'a 'b))
PAIR

[13] (define str "string")
STR

[14] (define sym 'symbol)
SYM

[15] str
"string"
```

```
[16] sym
SYMBOL

[9] one
1

[10] two
2

[11] lst
(1 2 3)

[12] pair
(A . B)


[17] ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (= one two)   ; = is used for numerical comparison only
()

[18] (= 2 two)
#T

[19] (= 3 (+ one two))
#T

[20] (= "123" "123")

[VM ERROR encountered!] Non-numeric operand to arithmetic operation
(=? "123" "123")

[Inspect] Quit

[21] (= #\7 #\7)          ;  #\7  is a character, same as #\a

[VM ERROR encountered!] Non-numeric operand to arithmetic operation
(=? #\7 #\7)

[Inspect] Quit


[22] ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (equal? str "string"); equal? causes evaluation of expressions
                      ; and can be time consuming; Rule of thumb:
                      ; equal? succeeds for objects that have the
                      ; same printed representation
#T

[23] (equal? sym 'symbol)
#T
```

```
[25] (equal? lst '(1 2 3))
#T

[26] (equal? pair (cons 'a 'b))
#T

[27] (equal? one 1)
#T
[29] (equal? (+ one 1) two)
#T

[30] (equal? str 1)
()

[31] (equal? str sym)
()

[32] (equal? lst (reverse '(3 2 1)))
#T

[33] (equal? #\7 #\7)
#T

[34] (define fun1 (lambda (n) (* n n)))
FUN1

[35] (define fun2 (lambda (x) (* x x)))
FUN2
[36] (equal? fun1 fun2)
()

[37] (define fun3 (lambda (n) (* n n)))
FUN3

[38] (equal? fun1 fun3) ; fun1 and fun3 do not reference the same
                        ; structure
()

[39] (equal? fun1 fun1)
#T

[40] (= 1 1.0)
#T

[41] (equal? 1 1.0)
()

[42] ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

 (eq? 1 1.0) ; eq? is a restrictive pointer-based equivalence
()
```

```
[43] (eqv? 1 1.0)
#T

[44] (equal? #\A 65)
()

[46] (equal? #\A (integer->char 65))
#T

[47] lst
(1 2 3)

[48] (eq? lst '(1 2 3))
()

[49] (eq? 1 1)
#T

[50] (eq? '(1 2 3) '(1 2 3))
()

[51] pair
(A . B)

[52] (eq? (cons 1 2) (cons 1 2))
()

[53] (eq? (cons 'a 'b) pair)
()

[54] (define p pair)
P

[55] (eq? p pair)
#T

[56] (eq? fun1 fun3)
()

[57] (define fun3 fun1)
FUN3

[58] (eq? fun1 fun3)
#T

[59] (eq? 'x 'x)
#T

[60] (car lst)
1

[61] (eq? 1 (car lst))
```

```
#T

[62] (cdr pair)
B

[63] (eq? (cdr pair) 'b)
#T

[64]                ; eq? is based on pointer comparison (i.e. it is very efficient)
(eq? "a" "a")
()

[65] (eq? #\a #\a)
#T

[66] (eq? () ())
#T

[67] (eq? "" "")  ; null string comparison
()

[68] ;eq? may behave differently from one implementation to another
   ; causing portability problems (i.e. eq? is a low-level function
   ; testing physical equivalence of data structures)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(eqv? "" "")
#T

[69] lst
(1 2 3)

[70] (eqv? lst '(1 2 3))
()

[71] (define lst2 lst)
LST2

[72] (eqv? lst2 lst)
#T

[73] (eqv? lst2 '(1 2 3)) ; this is not pointer equivalent
()

[74] (eqv? "" "")
#T

[75] (eqv? '(a) '(a))
()

[76] (eqv? sym 'symbol)
#T
```

```
[77] (eqv? #\a #\a)
#T

[78] (eqv? 123 123)
#T

[79] (eqv? '() #f)
#T

[80] (eqv? 1 1.0)
#T

[81] (eqv? (/ 2 2) 1.0)
#T

[82] (exit)

------------------------------------------------------------
PROGRAM :  EQUIVALENCES - Comparison of eq?, eqv?, equal?
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 25, 1994
------------------------------------------------------------
[66] (pp e)
#<PROCEDURE E> =
(LAMBDA (X Y)
  (DISPLAY (EQ? X Y))
  (DISPLAY (EQV? X Y))
  (DISPLAY (EQUAL? X Y)))
[67] (pp f)
#<PROCEDURE F> =
(LAMBDA (K)
  (* K K))

[68] n
1

[69] r
1.2

[70] b
#T

[71] c
#\a

[72] s
SYM

[73] st
"str"

[74] (define p (cons 'a 'b))
P
```

```
[75] p
(A . B)

[77] '(a . b)
(A . B)

[78] lst
(1 2 3)

[79] v
#(1 2 3)

[80] (e 1 1)
#T#T#T

[81] (e 1 n)
#T#T#T

[82] (e n n)
#T#T#T

[83] (e 1.2 1.2)
()#T#T

[84] (e 1.2 r)
()#T#T

[85] (e r r)
#T#T#T

[86] (e #T #T)
#T#T#T

[87] (e #T b)
#T#T#T

[88] (e b b)
#T#T#T

[89] (e #\a #\a)
#T#T#T

[90] (e #\a c)
#T#T#T

[91] (e c c)
#T#T#T

[92] (e 'sym 'sym)
#T#T#T

[93] (e 'sym s)
```

```
#T#T#T

[94] (e s s)
#T#T#T

[96] (e "str" "str")
()#T#T

[97] (e "str" st)
()#T#T

[98] (e st st)
#T#T#T

[99] (e '(a . b) '(a . b))
()()#T

[100] (e '(a . b) p)
()()#T

[101] (e p p)
#T#T#T

[103] (e '(a . b) (cons 'a 'b))
()()#T

[104] (e '(1 2 3) '(1 2 3))
()()#T

[105] (e '(1 2 3) lst)
()()#T

[106] (e lst lst)
#T#T#T

[107] (e '#(1 2 3) '#(1 2 3))
()()#T

[108] (e '#(1 2 3) v)
()()#T

[109] (e v v)
#T#T#T

[110] (e 1 1.0)
()#T()

[111] (= 1 1.0)
#T

[112] (equal? 1 1.0)
()
```

```
[113] (e 0 0.0)
()#T()

[114] (e 2 (/ 4 2))
#T#T#T

[115] (e f f)
#T#T#T

[116] (e (f 2) (f 2))
#T#T#T

[117] (e (f 2) 4)
#T#T#T

[118] (e (f 2) 4.0)
()#T()

[119] (e (/ 8 2) 4.0)
()#T()

[120] (e (/ 8.0 2.0) 4.0)
()#T#T

[121] (e "" "")
()#T#T

[122] (e '() #F)
#T#T#T

[123] (e () #F)
#T#T#T

[124] (e '(()) #F)
()()()

[125] (e () (not #T))
#T#T#T

[126] (e #T 1)
()()()

[127] (not 1)
()

[128] (e () (not 1))
#T#T#T

[129] (e () (not "str"))
#T#T#T

[130] (and 1 1)
1
```

```
[131] (or 1 1)
1

[132] (e (and 1 1) (or 1 1))
#T#T#T

[133] (e 1 (and 1 1))
#T#T#T

[134] (e (reverse '(3 2 1)) lst)
()()#T

[135] (e f (lambda(k) (*k k)))
()()()

[137] (e f (lambda(k) (* k k)))
()()()

[138] u
#!UNASSIGNED

[140] (* 8 (atan 1))
6.28318530717959

[141] (* 4 (atan 1))
3.14159265358979

[143] (e pi (* 4 (atan 1)))
()#T#T

[144] (exit)
```

CONTROL STRUCTURES

(begin <expr1> <expr2> ...) — Evaluate all expressions and return the value of the last expression in the sequence.

```
[53] x
123
[54] (begin (set! x 5) (1+ x))  ; 1+ is increment operator
6
[55] (begin (set! x 7) (+ 1 x))
8
```

(begin0 <expr1> <expr2> ...) — Evaluate all expressions and return the value of <expr1>

(if <condition> <expr1> <expr2>)

Conditional execution: it returns the value of <expr1> in cases where <condition>=#T. If <condition>=#F then the returned value is <expr2>; in cases where <expr2> is not specified the result of **if** expression is unspecified (some implementations, including pcs, return **( )**).

```
[58] (define one 1)
ONE
[59] (define seven 7)
SEVEN
[60] one
1
[61] seven
7
[62] (if (> one seven) 'yes  'no)
NO
[63] (if (= seven (+ 3 4)) 'yes 'no)
YES
```

(cond
(<test1> <expr11> <expr12> ...)
(<test2> <expr21> <expr22> ...)
.....
(<testN> <exprN1> <exprN2> ...)
(else <expr1> <expr2> ...)
)

Conditional execution: evaluate tests and execute either the sequence of expressions following the first ... that is nonnull, or the expressions in the last line. It returns the value of the last executed expression. Else clause can be omitted (but in such a case the result of **cond** for all false tests is unspecified.)

```
[5] (define x 3)
X
[6] (define y 4)
Y
[7] (cond ((> x y) 'greater)
          ((< x y) 'less)
          (else 'equal))
LESS
[8] (set! y 3)
3
[9] (cond ((> x y) 'greater)
          ((< x y) 'less)
          (else 'equal))
EQUAL
```

(case <expr>
(<atom1> <expr11> <expr12> ...)
(<atom2> <expr21> <expr22> ...)
. . . . . .
(<atomN> <exprN1> <exprN2> ...)
(else <expr1> <expr2> ...))

Conditional execution: evaluate <expr> and compare its value with <atom1>, <atom2>, ... until the eqv? comparison returns true. Evaluate all expressions in the selected list and return the value of the last executed expression.

```
[22] (define test
 (lambda (x)
   (case x
         (1 'odd)
         (3 'odd)
         (5 'odd)
         (7 'odd)
         (9 'odd)
         (0 'even)
         (2 'even)
         (4 'even)
         (6 'even)
         (8 'even)
         (else 'not_from_0_to_9))))
TEST
[23] (test 5)
ODD
[24] (test 6)
EVEN
[25] (test 12)
NOT_FROM_0_TO_9
```

(case <expr>
(<atom-list1> <expr11> <expr12> ...)
(<atom-list2> <expr21> <expr22> ...)
. . . . . .
(<atom-listN> <exprN1> <exprN2> ...)
(else <expr1> <expr2> ...))

Conditional execution: evaluate <expr> and compare its value with <atom-list1>, <atom-list2>, ... until the membership is found. Evaluate all expressions in the selected list and return the value of the last executed expression.

```
[11] (set! pcs-debug-mode #t)
#T
[12] (define test    ;  bad indenting
 (lambda (x)
   (case x
 ((1 3 5 7 9) 'odd)
 ((0 2 4 6 8) 'even)
 (else 'not_from_0_to_9))))
TEST
[13] (test 7)
ODD
[14] (test 4)
EVEN
```

```
[15] (test 123)
NOT_FROM_0_TO_9
[16] (pp test)
#<PROCEDURE TEST> =
(LAMBDA (X)
  (CASE X
    ((1 3 5 7 9) 'ODD)
    ((0 2 4 6 8) 'EVEN)
    (ELSE 'NOT_FROM_0_TO_9)))
```

DO LOOP

(do ((<variable> <initial-value> <update>) ...)
(<termination-test> <expression> ...)
<statement> ...)

Initial values are bound to corresponding variables in unspecified order. If the termination test fails the sequence of statements (the body of the loop) is executed. After the execution the do variables are updated in unspecified order. The update is optional. If the termination-test is true then expressions are evaluated from first to last and the last value is returned (if expressions are not specified then the returned value of do loop is the value of the <termination-test>).

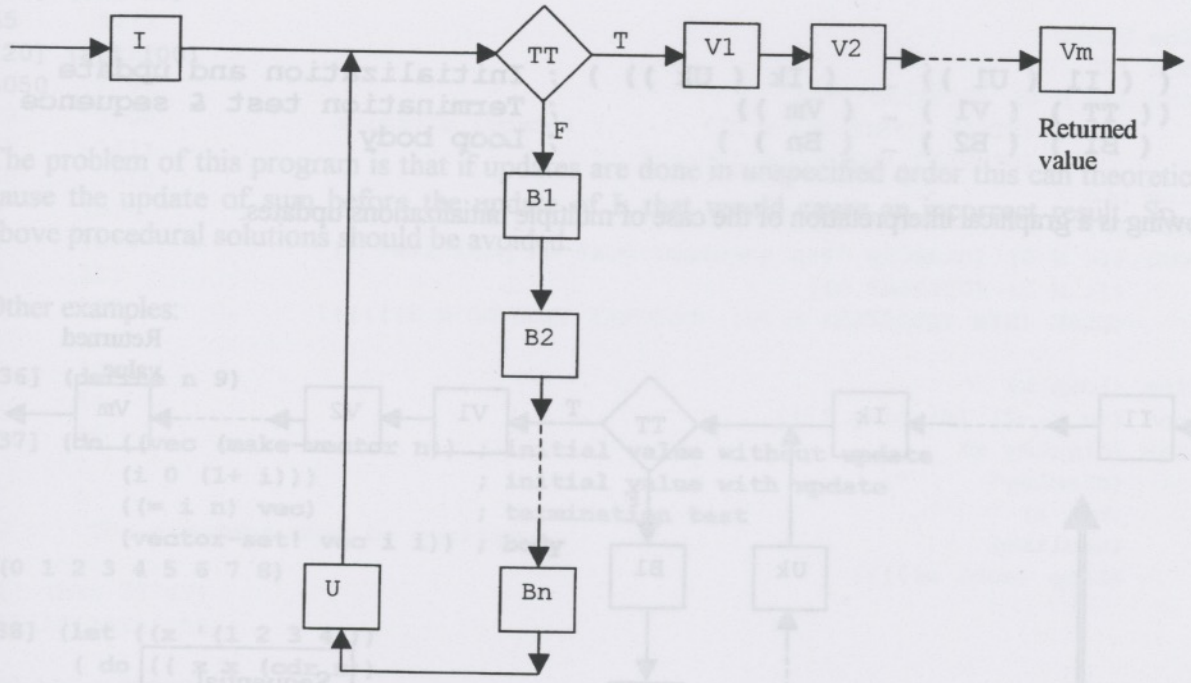
Do loop in Scheme is different from do loops in FORTRAN, Pascal, and C in the following:

- The number of initializations and updates can be one or more.
- Initializations and updates are performed in unspecified order
- After the termination test it is possible to perform a sequence of expressions.

Do loop with single initialization/update

```
(do ( ( I   ( U )) )                ; Initialization and update
    (( TT ) ( V1 ) … ( Vm ))        ; Termination test & sequence
     ( B1 ) ( B2 ) … ( Bn ) )       ; Loop body
```

Following is a graphical interpretation of the case of single variable:




```
[11] ; Simplified syntax of do loop:
     ; (do (init and update variables)
     ;     (termination test and statements)
     ;     body )

(do ((k 1 (+ k 1)))
    ((> k 5) (display ""))
    (display k) (display " "))
1 2 3 4 5
```

The loop body and update are performed sequentially. Therefore, it is possible to move a (short) loop body inside the update. To illustrate problems related to this, consider the following function that computes 1 + 2 + … + n = n(n+1)/2. A program that is written in the Scheme style can be based on the following recursive relation:

$$sum(n) = 1 + 2 + \ldots + (n-1) + n = sum(n-1) + n$$

This relation yields the following recursive program:

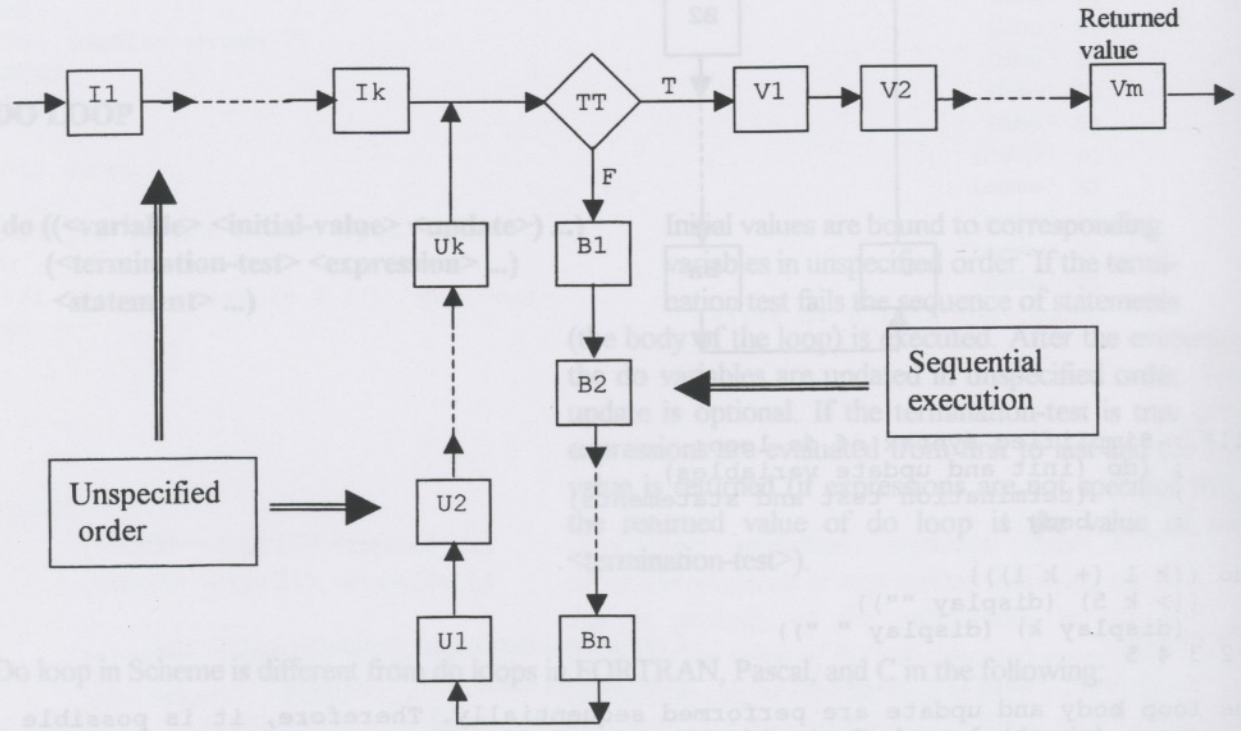
```
[21] (define (ari n)
       (cond (( <= n 0)  0)
          (else  (+ n (ari (- n 1))))))
ARI
[22] (ari 10)
55
[23] (ari 100)
5050
```

The same program can be written using the do loop, but the corresponding solutions are not elegant. They use the do loop with multiple initializations/updates and are presented below.

Do loop with multiple initializations/updates

```
(do ( ( I1 ( U1 )) …  ( Ik ( Uk )) ) ; Initialization and update
    (( TT ) ( V1 ) … ( Vm ))         ; Termination test & sequence
     ( B1 ) ( B2 ) … ( Bn ) )        ; Loop body
```

Following is a graphical interpretation of the case of multiple initializations/updates:




```
[13] (define (ari N)
  (DO ((I 1 (+ I 1)) (SUM 0))  ; Init. and updates of variables
      ((> I N) SUM)            ; Termination test and expression
    (SET! SUM (+ SUM I))))     ; Loop body
ARI
[14] (ari 10)
55
[15] (ari 100)
5050
```

A body-less version of this program is

```
[18] (define (ari N)
  (DO ((I 1 (+ I 1)) (SUM 0 (+ SUM I)))  ; Inits and updates
      ((> I N) SUM)))                    ; Term. test & expression
```

```
ARI
[19] (ari 10)
55
[20] (ari 100)
5050
```

The problem of this program is that if updates are done in unspecified order this can theoretically cause the update of sum before the update of I; that would cause an incorrect result. So, the above procedural solutions should be avoided.

Other examples:

```
[36] (define n 9)
N
[37] (do ((vec (make-vector n)) ; initial value without update
          (i 0 (1+ i)))         ; initial value with update
          ((= i n) vec)         ; termination test
          (vector-set! vec i i)) ; body
#(0 1 2 3 4 5 6 7 8)

[38] (let ((x '(1 2 3 4)))
      ( do (( x x (cdr x))
            (sum 0 (+ sum (car x))))
            ((null? x) sum)))
10

[39] ( do ((x '(1 2 3 4) (cdr x))  ; initial value with update
           (sum 0 (+ sum (car x)))); initial value with update
           ((null? x) sum)  )      ; termination test
10

(exit)
```

Close the transcript file and return to the level of the operating system.

```
Decimal to binary conversion (file  bin.s)

(define binar
(LAMBDA (N)
  (COND ((= N 0) (DISPLAY ""))
        (ELSE (BINAR (QUOTIENT N 2))       ; Integer division
              (DISPLAY (MODULO N 2))))))
(define bin
(LAMBDA (N)
  (COND ((< N 0) (display "The argument must be positive!"))
        ((< N 2) (DISPLAY n))
        (ELSE (BIN (QUOTIENT N 2)) (DISPLAY (MODULO N 2))))))

(define (loop x)
  (cond ((= x -2) (display ""))
  (else (display x)
        (display"      ")
        (bin x)
        (newline)
        (loop (sub1 x)))))



-------------------------------------------------
PROGRAM :  Dec/Bin Conversion
AUTHOR  :  Jozo J. Dujmovic
DATE    :  94/12/27
-------------------------------------------------
[3] (load "bin.s")
OK

[4] (pp bin)
#<PROCEDURE BIN> =
(LAMBDA (N)
  (COND ((< N 0)
         (DISPLAY "The argument must be positive!"))
        ((< N 2) (DISPLAY N))
        (ELSE (BIN (QUOTIENT N 2))
              (DISPLAY (MODULO N 2)))))


[5] (pp loop)
#<PROCEDURE LOOP> =
(LAMBDA (X)
  (COND ((= X -2) (DISPLAY ""))
        (ELSE (DISPLAY X)
              (DISPLAY "      ")
              (BIN X)
              (NEWLINE)
              (LOOP (SUB1 X)))))
```

```
[6] (loop 16)
16      10000
15       1111
14       1110
13       1101
12       1100
11       1011
10       1010
9       1001
8       1000
7       111
6       110
5       101
4       100
3       11
2       10
1       1
0       0
-1      The argument must be positive!

[8] (bin 32767)
111111111111111

[9] (bin 32768)
1000000000000000

#<PROCEDURE BINAR> =
(LAMBDA (N)
  (COND ((= N 0) (DISPLAY ""))
        (ELSE (BINAR (QUOTIENT N 2))
              (DISPLAY (MODULO N 2)))))

[9] (binar 0)
[10] (binar 1)
1

[11] (binar 7)
111

[12] (binar 32767)
111111111111111

[13] (binar 32768)
1000000000000000

[14] (binar 9999999999)
1001010100000010111110001111111111

[16] (bin 9999999999)
1001010100000010111110001111111111

[17] (exit)
```

```
;-----------------------------------------------
; PROGRAM :  Maximum of F(x)
; AUTHOR  :  Jozo J. Dujmovic
; DATE    :  10/28/94
;-----------------------------------------------

; The trisection method for finding the maximum of f(x) within the
; interval [x1, x2] starts by dividing the interval in three equal
; segments using the middle points a1 = x1 + (x2 - x1)/3, and a2 =
; x2 - (x2 - x1)/3. If f(a1) < f(a2) then a1 becomes the new value
; of x1; otherwise a2 becomes the new value of x2. Each such a step
; reduces the size od the [x1, x2] interval and this process terminates
; when the interval becomes sufficiently small. Then the coordinate
; of the maximum is xmax = (x1+x2)/2, yielding fmax = f(xmax).

(define disp
  (lambda (x n)
   (display (/ (round (* x (expt 10 n))) (expt 10 n)))))

(define fmax
 (lambda(f x1 x2)
  (cond
   ((< (- x2 x1) 1e-10)
       (display "Maximum:  f(")
       (disp (/ (+ x1 x2) 2) 4)
       (display ") = ")
       (disp (f (/ (+ x1 x2) 2)) 4))
   (else (let ((a1 (+ x1 (/ (- x2 x1) 3)))
               (a2 (- x2 (/ (- x2 x1) 3))))
           (if (< (f a1) (f a2))
               (fmax f a1 x2)
               (fmax f x1 a2)))))))

[2] (pp disp)
#<PROCEDURE DISP> =
(LAMBDA (X N)
  (DISPLAY
     (/ (ROUND (* X (EXPT 10 N))) (EXPT 10 N))))
[3] (pp fmax)
#<PROCEDURE FMAX> =
(LAMBDA (F X1 X2)
  (COND ((< (- X2 X1) 1.e-10)
         (DISPLAY "Maximum:  f(")
         (DISP (/ (+ X1 X2) 2) 4)
         (DISPLAY ") = ")
         (DISP (F (/ (+ X1 X2) 2)) 4))
        (ELSE (LET ((A1 (+ X1 (/ (- X2 X1) 3)))
                    (A2 (- X2 (/ (- X2 X1) 3))))
                (IF (< (F A1) (F A2))
                    (FMAX F A1 X2)
                    (FMAX F X1 A2))))))
```

```
[4] (fmax cos -1 1)
Maximum:  f(0) = 1

[5] (fmax (lambda(x) (+ x (/ 1 x))) 0.01 100)
Maximum:  f(100) = 100.01

[7] (fmax (lambda(x) (- (/ -1 x) x)) 0.01 100)
Maximum:  f(1) = -2

[8] (fmax (lambda(x) x) 0 10)
Maximum:  f(10) = 10

[9] (fmax (lambda(x) (minus x)) 0 10)
Maximum:  f(0) = 0

[10] (fmax (lambda(x) (* x (- 1 x))) 0 10)
Maximum:  f(0.5) = 0.25

[11] (fmax sqrt 0 1)
Maximum:  f(1) = 1

[3] (define (ff f g) (lambda(x) (f (g x))))
FF

[4] (fmax (ff minus abs) -1 1)
Maximum:  f(0) = 0

[5] (define (fff f g h) (lambda(x) (f (g (h x)))))
FFF

[7] (fmax (fff minus sqrt abs) -100 1000)
Maximum:  f(0) = 0

[1]
;---------------------------------------------------------------------
;   LISTS AND LIST OPERATIONS
;   List is a sequence of expressions enclosed in parantheses.
;   Strings of spaces are used as separators between the
;   expressions. Each list can be interpreted either as a list
;   of operands
;                  (<operand> <operand> ... <operand>)
;
;   or as an expression to be evaluated by applying the operator
;   (which must be the first element in the list) to operands
;   (which are the remaining elements of the list):
;
;                (<operator> <operand> ... <operand>)
;
;   Lists that are quoted are not evaluated as expressions:
;
;                '(<operator> <operand> ... <operand>)
;
```

```
;   Procedure eval evaluates the expression written in quoted
;   form:
;                (eval '(<operator> <operand> ... <operand>))
;
;   Each list, except the empty list, is a pair consisting of the
;   first element (head of list obtained by applying the function
;   car) and the remaining tail of list (obtained by applying the
;   function cdr). The empty list is a special object of its own
;   type: it contains no elements and its length is zero.
;----------------------------------------------------------------

'(1 2 3)                    ;  list of numbers
(1 2 3)

[2] ()                      ;  empty list
()
[3] '()                     ;  empty list
()
[4] (symbol? (1 2 3))       ;  without quote the list (1 2 3) is evaluated
                            ;  and the first element (1) is expected to be
                            ;  an operator
[ERROR encountered!] Attempt to call a non-procedural object
(1 2 3)

[Inspect] Quit

[5] (symbol? '(1 2 3))      ;  the list is not a symbol
()

[6] (length '(1 2 3))       ;  the number of elements in the list
3

[7] (length '(a b c d))     ;  list of symbols
4

[8] (length (* 2 3))        ;  the length of non-list is zero
0

[9] (reverse '(a b c d))
(D C B A)

[10] (length '())
0

[11] (length ())
0

[12] (null? '())            ;  function null? recognizes the empty list
#T

[13] (null? '(1 2 3))       ;  this list is NOT empty
()
```

```
[14] (list 1 2 '(1 2) '(a b))          ; function list makes the list of
                                       ;  its arguments
(1 2 (1 2) (A B))

[15] ; Expressions within a list may be atoms and/or lists.
     ; Lists can be evaluated by the Scheme evaluator as follows:
     ; 1. The first element is evaluated and must yield a function
     ; 2. Other elements are evaluated and their values are passed as
     ;    arguments to the function that is the value of the first
     ;    element
     ; 3. The value of the list is the result of applying the function
     ;    of the first element to the arguments (i.e. to the remaining
     ;    elements of the list)

(max 1 (+ 1 1) (* 2 3))
6

[16] ()              ; an empty list evaluates to itself
()

[17] '(1 2 3)        ; any quoted list evaluates to itself
(1 2 3)

[18] (1 2 3)         ; evaluation error (the first element is not
                     ; a function)

[ERROR encountered!] Attempt to call a non-procedural object
(1 2 3)
[Inspect] Quit

[19] ; Lists can be obtained as superposition of pairs, where pairs
     ; are denoted (a . b) and are constructed by the function cons:
(cons 'a nil)
(A)

[20] (cons 'a 'b)
(A . B)

[21] (cons 'a (cons 'b (cons 'c nil)))
(A B C)

[22] ; Instead of working with pointers lists can be directly defined:
'(a b c)
(A B C)

[23] ; Head of a list is obtained by applying the function car and the
     ; tail (which is always a list), by applying the function cdr:
(car '(a b c))
A

[24] (cdr '(a b c))
(B C)
```

```
[25] (car '(a . b)) ; car and cdr can also be applied to pairs
A

[26] (cdr '(a . b)) ; the pair is not a list and its cdr is not a list
B

[27] ; Relationship between car, cdr, and cons :
     ;  (cons (car List) (cdr List)) = List
     ;  (car (cons Head Tail)) = Head
     ;  (cdr (cons Head Tail)) = Tail
(cons (car '(a b c)) (cdr '(a b c)))
(A B C)

[28] (car (cons '(Head) '(t a i l)))
(HEAD)

[29] (cdr (cons '(Head) '(t a i l)))
(T A I L)

[33] (car '(1 2 3 4 5 6 7))
1

[34] (cdr '(1 2 3 4 5 6 7))
(2 3 4 5 6 7)

[35] (cons '(1) '(2 3 4 5 6 7))  ;  the head is a list
((1) 2 3 4 5 6 7)

[36] (cons '1 '(2 3 4 5 6 7))    ;  the head is an element
(1 2 3 4 5 6 7)

[37] (cadr '(1 2 3 4 5 6 7))     ;  cadr = car ( cdr .. )
2

[38] (caddr '(1 2 3 4 5 6 7))    ;  the "add" is evaluated right to left
3

[39] (cadddr '(1 2 3 4 5 6 7))
4

[40] (caddddr '(1 2 3 4 5 6 7)) ; the max length of "ad" operations is 4

[VM ERROR encountered!] Variable not defined in current environment
CADDDDR

[Inspect] Quit

[41] (cons '(1 2 3) '(4 5 6 7))
((1 2 3) 4 5 6 7)

[42] (car '( '(1 2 3) 4 5 6 7))
(QUOTE (1 2 3))
```

```
[43] (car '( (1 2 3) 4 5 6 7))
(1 2 3)

[44] (cdr '((1 2 3) 4 5 6 7))
(4 5 6 7)

[45] (caddr '((1 2 3) 4 5 6 7))
5

[48] (eval 6)
6

[49] (eval (* 2 3))
6

[50] (eval '(* 2 3))
6

[51] (eval ''(* 2 3))
(* 2 3)

[52] (eval (eval ''(* 2 3))))
6

[53] (eval (eval (eval '''(* 2 3))))
6

[54] (eval (eval (eval 6)))
6

[55] (length (eval '(* 2 3)))
0

[56] (length (eval ''(* 2 3)))
3

[57] (eval '())
()

[58] (eval '(3))
[ERROR encountered!] Attempt to call a non-procedural object
(3)

[Inspect] Quit

[59] (exit)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;                  DISPLAYING ELEMENTS OF A LIST
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;  WL function writes components of a list separated by a space:

[12] (define (wl lst)
       (cond
        ((null? lst) (display ""))
        (else (display (car lst)) (display " ") (wl (cdr lst)))))
WL

[13] (wl '())

[14] (wl '( 1 2 3 4))
1 2 3 4

[15] (wl '(a b c))     ; the components are interpreted as symbols
A B C

[16] (wl '( 'a 'b 'c))
(QUOTE A) (QUOTE B) (QUOTE C)

[17] (begin (wl '(1 2 3)) (wl '(a b c)))
1 2 3 A B C

[18] (wl '(asdfdfdafsd =))    ;  components are interpreted as symbols
ASDFDFDAFSD =

[19] (wl '("sadfdfdfs" =))
sadfdfdfs =

[20] (wl '((a b c) (d e f)))
(A B C) (D E F)

[21] (wl '( a + b = c 123))
A + B = C 123

; In the following WRITE function x is an argument that will be
; bound to a list of the actual arguments in the call

[23] (define w (lambda x (display x)))
W

[24] (w 1 2 3)
(1 2 3)

[25] (define w (lambda x (wl x)))
W

[26] (w 1 2 3)
```

```
1 2 3

[27] (w a + b = c)  ;  now the arguments are interpreted as variables
[VM ERROR encountered!] Variable not defined in current environment
A
[Inspect] Quit

[28] (w '(a + b = c))
(A + B = C)

[29] (w "a" "b" "c")
a b c

[30] (define a 1)
A

[31] (define b 2)
B

[32] (w a b)
1 2

[33] (w 'a 'b)
A B

[34] (pp writeln)
#<PROCEDURE WRITELN>

[35] (begin (w 'a 'b 'c) (w 1 2 3))
A B C 1 2 3

; An example of prompted input and the corresponding output:

[36] (begin (w "Enter an integer:") (define i (read)) (w "i =" i))
Enter an integer: 123
i = 123

[37] i
123

; Therefore, the following functions are useful for displaying the
; contents of lists:
[12] (pp wl)
#<PROCEDURE WL> =
(LAMBDA (LST)
  (COND ((NULL? LST) (DISPLAY ""))
        (ELSE (DISPLAY (CAR LST))
              (DISPLAY " ")
              (WL (CDR LST)))))
[13] (pp w)
#<PROCEDURE W> =
(LAMBDA LST
  (WL LST))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;           Making and correcting errors (a true story)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
[15] (pair? '(1 2))          ; Each nonempty list is a pair ...
#T
[16] (pair? '(1 . 2))        ; ... but each pair is not a list ...
#T
[17] (define list?           ; Let us make a list recognizer!
      (lambda (lst)
       (cond ((not (pair? lst)) #F)       ; Empty list is NOT a pair and
             ((null? lst))                ; recursion will eventually
             (else (list? (cdr lst))))))  ; create and unsuccessfully
                                          ; test the empty list ...

LIST?
[18] (list '(1 2))      ;   Oops!  "?" is missing
((1 2))
[19] (list? '(1 2))     ;-)  Let us test our first list!
()                      <--- According to our program '(1 2) is not a list!
[20] (define list?      ;   To change the list, or to change the program...?
      (lambda (lst)
        (cond ((not (pair? lst)) #F)
              ((null? lst)  #T)           ; This is better, but '() will
              (else (list? (cdr lst)))))) ; never reach this point!
LIST?
[21] (list? '(1 2))
()                      <--- The wrong answer persists ...    :-(
[22] (null? 12)         ;  What null? makes to scalars...?
()
[23] (pair? '())        ; Finally, the right question!
()                      <--- Voila! :-))

[24] (define (list? lst)
       (cond ((null? lst) #T)             ; This should be the right
             ((not (pair? lst)) #F)       ; program ...
             (else (list? (cdr lst)))))
LIST?
[25] (list? '(1 2))    ;  ... but does it really work ???
#T
[26] (list '())        ;  ... few more tests ... and too fast typing
(())
[27] (list? '())       ;  the empty list is now recognized
#T
[28] (list? '(1 . 2)) ;  this pair is not a list
()
[29] (list? 13)        ;  the scalar is also not a list
()
[30] (list? '((())))   ;  ... but this should be a list!
#T
[31] (list? 'a)        ;  ... and this is again a scalar ...
()
[32] (exit)            ;  After all, programming is FUN!
                       ;  (... but program testing proves only the
                       ;  presence of errors, not their absence ;-)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                    Basic operations with vectors
;                    v[0] , v[1] , v[2] , ...
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

[19] (define v (make-vector 3)); vector with unspecified components
V

[20] v
#(() () ())

[22] (vector-set! v 0 #(1 1 1 1))  ;  v[0] = #(1 1 1 1)
#(#(1 1 1 1) () ())

[23] (vector-set! v 2 #(3 3 3 3))
#(#(1 1 1 1) () #(3 3 3 3))

[24] (vector-set! v 1 "Second row")
#(#(1 1 1 1) "Second row" #(3 3 3 3))

[25] (vector-set! v 1 #(5 6 7 8))
#(#(1 1 1 1) #(5 6 7 8) #(3 3 3 3))

[26] (vector-ref v 1)   ;  returns v[1]
#(5 6 7 8)

[27] (vector-ref (vector-ref v 1) 2)    ;  matrix   v[1][2]
7

[28] (vector-length v)
3

[29] (vector-length (vector-ref v 1))
4

[30] v
#(#(1 1 1 1) #(5 6 7 8) #(3 3 3 3))

[31] (vector->list v)
(#(1 1 1 1) #(5 6 7 8) #(3 3 3 3))

[32] (make-vector 10 123)
#(123 123 123 123 123 123 123 123 123 123)

[33] (define v (make-vector 3))
V

[34] (vector-set! v 0 (make-vector 3))   ;  Matrix structure
#(#(() () ()) () ())

[35] (vector-set! (vector-ref v 0) 1 2)  ;  Assignment  v[0][1] = 2
#(() 2 ())
```

```
[36] v
#(#(() 2 ()) () ())

[37] (vector-set! (vector-ref v 1) 1 5)  ;  This assignment is
                                         ;  NOT allowed

[VM ERROR encountered!] Invalid operand to VM instruction
(VECTOR-SET!)

[Inspect] Quit

[38] (vector-set! v 1 (make-vector 3))
#(#(() 2 ()) #(() () ()) ())

[39] (vector-set! (vector-ref v 1) 1 5)
#(() 5 ())

[40] v
#(#(() 2 ()) #(() 5 ()) ())

[41] (begin (define r 5) (define c 3))
C

[42] r
5

[43] c
3

[44] (define mat (make-vector r (make-vector c)))
MAT

[45] mat
#(#(() () ()) #(() () ()) #(() () ()) #(() () ()) #(() () ()))

[46] (exit)

;-----------------------------------------------------------
; Functions accepting an arbitrary number of arguments
; (define f (lambda x <expressions>))    or
; (define (f . x) <expressions>)
; (f 1 2 3 4) ; When the function is called x is bound to
; the list of actual arguments  x -> (1 2 3 4)
;-----------------------------------------------------------

[1] (define (sumlist lst)
      (if (null? lst) 0 (+ (car lst) (sumlist (cdr lst)))))
SUMLIST
[2] (define (sum . x) (sumlist x)) ; sum of any number of args
SUM
[3] (sum 1 2 3)
6
```

```
[4] (define sumx (lambda x (sumlist x)))
SUMX
[5] (sumx 1 2 3)
6
[6] (sum 1 2)
3
[7] (sumx 1)
1
[8] (sum)
0

[12] (define (sum2 x y) (+ x y))  ;  # of arguments = const = 2
SUM2

[13] (sum2 3 4)
7

[14] (sum2 3 4 5) ; sum2 is a function with fixed number of arguments

[VM ERROR encountered!]
Invalid argument count: Function expected  2 argument(s)
but was called with  3 as follows:
(#<PROCEDURE SUM2> 3 4 5)

[Inspect] Quit

[18] (sum 1 2 3 4)    ;  x is here the list '(1 2 3 4)
10

[19] (sum 1 2 3)
6

[20] (sum 1 2)
3

[21] (sum 1)
1

[22] (sum )
0

[23] sum
#<PROCEDURE SUM>

[24] (define (sum . x) (sumlist x))
SUM

[25] (sum)
0

[26] (sum 1)
1
```

```
[27] (sum 1 2 3 4 5)
15

[29] (define (sumlist lst)
       (if (null? lst) 0 (+ (car lst) (sumlist(cdr lst)))))
SUMLIST

[30] (sumlist '(1 2 3 4))
10

[31] (define sum (lambda x (sumlist x)))
SUM

[32] (sum)
0

[33] (sum 1)
1

[34] (sum 1 2 3)
6

[35] (define mean (lambda x (/ (sumlist x) (length x)))) ; x bound to list
MEAN

[36] (mean 1 2 3)
2

[37] (mean 1 2)
1.5

[38] (mean 1)
1

[39] (mean )

[VM ERROR encountered!] Divide by zero
(/ 0 0)
[Inspect] Quit

[40] ; Let us define mean that has 1 mandatory (required) argument
     ; x = required argument
     ; y = optional argument(s)

     (define mean (lambda (x . y)
        (/ (sum x (sumlist y)) (+ 1 (length y)))))
MEAN

[41] (mean 1 2 3)
2

[42] (mean 1 2)
1.5
```

```
[43] (mean 1 2 3 4 5)
3

[44] (mean 1)
1

[45] (mean )
[VM ERROR encountered!]
Invalid argument count: Function expected  1 argument(s)
but was called with  0 as follows:
(#<PROCEDURE MEAN>)

[Inspect] Quit

[46] (pp mean)
#<PROCEDURE MEAN> =
(LAMBDA (X . Y)
  (/ (SUM X (SUMLIST Y)) (+ 1 (LENGTH Y))))

[47] ; Let us define mean that has 2 required arguments plus optional arguments
     ; x = required argument
     ; y = required argument
     ; z = optional argument(s)

(define mean (lambda(x y . z)
      (/ (sum x y (sumlist z)) (+ 2 (length z)))))
MEAN

[48] (mean 1 2 3 4 5)
3
[49] (mean 1 2)
1.5
[50] (mean 1)

[VM ERROR encountered!]
Invalid argument count: Function expected  2 argument(s)
but was called with  1 as follows:
(#<PROCEDURE MEAN> 1)

[Inspect] Quit

[51] (pp sum)
#<PROCEDURE SUM> =
(LAMBDA X
  (SUMLIST X))

[52] (pp mean)
#<PROCEDURE MEAN> =
(LAMBDA (X Y . Z)
  (/ (SUM X Y (SUMLIST Z)) (+ 2 (LENGTH Z))))

[53] (exit)
```

```
[3] ;-----------------------------------------------------
    ;  First-class objects:
    ;    - may be passed to procedures
    ;    - may be returned from procedures
    ;    - may be stored in data structures
    ;
    ;  In Scheme all objects (including procedures) are
    ;  first-class. First-class procedures contribute
    ;  to the expressive power of language (e.g procedure
    ;  can be an argument/result of a procedure)
    ;-----------------------------------------------------

(procedure? 'car)     ;  procedure recognizer
()

[4] (procedure? car) ;  car is recognized as a procedure
#T

[5]
(define f (car (list cdr))) ; same as (define f cdr)
F

[6] (f '(1 2 3 4))  ;  f is cdr
(2 3 4)

[7] (if #f car cdr) ;  expression can return procedure
#<PROCEDURE CDR>

[8] ((if (procedure? 3) car cdr) '(a b c d))
(B C D)

[9] (apply + '(1 2 3 4)) ; apply takes elements of the list as operands
10

[10] (define abc '(a b c))
ABC

[12] (apply cons (cdr abc))  ;  same as (cons 'b 'c)
(B . C)

[13] (procedure? (lambda (n) (+ n 2))) ;   anonymous procedure
#T

[15] ((lambda (n) (+ n 2)) 4) ; apply this procedure to argument 4
6

[17] (define select       ; select the first/second element of lst
       (lambda (b lst)   ; b should be #T or #F
         (if b
            (car lst)
            (cadr lst))))
SELECT
```

```
[18] (select #F '(a b))
B

[19] ((select #t (list cdr car)) '(a b c)) ; select function returns cdr
(B C)

[20] (map (lambda (n) (+ n 2)) '(1 2 3 4))
(3 4 5 6)

[21] (map list '(a b c d))  ; make lists of individual elements
((A) (B) (C) (D))

[22] (define add2 (lambda (n) (+ n 2)))
ADD2

[23] (map add2 '(1 2 3 4))
(3 4 5 6)

[24] (define compose       ;  compose is the function of two arguments
       (lambda (f g)       ;  both arguments are functions
         (lambda (x)       ;  function f(g(x))
           (f ( g x)))))
COMPOSE

[25] (define add4 (compose add2 add2)) ; add4(x)=add2(add2(x))=x+2+2
ADD4

[26] (add4 5)
9

[27] ((compose car cdr) '(a b c d)) ; car(cdr(lst))=cadr(lst)=second element
B

[28] (map (lambda (f) (f '(a b c d)))      ; argument f is an operator
          (list car cdr cadr cddr caddr)) ; elements of this list will replace f
(A (B C D) B (C D) C)

[28] (define f                  ;  list of three functions
       '( (LAMBDA (N) (+ N 2))
          (LAMBDA (N) (* N 2))
          (LAMBDA (N) (* N N))))
F

[29] (map (eval (caddr f)) '(1 2 3 4 5))
(1 4 9 16 25)

[30] (map (eval (cadr f)) '(1 2 3 4 5))
(2 4 6 8 10)
```

```
[31] ;----------------------------
     ;  Scope
     ;----------------------------

(define x 3)  ; definition of a global x
X

[32] x
3

[33] (define (f)
       (begin
         (define x 12)  ;  this x is a local object of function f
         (display x)
         (set! x 34)
         (display x)
         *the-non-printing-object*))  ;  returned value
F

[34] (f)
1234

[35] x  ;  global x remains unchanged
3

[36] (define (g)
       (begin
          (set! x 3.14159)   ;  this modifies the global x
          (display x)
          *the-non-printing-object*))
G

[37] (g)
3.14159

[38] x    ;  global x is now modified by function g
3.14159

-------------------------------------------------
PROGRAM :  REGION & SCOPE
AUTHOR  :  Jozo J. Dujmovic
DATE    :  11/14/94
-------------------------------------------------

[2] (define one 1)
ONE

[3] (let* ((two (+ one one))
           (three (+ one two)))
         (* two three))
6

[4] one
1
```

```
[5] two
[VM ERROR encountered!] Variable not defined in current environment
TWO
[Inspect] Quit
[6] three
[VM ERROR encountered!] Variable not defined in current environment
THREE
[Inspect] Quit

[7] (let* ((one 100)
           (two (+ one one))
           (three (+ one two)))
          (* two three))
60000

[8] one
1

[9] two
[VM ERROR encountered!] Variable not defined in current environment
TWO
[Inspect] Quit

[11] (let* ((one 100)
            (two (+ one one))
            (three (+ one two)))

          (do ((one 1 (+ 1 one)))
              ((> one 3) (newline) (display ""))
              (display one) (display " "))

          (display one) (display " ") (display two)(display " ")
          (display three) (newline) one)
1 2 3
100 200 300
100

[12] one
1

[13] (let* ((one 100)
            (two (+ one one))
            (three (+ one two)))

           (do ((one 1 (* 2 one)))
               ((> one three) (newline) (display ""))
               (display one) (display " "))

           (display one)(display " ")(display two)(display " ")
           (display three) (newline) one)

1 2 4 8 16 32 64 128 256
100 200 300
100
[14] one
1

[15] two

[VM ERROR encountered!] Variable not defined in current environment
TWO
[Inspect] Quit

[16] (exit)
```

```
=============================================================
                    FILE INPUT/OUTPUT
=============================================================

In this section we are going to use the files VEC.DAT, MAT.DAT,
SYMBOL.DAT, STRING.DAT, and ALLDATA.DAT having the following
contents:

VEC.DAT

1
2
3

MAT.DAT

1 2 3
4 5 6
7 8 9

SYMBOL.DAT

for i := 1 to 10 do

STRING.DAT

"First line"
"Second line"
"Third line"

ALLDATA.DAT

123 #F symbol "string" ((1 2) (3 4 5)) #(#(1 2) #(3 4 5))


-------------------------------------------------
PROGRAM :  FILE I/O
AUTHOR  :  Jozo J. Dujmovic
DATE    :  11/6/1994
-------------------------------------------------

[2] (define inport (open-input-file "alldata.dat"))
INPORT
[3] inport
#<PORT>
[4] (read inport)
123
[5] (read inport)
()
[6] (read inport)
SYMBOL
[7] (read inport)
"string"
[8] (read inport)
((1 2) (3 4 5))
[9] (read inport)
#(#(1 2) #(3 4 5))
[10] (read inport)
#!EOF

[11] (read inport)
#!EOF
```

```
[12] (close-input-port inport)
#<PORT>

[13] (read inport)  ;  inport is no longer defined

[VM ERROR encountered!] DOS I/O error number 6
#<PORT>

[Inspect] Quit

[15] (define (fread inport)
       (let ((item (read inport)))
       (cond
         ((eof-object? item) (display ""))
         (else (display " ") (display item) (fread inport)))))
FREAD
[16] (pp fread)
#<PROCEDURE FREAD> =
(LAMBDA (INPORT)
  (LET ((ITEM (READ INPORT)))
    (COND ((EOF-OBJECT? ITEM) (DISPLAY ""))
          (ELSE (DISPLAY " ")
                (DISPLAY ITEM)
                (FREAD INPORT)))))
[17] (define inport (open-input-file "vec.dat"))
INPORT
[18] (fread inport)
 1 2 3
[19] (fread (open-input-file "mat.dat"))    ;  sequential scanning of
                                            ;  matrix elements
 1 2 3 4 5 6 7 8 9
[20] (fread inport)
[21] (fread (open-input-file "vec.dat"))
 1 2 3
[22] (fread (open-input-file "symbol.dat"))
 FOR I := 1 TO 10 DO
[23] (fread (open-input-file "string.dat"))
 First line Second line Third line
[24] (fread (open-input-file "alldata.dat"))
 123 () SYMBOL string ((1 2) (3 4 5)) #(#(1 2) #(3 4 5))
[25] (close-input-port inport)
#<PORT>

[26] (define (show file)
       (let ((inport (open-input-file file)))
            (fread inport)
            (close-input-port inport)))   ; Close command returns the
                                          ; string #<PORT> !
SHOW

[27] (show "vec.dat")    ;  This will append #<PORT> at the end:
 1 2 3#<PORT>

[28] (show "symbol.dat")
 FOR I := 1 TO 10 DO#<PORT>

[29] (define (show file)
      (let ((inport (open-input-file file)))
           (fread inport)
           (close-input-port inport)
           (display "")))             ;  In this way we omit the #<PORT>
SHOW
```

```
[30] (pp show)
#<PROCEDURE SHOW> =
(LAMBDA (FILE)
  (LET ((INPORT (OPEN-INPUT-FILE FILE)))
    (FREAD INPORT)
    (CLOSE-INPUT-PORT INPORT)
    (DISPLAY "")))

[31] (show "vec.dat")
 1 2 3

[32] (show "mat.dat")
 1 2 3 4 5 6 7 8 9

[33] (show "string.dat")
 First line Second line Third line

[34] (show "symbol.dat")
 FOR I := 1 TO 10 DO

[35] (show "alldata.dat")
 123 () SYMBOL string ((1 2) (3 4 5)) #(#(1 2) #(3 4 5))

[36] (show "nonexisting.file")

[VM ERROR encountered!] DOS I/O error - File not found
"nonexisting.file"

[Inspect] Quit

[37] (show nonexisting.file)

[VM ERROR encountered!] Variable not defined in current environment
NONEXISTING.FILE

[Inspect] Quit

[38] (show 123)

[VM ERROR encountered!] Invalid operand to VM instruction
(OPEN-PORT 123 READ)

[Inspect] Quit

[42] (load "fio.s")
OK

[43] (pp portcopy)

#<PROCEDURE PORTCOPY> =
(LAMBDA (INPORT OUTPORT)
  (LET ((ITEM (READ INPORT)))
    (COND ((EOF-OBJECT? ITEM)
           (NEWLINE OUTPORT)
           (CLOSE-OUTPUT-PORT OUTPORT)
           (CLOSE-INPUT-PORT INPORT)
           (DISPLAY ""))
          (ELSE (DISPLAY " " OUTPORT)
                (DISPLAY ITEM OUTPORT)
                (PORTCOPY INPORT OUTPORT)))))
```

```
[44] (pp fcopy)

#<PROCEDURE FCOPY> =
(LAMBDA (INFILE OUTFILE)
  (LET ((INPORT (OPEN-INPUT-FILE INFILE))
        (OUTPORT (OPEN-OUTPUT-FILE OUTFILE)))
    (PORTCOPY INPORT OUTPORT)
    (DISPLAY "File copy completed.")))

[45] (fcopy "mat.dat" "mat1.dat")
File copy completed.

[46] (show "mat.dat")
 1 2 3 4 5 6 7 8 9

[47] (show "mat1.dat")
 1 2 3 4 5 6 7 8 9

[48] (fcopy "mat.dat" "mat1.dat")
File copy completed.

[49] (show "mat1.dat")
 1 2 3 4 5 6 7 8 9

[50] (fcopy "symbol.dat" "mat1.dat")
File copy completed.

[51] (show "mat1.dat")
 FOR I := 1 TO 10 DO

[52] (exit)
```

```
------------------------------------------------------------
PROGRAM :  BNF example with short-circuit and/or operations
AUTHOR  :  Jozo J. Dujmovic
DATE    :  11/4/94
------------------------------------------------------------

; <list of numbers> ::= () | <number> . <list of numbers>
(define list-of-numbers?
 (lambda (lst)
  (if (null? lst)
      #T
      (if (pair? lst)
          (if (number? (car lst))
              (list-of-numbers? (cdr lst))
              #F)
          #F))))

; <list of numbers> ::= () | <number> . <list of numbers>
(define lon?
 (lambda (lst)
  (or (null? lst)
      (and (pair? lst)
           (number? (car lst))
           (list-of-numbers? (cdr lst))))))


[2] (define l '(1 2 3 4 5))
L
[3] (define n '(1 2 3 four five))
N
[4] (define v '#(1 2 3 4 5))
V
[5] l
(1 2 3 4 5)
[6] n
(1 2 3 FOUR FIVE)
[7] v
#(1 2 3 4 5)
[8] (list-of-numbers? l)
#T
[9] (lon? l)
#T
[10] (list-of-numbers? n)
()
[11] (lon? n)
()
[12] (list-of-numbers? v)
()
[13] (lon? v)
()
[14] (list-of-numbers? 123)
()
[15] (lon? 123)
()
[16] (lon? '(+ 1 2))
()
[17] (list-of-numbers? '(+ 1 2))
()
[18] (exit)
```

CSc 600: PROGRAMMING LANGUAGE DESIGN **HOMEWORK #1**

Date Due: 9/19/1994 Dr. Jozo Dujmovic

1. Write the BNF definitions of the syntax for

   (a) nonzero integers,
   (b) even integers,
   (c) odd integers,
   (d) natural numbers.

   In all cases the numbers can include the leading zeros.

2. Create a Scheme transcript that illustrates arithmetic operations, and elementary functions **abs, quotient, remainder, modulo, round, truncate, floor, ceiling, exp, log, gcd, lcm,** and **sqrt**.

3. Create Scheme functions **ftoc** and **ctof** that convert a temperature from Fahrenheit to Celsius and from Celsius to Fahrenheit. Create a transcript illustrating the definitions and the use of these functions.

4. Write a Scheme function **power** that returns x^y for real x and y and create a transcript illustrating its use.

NOTES: *All Scheme programs must be presented as complete transcripts which include the headers with author name, program name, program description, date, and homework and problem numbers (e.g. HW1/4). Transcripts must include sample program outputs. All critical points in a program must be explained using comments.*

Late homework cannot be accepted after the distribution of solutions.

A N S W E R S

PROBLEM #1

(a)

```
<nonzero integer> ::= <sign> <natural number>
```

(b)

```
<nonnegative even integer> ::= <even digit> |
                               <digit> <nonnegative even integer>
<even integer> ::= <sign> <nonnegative even integer>
```

(c)

```
<positive odd integer> ::= <odd digit> |
                           <digit> <positive odd integer>
<odd integer> ::= <sign> <positive odd integer>
```

(d)

```
<natural number>  ::= <nonzero digit> | <digit> <natural number>
                                      | <natural number> <digit>
```

```
<sign>          ::= + | - | <empty>
<empty>         ::=
<even digit>    ::= 0 | 2 | 4 | 6 | 8
<odd digit>     ::= 1 | 3 | 5 | 7 | 9
<nonzero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digit>         ::= 0 | <nonzero digit>
```

```
PROBLEM #2

PROGRAM :  Elementary functions
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 15, 1994
-------------------------------------------------

[3] (abs -3.14)
3.14

[4] (quotient -7 2)
-3

[5] (remainder -7 2)
-1

[6] (modulo -7 2)
1

[7] (modulo 7 3)
1

[8] (modulo 7 2)
1

[9] (round 7.5)
8

[10] (truncate 7.999)
7

[11] (floor 3.14)
3

[12] (ceiling 3.14)
4

[13] (floor -3.14)
-4

[14] (ceiling -3.14)
-3

[15] (log (exp 1))
1.

[16] (sqrt 1024)
32.

[17] (gcd 11 13)
1

[18] (gcd 11 33)
11

[19] (lcm 11 33)
33

[20] (exit)
```

PROBLEM #3

temper.s file:

```
(set! pcs-debug-mode #T)

(define (ftoc f) (/ (* 5 (- f 32)) 9))
(define (ctof c) (+ 32 (/ (* 9 c) 5)))

(set! pcs-debug-mode #F)
(pp ftoc)(newline) (pp ctof)(newline)
```

```
PROGRAM :  Temperature conversion
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 15, 1994
-----------------------------------------------

[3] (load "temper.s")

#<PROCEDURE FTOC> =
(LAMBDA (F)
  (/ (* 5 (- F 32)) 9))

#<PROCEDURE CTOF> =
(LAMBDA (C)
  (+ 32 (/ (* 9 C) 5)))

OK

[4] (ftoc 212)
100

[5] (ctof 0)
32

[6] (ctof 36.6)
97.88

[7] (ftoc 350)
176.666666666667

[8] (exit)
```

Problem #4

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; The function power raises x to the y power. It is frequently considered
; the most complex of all math functions. The complexity is related to
; numerical accuracy and to various domain errors. A discussion of these
; problems can be found in P.J. Plauger, The Standard C Library, Prentice
; Hall, 1992, p. 164. Our goal is not to achieve the maximum numerical
; accuracy, and therefore we are going to directly use the simplest
; approach based on the formula x^y = exp(y log x).
;
; The ISO standard of C language specifies double pow(double x, double y)
; as the power function generating a domain error if x is negative and y
; is not an integral value. A domain error occurs if the result cannot be
; represented when x=0 and y<=0 (if y>0 then 0^y=0; both 1/0 and 0^0 are
; undefined, but if x->0 then lim 1/x = +infinity, and lim x^x = 1, and
; some compilers return these limit values [e.g. both VAX/Ultrix f77 and
; Microsoft Fortran 5.1 return 0.**0.=1., while Microsoft Quick C for
; pow(0.,0.) returns a domain error; the compilers for 0.^(-1.2) return
; a domain error, but strange things can also be encountered: e.g. f77
; returns (-2.)**3. = -8.00, but (-2.)**3.0001 = 0. without any error!])
;
; The majority of modern compilers and interpreters properly compute
; x^y for negative x and integral y, i.e. usually (-2)^3.0=-8 (some
; older compilers used to generate in this case a run time error!)
;
; Let us now define all cases that can be computed using the formula
;
;          x^y = exp log (x^y) = exp (y log x) =: p(x,y)
;
; or using the limit values as shown in the following tree:
;
;
; +-- x>0 ------ y can take any value -------------------> p(x,y)
; |
; +-- x<0 --+--- y is an integral value --+-- y is even --> p(-x,y)
; |         |                             |
; |         |                             `  y is odd ---> -p(-x,y)
; |         |
; |         `--- y is not an integral value -------------> Domain error
; |
; `-- x=0 --+--- y>0 ------------------------------------> 0
;           |
;           +--- y<0 ------------------------------------> +Infinity
;           |
;           `--- y=0 ------------------------------------> 1
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(set! pcs-debug-mode #T)

(define (p x y) (exp (* y (log x))))

(define (power x y)
(if (> x 0)
    (p x y)
    (if (< x 0)                    ; The case where x < 0
        (if (= y (round y))        ; If y = round(y) then y is an
                                   ;                 integral value
            (if (even? (round y))  ; Argument of even must be an
                (p (- x) y)        ; integer; use round to convert
                (- (p (- x) y)))   ; real y to integer
            (display "Domain error: x<0, and y is not an integral value"))
        (if (> y 0)                ; The case where x = 0
            0
            (if (< y 0)
                (display "+Infinity")
                1)))))

(set! pcs-debug-mode #F)
(pp p)(newline) (pp power)(newline)

------------------------------------------------
PROGRAM :  Power function
AUTHOR  :  Jozo J. Dujmovic
DATE    :  SEP 15, 1994
------------------------------------------------
[3] (load "power.s")
#<PROCEDURE P> =
(LAMBDA (X Y)
  (EXP (* Y (LOG X))))
#<PROCEDURE POWER> =
(LAMBDA (X Y)
  (IF (> X 0)
      (P X Y)
      (IF (< X 0)
          (IF (= Y (ROUND Y))
              (IF (EVEN? (ROUND Y))
                  (P (- X) Y)
                  (- (P (- X) Y)))
              (DISPLAY
                 "Domain error: x<0, and y is not an integral value"))
          (IF (> Y 0) 0 (IF (< Y 0) (DISPLAY "+Infinity") 1)))))
OK

[4] (power 2 3)
8.
[5] (power 2.001 3.001)
8.01756543329107
[6] (power 2 -3)
0.125
[7] (power -2 4)
16.
```

```
[8] (power -2. 4.)
16.
[9] (power -2 -4)
0.0625
[10] (power -2. -4.)
0.0625
[11] (power -2 3)
-8.
[12] (power -2 -3)
-0.125
[13] (power -2. -3.)
-0.125
[14] (power -2. -3.001)
Domain error: x<0, and y is not an integral value
[15] (power 0 3)
0
[16] (power 0 -3.)
+Infinity
[17] (power 0 0)
1
[18] (power 0. 0.)
1
[19] (exit)
```

```
#include <math.h>

/*********************************************\
     Microsoft Quick C
\*********************************************/

main()
{
   printf("0.**0          = %g\n", pow(0.,0));
   printf("0.**0.         = %g\n", pow(0.,0.));
   printf("0.**1.2        = %g\n", pow(0.,1.2));
   printf("2.**3.         = %g\n", pow(2.,3.));
   printf("-2.**3         = %g\n", pow(-2.,3));
   printf("-2.**3.0000    = %g\n", pow(-2.,3.0000));
   printf("-2.**3.00001   = %g\n", pow(-2.,3.00001));
   printf("-2.**(-3.1)    = %g\n", pow(-2.,-3.1));
   printf("0.**(-1.2)     = %g\n", pow(0.,-1.2));
}
```

```
0.**0          = 0
0.**0.         = 0
0.**1.2        = 0
2.**3.         = 8
-2.**3         = -8
-2.**3.0000    = -8
-2.**3.00001   = 0
-2.**(-3.1)    = 0
0.**(-1.2)     = 0
```

```
pow: DOMAIN error
pow: DOMAIN error


pow: DOMAIN error
pow: DOMAIN error
pow: DOMAIN error
```

CSc 600: PROGRAMMING LANGUAGE DESIGN **HOMEWORK #2**

Date Due: 10/3/1994

Dr. Jozo Dujmovic

1. Scheme provides the following character predicates: char-alphabetic?, char-numeric?, char-whitespace?, char-upper-case?, and char-lower-case?. Write your own functions char-alphabetic, char-numeric, char-whitespace, char-upper-case, and char-lower-case, that perform the same operations as the corresponding Scheme functions.

2. Develop your own version of **max**, a function that returns the largest number in a list, passed as an argument:

```
> (maximum '(1 2 4 3 4))
4
```

3. Define a procedure **juggle** that rotates a three-element list as follows:

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

4. Define procedure **remove-last-item** that reduces the list by removing its last item:

```
> (remove-last-item '(1 2 3 4))
(1 2 3)
```

5. Define procedure **first-and-last** that makes a list consisting of the first and the last item in a list, passed as an argument:

```
> (first-and-last '(1 2 3 4))
(1 4)
> (first-and-last '(1))
(1 1)
```

NOTES: *All Scheme programs must be presented as complete transcripts which include the headers with author name, program name, program description, date, and homework and problem numbers (e.g. HW1/4). Transcripts must include sample program outputs. All critical points in a program must be explained using comments.*

Late homework cannot be accepted after the distribution of solutions.

```
------------------------------------------------
PROGRAM :   Homework #2  -  ANSWERS
AUTHOR  :   Jozo J. Dujmovic
DATE    :   SEP 25, 1994
------------------------------------------------

[2] (load "hw2.s")

=======================================================
                PROBLEM # 2.1
=======================================================

#<PROCEDURE CHAR-ALPHABETIC> =
(LAMBDA (C)
  (AND (CHAR? C)
       (OR (AND (CHAR>=? C #\a) (CHAR<=? C #\z))
           (AND (CHAR>=? C #\A) (CHAR<=? C #\Z)))))

#<PROCEDURE CHAR-NUMERIC> =
(LAMBDA (C)
  (AND (CHAR? C) (CHAR>=? C #\0) (CHAR<=? C #\9)))

#<PROCEDURE CHAR-WHITESPACE> =
(LAMBDA (C)
  (CASE C
    ((#\SPACE #\TAB #\NEWLINE #\PAGE #\RETURN) #T)
    (ELSE ())))

#<PROCEDURE CHAR-UPPER-CASE> =
(LAMBDA (C)
  (AND (CHAR? C) (CHAR>=? C #\A) (CHAR<=? C #\Z)))

#<PROCEDURE CHAR-LOWER-CASE> =
(LAMBDA (C)
  (AND (CHAR? C) (CHAR>=? C #\a) (CHAR<=? C #\z)))


[3] (char-alphabetic "string")
()
[4] (char-alphabetic #\3)
()
[5] (char-alphabetic #\x)
#T
[6] (char-numeric #\x)
()
[7] (char-numeric #\3)
#T
[8] (char-whitespace #\tab)
#T
[9] (char-upper-case #\x)
()
[10] (char-upper-case #\X)
#T
[12] (char-lower-case #\X)
()
[13] (char-lower-case #\x)
#T
```

```
=========================================================
                    PROBLEM # 2.2
=========================================================

#<PROCEDURE MAXIMUM> =
(LAMBDA (LST)
  (COND ((NULL? (CDR LST)) (CAR LST))
        ((> (CAR LST) (MAXIMUM (CDR LST))) (CAR LST))
        (ELSE (MAXIMUM (CDR LST)))))

[14] (maximum '(1 2 3 4 3 2 1))
4

[15] (maximum '())
()


=========================================================
                    PROBLEM # 2.3
=========================================================

#<PROCEDURE JUGGLE> =
(LAMBDA (LST)
  (COND ((= 3 (LENGTH LST))
         (LIST (CADDR LST) (CAR LST) (CADR LST)))
        (ELSE (DISPLAY "The length of list is not 3. Try again!"))))

[16] (juggle '())
The length of list is not 3. Try again!

[17] (juggle '(a b c))
(C A B)

[18] (juggle '(111 222 333))
(333 111 222)



=========================================================
                    PROBLEM # 2.4
=========================================================

#<PROCEDURE REMOVE-LAST-ITEM> =
(LAMBDA (LST)
  (REVERSE (CDR (REVERSE LST))))


[19] (remove-last-item '(first second last))
(FIRST SECOND)

[20] (remove-last-item '())
()

[21] (remove-last-item '(single))
()
```

```
============================================================
                    PROBLEM # 2.5
============================================================

#<PROCEDURE FIRST-AND-LAST> =
(LAMBDA (LST)
  (LIST (CAR LST) (CAR (REVERSE LST))))

[23] (first-and-last '(single))
(SINGLE SINGLE)

[24] (first-and-last '())
(() ())

[25] (first-and-last '(first second third last))
(FIRST LAST)

[26] (exit)
```

CSc 600: PROGRAMMING LANGUAGE DESIGN **HOMEWORK #3**

Date Due: 10/11/1994

Dr. Jozo Dujmovic

1. Write an interactive program that prompts for a number and then prints the square and the square root of that number. It continues prompting for numbers until *stop* is entered. The display should include the appropriate text to identify the input and output. For example:

```
> (square-and-root)
Enter the number whose square and square root you want, or enter stop to quit: 2
The square of 2 is 4
The square root of 2 is 1.41
Enter the number whose square and square root you want, or enter stop to quit: stop
Good bye.
>
```

2. Write an interactive program for solving the quadratic equation $ax^2+bx+c=0$. The program must prompt the user to enter the values of a, b, and c. Then, the program must perform a logic control and should not accept the case where $a=b=0$ (in such a case the program must give an error message, and ask the user to enter correct data). For all other values of input data the program must give correct solutions, including the cases of real and complex results.

3. Develop a program that computes the scalar product of two vectors. The program must not accept vectors having different size (in such a case print an error message). For example

```
>(scalar-product '#(1 2 3) '#(2 1 1))
7
>(scalar-product '#(1 2 3) '#(1 2 3 4 5))
ERROR: Different sizes of vectors!
>
```

NOTES: *All Scheme programs must be presented as complete transcripts which include the headers with author name, program name, program description, date, and homework and problem numbers (e.g. HW1/4). Transcripts must include sample program outputs. All critical points in a program must be explained using comments.*

Late homework cannot be accepted after the distribution of solutions.

```
------------------------------------------------
PROGRAM :  HW # 3  -  ANSWERS
AUTHOR  :  Jozo J. Dujmovic
DATE    :  OCT 8, 1994
------------------------------------------------

[PCS-DEBUG-MODE is ON]


================================================
                PROBLEM #1
================================================

[1] (pp square-and-root)

#<PROCEDURE SQUARE-AND-ROOT> =
(LAMBDA ()
  (WRITELN
     "Enter the number whose square and square root you want,")
  (DISPLAY "or enter stop to quit: ")
  (LET ((N (READ)))
    (IF (EQV? N 'STOP)
        (DISPLAY "Good bye.")
        (BEGIN
          (WRITELN "The square of " N " is " (* N N))
          (WRITELN "The square root of " N " is " (SQRT N))
          (SQUARE-AND-ROOT)))))

[2] (square-and-root)
Enter the number whose square and square root you want,
or enter stop to quit: 144
The square of 144 is 20736
The square root of 144 is 12.
Enter the number whose square and square root you want,
or enter stop to quit: 1024
The square of 1024 is 1048576
The square root of 1024 is 32.
Enter the number whose square and square root you want,
or enter stop to quit: 1023
The square of 1023 is 1046529
The square root of 1023 is 31.984371183439
Enter the number whose square and square root you want,
or enter stop to quit: stop
Good bye.
```

```
================================================
                  PROBLEM #2
================================================

[3] (pp qe)

#<PROCEDURE QE> =
(LAMBDA (A B C)
  (COND ((AND (EQV? A 0) (EQV? B 0))
         (DISPLAY "Error: a = b = 0"))
        ((EQV? A 0)
         (WRITELN "Linear equation: x = " (- (/ C B))))
        (ELSE (LET ((D (- (* B B) (* 4 A C))))
                (COND ((EQV? D 0)
                       (WRITELN "X1 = X2 = " (- (/ B (* 2 A)))))
                      ((> D 0)
                       (WRITELN "X1 = " (/ (+ (- B) (SQRT D)) (* 2 A)))
                       (WRITELN "X2 = " (/ (- (- B) (SQRT D)) (* 2 A))))
                      (ELSE (WRITELN
                                "X1 = "
                                (- (/ B (* 2 A)))
                                " + j * "
                                (/ (SQRT (- D)) (* 2 A)))
                            (WRITELN
                                "X2 = "
                                (- (/ B (* 2 A)))
                                " + j * "
                                (- (/ (SQRT (- D)) (* 2 A))))))))))

[4] (qe 0 0 0)
Error: a = b = 0
[5] (qe 0 1 1)
Linear equation: x = -1
()
[6] (qe 1 0 0)
X1 = X2 = 0
()
[7] (qe 1 2 1)
X1 = X2 = -1
()
[8] (qe 1 0 -1)
X1 = 1.
X2 = -1.
()
[9] (qe 1 -1 0)
X1 = 1.
X2 = 5.551115123125 78e-17
()
[10] (qe 1 0 1)
X1 = 0 + j * 1.
X2 = 0 + j * -1.
()
[11] (qe 1 1 1)
X1 = -0.5 + j * 0.866025403784439
X2 = -0.5 + j * -0.866025403784439
()
```

CSc 600: PROGRAMMING LANGUAGE DESIGN **HOMEWORK #4**

Date Due: 11/2/1994 Dr. Jozo Dujmovic

1. If we have an estimate p for the square root of a positive number x, a better estimate (denoted r) is given by the average of p and x/p:

$$r = 0.5(p + x/p)$$

Develop a program that computes and prints a sequence of successive approximations of the square root. The computation should stop when the difference between p and r becomes sufficiently small (e.g. 0.0000001).

2. Write the procedure **compose3** that takes as arguments three procedures, f, g, and h, and returns their composition such that for each argument x the resulting function is f(g(h(x))). Similarly, define **compose4** that returns e(f(g(h(x)))). As an additional exercise make the procedures **compose3** and **compose4** using (similarly defined) **compose2**. Show several examples of the use of your compose procedures.

3. Curry the procedure * to get a procedure **curried*** and use it to define the procedure **times10** that multiplies its argument by 10. Show several examples of the use of **times10**.

NOTES: *All Scheme programs must be presented as complete transcripts which include the headers with author name, program name, program description, date, and homework and problem numbers (e.g. HW1/4). Transcripts must include sample program outputs. All critical points in a program must be explained using comments.*

Late homework cannot be accepted.

```
================================
PROGRAM :  HW #4  -  ANSWERS
AUTHOR  :  Jozo J. Dujmovic
DATE    :  10/27/1994
================================

PROBLEM #1
==========

(define next-iteration
  (lambda(x p ermax iter)
    (let ((r (* 0.5 (+ p (/ x p)))))
      (newline) (display iter) (display #\tab)
      (display r) (display "      ") (display p)
      (if ( > (abs (- r p)) ermax)
          (next-iteration x r ermax (add1 iter))
          (display "")))))

(define (sqr x)
        (newline)
        (display "Computation of the square root of ") (display x)
        (newline)
        (display "-----------------------------------------------")
        (display "Iteration      r                    p")
        (display "-----------------------------------------------")
        (if (zero? x) (begin (newline) (display 0))
                      (next-iteration (abs x) (abs x)  0  1))
        (if (negative? x) (display " i ") (display ""))
        (display "    <---- Square root of ") (display x)
        (newline) (newline) (display ""))

[2] (sqr 5)  ;

Computation of the square root of 5
-----------------------------------------------
Iteration       r                  p
-----------------------------------------------
1       3.      5
2       2.33333333333333   3.
3       2.23809523809524   2.33333333333333
4       2.23606889564336   2.23809523809524
5       2.23606797749998   2.23606889564336
6       2.23606797749979   2.23606797749998
7       2.23606797749979   2.23606797749979   <---- Square root of 5

[3] (sqr -5)

Computation of the square root of -5
-----------------------------------------------
Iteration       r                  p
-----------------------------------------------
1       3.                 5
2       2.33333333333333   3.
3       2.23809523809524   2.33333333333333
4       2.23606889564336   2.23809523809524
5       2.23606797749998   2.23606889564336
6       2.23606797749979   2.23606797749998
7       2.23606797749979   2.23606797749979 i    <---- Square root of -5
```

```
[4] (sqr 0)

Computation of the square root of 0
-----------------------------------------------
Iteration       r                    p
-----------------------------------------------
0   <---- Square root of 0

[5] (sqr 0.25)

Computation of the square root of 0.25
-----------------------------------------------
Iteration       r                    p
-----------------------------------------------
1       0.625                    0.25
2       0.5125                   0.625
3       0.50015243902439         0.5125
4       0.500000023230574        0.50015243902439
5       0.500000000000001        0.500000023230574
6       0.5                      0.500000000000001
7       0.5                      0.5   <---- Square root of 0.25

[6] (sqr 7654321)

Computation of the square root of 7654321
-----------------------------------------------
Iteration       r                    p
-----------------------------------------------
1    3827161.                 7654321
2    1913581.49999987         3827161.
3    956792.749998628         1913581.49999987
4    478400.37498834          956792.749998628
5    239208.187405332         478400.37498834
6    119620.092989904         239208.187405332
7    59842.0407895923         119620.092989904
8    29984.9747728471         59842.0407895923
9    15120.1233283909         29984.9747728471
10   7813.17868029902         15120.1233283909
11   4396.42333174248         7813.17868029902
12   3068.72849539665         4396.42333174248
13   2781.51286502992         3068.72849539665
14   2766.6840969584          2781.51286502992
15   2766.64435762517         2766.6840969584
16   2766.64435733977         2766.64435762517
17   2766.64435733977         2766.64435733977  <--- Square root of 7654321
```

```
PROBLEM #2
==========
(define compose2
  (lambda (f g)
    (lambda (x) (f (g x)))))

(define (compose3 f g h)
    (lambda (x) (f (g (h x)))))
(define (compose31 f g h)
    (lambda (x) (f ((compose2 g h) x))))
(define (compose32 f g h)
    (lambda (x) ((compose2 (compose2 f g) h) x)))
(define (t3 f g h x)
    (list ((compose3 f g h) x) ((compose31 f g h) x)
                               ((compose32 f g h) x)))


(define (compose4 e f g h)
    (lambda (x) (e (f (g (h x))))))
(define (compose42 e f g h)
    (lambda (x) ((compose2 e f) ((compose2 g h) x))))
(define (compose43 e f g h)
    (lambda (x) ((compose2 (compose2 e f) (compose2 g h)) x)))
(define (t4 e f g h x)
    (list ((compose4 e f g h) x)((compose42 e f g h) x)
                                 ((compose43 e f g h) x)))


[3] ; Functions can be directly replaced by lambda expressions.
    ; To test the superposition of functions it is suitable to use
    ; inverse functions, e.g. sqrt and square:
((compose2 sqrt (lambda(x) (* x x))) 123)
123.
[4] (define square (lambda(x) (* x x)))
SQUARE
[5] ((compose2 square sqrt) 123)
123.
[6] ((compose3 square sqrt square) 4)
16.
[7] ((compose32 square sqrt square) 4)
16.
[8] ((compose4 sqrt square sqrt square) 123)
123.
[9] ((compose42 sqrt square sqrt square) 123)
123.
[10] ((compose42 sqrt sqrt (lambda(x) (* x x)) (lambda(x) (* x x))) 123)
123.
[11] (t3 square square square 2)
(256 256 256)
[12] (t3 sqrt sqrt sqrt 256)
(2. 2. 2.)
[13] (t4 square square square square 2)
(65536 65536 65536)
[14] (t4 sqrt sqrt sqrt sqrt 65536)
(2. 2. 2.)
```

```
PROBLEM #3
==========

(define curried*
  (lambda (x)
    (lambda(y)
      (* x y))))

(define times10 (curried* 10))

[15] (times10 123)  ;
1230

[16] ((compose2 times10 times10) 123)
12300
```

CSc 600: PROGRAMMING LANGUAGE DESIGN **HOMEWORK #5**

Date Due: 11/14/1994 Dr. Jozo Dujmovic

The files "matrix1.dat" and "matrix2.dat" are created using a text editor and contain two rectangular matrices. For example,

matrix1.dat:

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

matrix2.dat:

```
3 3
1 2 3
1 2 3
1 2 3
```

In both cases the first row contains the size of the matrix (the number of rows and the number of columns). The remaining rows contain the values of elements.

1. Develop programs **row** and **col** that read a matrix from a file and display a specified row or column. For example:

```
> (row "matrix1.dat" 2)
4 5 6
> (col "matrix1.dat" 2)
2 5
```

Matrices should be stored in memory as vectors whose components are vectors.

2. Develop a program for matrix multiplication **mmul** that multiplies two matrices stored in specified input files, and creates and displays an output file containing the product. For example:

```
> (mmul "matrix1.dat" "matrix2.dat" "matrix3.dat")
 6  12  18
15  30  45
```

In this example the contents of the new file "matrix3.dat" should be

```
 2   3
 6  12  18
15  30  45
```

NOTES: All Scheme programs must be presented as complete transcripts which include the headers with author name, program name, program description, date, and homework and problem numbers (e.g. HW1/4). Transcripts must include sample program outputs. All critical points in a program must be explained using comments. ***Late homework cannot be accepted.***

```
------------------------------------------------
PROGRAM :  Homework #5   -   A N S W E R S
AUTHOR  :  Jozo J. Dujmovic
DATE    :  11/18/94
------------------------------------------------

PROBLEM #1
==========

;  Display components of the vector

(define (display-vector v)
 (do ((i 0 (add1 i)))
     ((>= i (vector-length v)) (display ""))
     (display (vector-ref v i)) (display " ")))

;  Read from file and return a matrix

(define (read-matrix filename)
 (let* ((inport (open-input-file filename))
        (nrow (read inport))
        (ncol (read inport))
        (mat  (make-vector nrow)))
       (do ((i 0 (add1 i)))
           ((>= i nrow) (close-input-port inport) mat )
           (let ((row (make-vector ncol)))
             (do ((j 0 (add1 j)))
                 ((>= j ncol) (vector-set! mat i row))
                 (vector-set! row j (read inport)))))))

;  Return i-th row of the matrix in filename

(define (ro filename i)
  (define mat (read-matrix filename))
  (vector-ref mat i))

;  Display i-th row of the matrix in filename

(define (row filename i)
 (display-vector (ro filename i)))

;  Return j-th col of the matrix in filename

(define (co filename j)
  (define mat (read-matrix filename))
  (define nrow (vector-length mat))
  (define column (make-vector nrow))
  (do ((i 0 (add1 i)))
      ((>= i nrow) column)
      (vector-set! column i (vector-ref (vector-ref mat i) j ))))

;  Display j-th col of the matrix in filename

(define (col filename j)
 (display-vector (co filename j)))
```

```
PROBLEM #2
==========

;  Return the dot product v1 * v2

(define (dot-product v1 v2)
  (do ((i 0 (add1 i)) (sum 0 (+ sum (* (vector-ref v1 i)
                                       (vector-ref v2 i)))))
  ((>= i (vector-length v1)) sum)))

;  Matrix multiplication, display and create f3

(define (mmul f1 f2 f3)
  (define m1 (read-matrix f1))
  (define m2 (read-matrix f2))
  (define nrow (vector-length m1))
  (define ncol (vector-length m2))
  (define outport (open-output-file f3))
  (display nrow outport) (display " " outport)
  (display ncol outport) (newline outport)
  (do ((i 0 (add1 i)))
      ((>= i nrow) (close-output-port outport) (display ""))
      (let ((row (make-vector ncol)))
      (do ((j 0 (add1 j)))
          ((>= j ncol) (display-vector row)(newline) (newline outport))
          (vector-set! row j (dot-product (ro f1 i)(co f2 j)))
          (display (vector-ref row j) outport)(display " " outport)))))


[2] (read-matrix "matrix1.dat")
#(#(1 2 3) #(4 5 6))

[3] (read-matrix "matrix2.dat")
#(#(1 2 3) #(1 2 3) #(1 2 3))

[4] (row "matrix1.dat" 0)
1 2 3

[5] (row "matrix1.dat" 1)
4 5 6

[6] (col "matrix1.dat" 0)
1 4

[7] (col "matrix1.dat" 1)
2 5

[8] (col "matrix1.dat" 2)
3 6

[9] (mmul "matrix1.dat" "matrix2.dat" "matrix3.dat")
6 12 18
15 30 45

[10] (read-matrix "matrix3.dat")
#(#(6 12 18) #(15 30 45))
```

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS

KHANH SUCKS