```
;;; (skex bina '(66    (11 22 33 44 55 66 77 88 99 1010) ) )

(:= bina   ;;; binary-search using an internal array
  '(def-block ( x   (a (1 10)) ) ;;; input
             ( r )   ;;; output
    (:= k 1)
    (:= m 10)
    (loop lup1    (unroll  10 no-back-jump)
          (while (<= k m))
          (do
             (:= j (// (+ k m)  2))
             (if (= x (a j)) (then (:= r j) (leave lup1 j)))
             (if (< x (a j)) (then (:= m (- j 1)))
             (else (:= k (+ j 1)))))
          (result  (:= r  0)))

    )))


;;; (skex bin '(66    11 22 33 44 55 66 77 88 99 1010 ) )

(:= bin         ;;; w/o array
  '(def-block ( x   a1 a2 a3 a4 a5 a6 a7 a8 a9 a10) ;;; input
             ( r )   ;;; output
   (:= r -1)
   (if (= x a5) (then (:= r 5))
   (else (if (< x a5) (then   ;;; x < a5
             (if (= x a2) (then (:= r 2))
             (else (if (< x a2) (then  ;;; x < a2
                        (if (= x a1) (then (:= r 1))
                        (else (:= r 0))))
                   (else (if (= x a4) (then (:= r 4))
                         (else (if (< x a4) (then ;;;  a2 < x < a4
                                    (if (= x a3) (then (:= r 3))
                                    (else (:= r 0))))
                               (else (:= r 0))))))))))     ;;; a4 < x < a5

         (else (if (= x a7) (then (:= r 7))  ;;;  a5 < x <? a10
               (else (if (< x a7) (then
                         (if (= x a6) (then (:= r 6))
                         (else (:= r 0))))
                     (else (if (= x a9) (then (:= r 9))
                           (else (if (< x a9) (then
                                     (if (= x a8) (then (:= r a8))
                                     (else (:= r 0))))
                                 (else (if (= x a10) (then (:= r 10))
                                       (else (:= r 0)))))))))))))))
   ))
```

```
;*** Build the latest version of SKEX
;***

(remprop 'loop 'build-information)      ;*** Get rid of Yale Loop info.

(:= *build-module-list* () )

(load 'utilities:build)                 ;*** This must go first.
(load 'interpreter:build)
(load 'trace:build)
(load 'ideal-code-generator:build)
(load 'diophantine:build)
(load 'flow-analysis:build)

(:= *exp.build-module-list* '(
    experiments:skex-options
    experiments:skex
    ) )

(:= *build-module-list* (append *build-module-list* *exp.build-module-list*) )

(build)
```

```
;;; Convolution, with k weights and vectors of length n.

;;; (skex convo1 convo-args)

(:= convo1 '
(def-block ( N                    ;;; 'real' size of N  (max 128)
             K                    ;;; 'real' size of K  (max 32)
             (x (1 128) )
             (y (1 128) )
             (w (1 32)   ) )
           ( (y (1 128) ) )


  (loop
     (incr i from 1 to n)
     (unroll 4)
     (do
      (loop
         (incr j from 1 to k)
         (unroll 8 fold-step-vars no-back-jump no-exit-tests)
         (do
            (:= (y i) (+$ (y i) (*$ (w j) (x (- (+ i j) 1)))))) ) ) ) ) )





(:= convo-args
    '(
      64
       8


      (1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0)

      (0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
       0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
       0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
       0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
       0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
       0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
       0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
       0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0)

      (1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0
       1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0 1.0 0.0 3.0 4.0) ) )
```

```
;  (skex delay1)
(:= delay1
    '(def-block

        nil                                          ;;; no input
        (P)
                                                     ;;; outputs via esc


        (declare P (0 5))

        (:= a (+ b c))
        (:= d (+ e a))
        (:= g (+ f d))
        (:= i (+ h g))
        (:= x (+ y i))
        (:= z (+ x x))
        (:= u (+ z z))
        (:= v (+ u u))
        (:= w (+ v v))
        (:= t (+ w w))

        (:= j a)
        (:= (P j) (+ t 1))
        (:= j1 (+ j 1))
        (:= (P j1) (+ t 1))
        (:= j2 (+ j1 1))
        (:= (P j2) (+ t 1))
        (:= j3 (+ j2 1))
        (:= (P j3) (+ t 1))
        (:= j4 (+ j3 1))
        (:= (P j4) (+ t 1))
        (:= j5 (+ j4 1))
        (:= (P j5) (+ t 1))

                                   ))
```

```
;==========================================================================
;
; Dotproduct
;
; (SKEX DOTPROD1 DOTPROD-ARGS)
; (SKEX DOTPROD2 DOTPROD-ARGS)
; (SKEX DOTPROD4 DOTPROD-ARGS)
; (SKEX DOTPROD5 DOTPROD-ARGS)
; (SKEX (DOTPROD.UNROLL n) DOTPROD-ARGS)
;
;==========================================================================

(:= dotprod-args
    '((5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0
       4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0
       3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0
       7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4
       7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4
       9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0
       8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1
       7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1
       7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0)

      (3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0
       7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0 7.0
       7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4
       5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0
       4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0 4.0
       9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0 9.0
       8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1 8.1
       7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1 7.1
       7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4 7.4) ) )

(:= *dotprod.template* '

(def-block ( (a (1 135) )
             (b (1 135) ) )

           (tot)

    (:= tot 0.0)
    (loop (incr u from 1 to 135)
          (unroll *unroll*)
    (do
        (:= tot (+$ tot (*$ (a u) (b u) ) ) ) ) )
    )
)


(defun dotprod.unroll ( n )
    (subst n '*unroll* *dotprod.template*) )


(:= dotprod1 (dotprod.unroll 1) )
(:= dotprod2 (dotprod.unroll 2) )
(:= dotprod4 (dotprod.unroll 4) )
(:= dotprod5 (dotprod.unroll 5) )
```

```
;; Really wide program intended to break everything's limits:

;;;  (SKEX FAT)

(:= fat '(def-block
          nil
          (B00 B01 B02 B03 B04 B05 B06 B07 B08 B09
           B10 B11 B12 B13 B14 B15 B16 B17 B18 B19
           B20 B21 B22 B23 B24 B25 B26 B27 B28 B29
           B30 B31 B32 B33 B34 B35 B36 B37 B38 B39
           B40 B41 B42 B43 B44 B45 B46 B47 B48 B49
           B50 B51 B52 B53 B54 B55 B56 B57 B58 B59
           B60 B61 B62 B63 B64 B65 B66 B67 B68 B69
           B70 B71 B72 B73 B74 B75 B76 B77 B78 B79
           B80 B81 B82 B83 B84 B85 B86 B87 B88 B89
           B90 B91 B92 B93 B94 B95 B96 B97 B98 B99
                                                   )  ;;; OUTPUTS!

(:= B05(+$ 3.0   4.0))(:= B06(+$ 3.0 4.0))(:= B07(+$ 3.0 4.0))(:= B08(+$ 3.0 4.0
(:= B15(+$ 3.0   4.0))(:= B16(+$ 3.0 4.0))(:= B17(+$ 3.0 4.0))(:= B18(+$ 3.0 4.0
(:= B25(+$ 3.0   4.0))(:= B26(+$ 3.0 4.0))(:= B27(+$ 3.0 4.0))(:= B28(+$ 3.0 4.0
(:= B35(+$ 3.0   4.0))(:= B36(+$ 3.0 4.0))(:= B37(+$ 3.0 4.0))(:= B38(+$ 3.0 4.0
(:= B45(+$ 3.0   4.0))(:= B46(+$ 3.0 4.0))(:= B47(+$ 3.0 4.0))(:= B48(+$ 3.0 4.0
(:= B55(+$ 3.0   4.0))(:= B56(+$ 3.0 4.0))(:= B57(+$ 3.0 4.0))(:= B58(+$ 3.0 4.0
(:= B65(+$ 3.0   4.0))(:= B66(+$ 3.0 4.0))(:= B67(+$ 3.0 4.0))(:= B68(+$ 3.0 4.0
(:= B75(+$ 3.0   4.0))(:= B76(+$ 3.0 4.0))(:= B77(+$ 3.0 4.0))(:= B78(+$ 3.0 4.0
(:= B85(+$ 3.0   4.0))(:= B86(+$ 3.0 4.0))(:= B87(+$ 3.0 4.0))(:= B88(+$ 3.0 4.0
(:= B95(+$ 3.0   4.0))(:= B96(+$ 3.0 4.0))(:= B97(+$ 3.0 4.0))(:= B98(+$ 3.0 4.0
(:= B00(+$ 3.0   4.0))(:= B01(+$ 3.0 4.0))(:= B02(+$ 3.0 4.0))(:= B03(+$ 3.0 4.0
(:= B10(+$ 3.0   4.0))(:= B11(+$ 3.0 4.0))(:= B12(+$ 3.0 4.0))(:= B13(+$ 3.0 4.0
(:= B20(+$ 3.0   4.0))(:= B21(+$ 3.0 4.0))(:= B22(+$ 3.0 4.0))(:= B23(+$ 3.0 4.0
(:= B30(+$ 3.0   4.0))(:= B31(+$ 3.0 4.0))(:= B32(+$ 3.0 4.0))(:= B33(+$ 3.0 4.0
(:= B40(+$ 3.0   4.0))(:= B41(+$ 3.0 4.0))(:= B42(+$ 3.0 4.0))(:= B43(+$ 3.0 4.0
(:= B50(+$ 3.0   4.0))(:= B51(+$ 3.0 4.0))(:= B52(+$ 3.0 4.0))(:= B53(+$ 3.0 4.0
(:= B60(+$ 3.0   4.0))(:= B61(+$ 3.0 4.0))(:= B62(+$ 3.0 4.0))(:= B63(+$ 3.0 4.0
(:= B70(+$ 3.0   4.0))(:= B71(+$ 3.0 4.0))(:= B72(+$ 3.0 4.0))(:= B73(+$ 3.0 4.0
(:= B80(+$ 3.0   4.0))(:= B81(+$ 3.0 4.0))(:= B82(+$ 3.0 4.0))(:= B83(+$ 3.0 4.0
(:= B90(+$ 3.0   4.0))(:= B91(+$ 3.0 4.0))(:= B92(+$ 3.0 4.0))(:= B93(+$ 3.0 4.0

                                                                         ))
```

```
;***
;*** FFT adpated from "Introduction to Discrete Systems" by Kenneth Steiglitz
;***
;*** See FFT.FOR and FFT1.FOR for some meaningful FORTRAN source.
;***
;***
;*** (SKEX FFT1 FFT-ARGS)
;*** (SKEX FFT2 FFT-ARGS)
;*** (SKEX FFT4 FFT-ARGS)
;*** (SKEX (FFT.UNROLL 8)  FFT-ARGS)
;*** (SKEX (FFT.UNROLL 16) FFT-ARGS)
;***

(:= fft-args
    '( ( 0.000000  0.382683  0.707106  0.923879  1.000000
         0.923879  0.707106  0.382683 -0.000000 -0.382683
        -0.707107 -0.923879 -1.000000 -0.923879 -0.707106
        -0.382682  0.000000  0.382684  0.707107  0.923879
         1.000000  0.923879  0.707106  0.382682 -0.000001
        -0.382684 -0.707107 -0.923880 -1.000000 -0.923879
        -0.707105 -0.382682)
       32) )

; Correct answer for FFT-ARGS:
;
;      1 = 7.0430812E-7
;      2 = 2.817241E-6
;      3 = 15.9999999
;      4 = 4.090138E-6
;      5 = 2.56225857E-6
;      6 = 1.5592826E-6
;      7 = 1.06658226E-6
;      8 = 1.08584835E-6
;      9 = 9.5298865E-7
;      10 = 8.9607337E-7
;      11 = 9.8725758E-7
;      12 = 7.64804715E-7
;      13 = 6.7512326E-7
;      14 = 7.61604667E-7
;      15 = 1.77594757E-7
;      16 = 7.9946803E-7
;      17 = 6.745058E-7
;      18 = 7.99468255E-7
;      19 = 8.858079E-7
;      20 = 7.616044E-7
;      21 = 6.7512319E-7
;      22 = 7.6480475E-7
;      23 = 5.9356097E-7
;      24 = 8.9607311E-7
;      25 = 9.5298859E-7
;      26 = 1.0858484E-6
;      27 = 1.08642147E-6
;      28 = 1.55928223E-6
;      29 = 2.56225836E-6
;      30 = 4.09013754E-6
;      31 = 15.9999992
;      32 = 2.81724155E-6
;
;
;


(:= *fft.template* '
```

```
(def-block ( (s (1 256) )
             n )
           ( (r (1 256) ) )

    (declare fr (1 256) )
    (declare fi (1 256) )

    (loop (incr i from 1 to n)
          (unroll *unroll*)
    (do
        (:= j (bit-reverse (- i 1) n) )
        (:= (fr i) (s (+ j 1) ) )
        (:= (fi i) 0.0) ) )

    (loop (step length from 2 using (+ length length) while (<= length n) )
    (do
        (loop (incr j from 1 to n by length)
        (do

            (loop (incr l from 1 to (// length 2) )
                  (unroll *unroll*)
            (do
                (assert (>= l 1) )
                (assert (<= l (// length 2) ) )

                (:= emjtr (cos (//$ (*$ (float (- l 1) ) -6.283185)
                                    (float length) ) ) )
                (:= emjti (sin (//$ (*$ (float (- l 1) ) -6.283185)
                                    (float length) ) ) )
                (:= loc1 (+ l (- j 1) ) )
                (:= loc2 (+ loc1 (// length 2) ) )
                (:= zr (-$ (*$ emjtr (fr loc2) ) (*$ emjti (fi loc2) ) ) )
                (:= zi (+$ (*$ emjtr (fi loc2) ) (*$ emjti (fr loc2) ) ) )
                (:= (fr loc2) (-$ (fr loc1) zr) )
                (:= (fi loc2) (-$ (fi loc1) zi) )
                (:= (fr loc1) (+$ (fr loc1) zr) )
                (:= (fi loc1) (+$ (fi loc1) zi) ) ) )
            ) )
        ) )

    (loop (incr i from 1 to n)
          (unroll *unroll*)
    (do
        (:= (r i) (sqrt (+$ (*$ (fr i) (fr i) )
                            (*$ (fi i) (fi i) ) ) ) ) ) )

    )
)


(defun fft.unroll ( n )
    (subst n '*unroll* *fft.template*) )

(:= fft1 (fft.unroll 1) )
(:= fft2 (fft.unroll 2) )
(:= fft4 (fft.unroll 4) )


;***
;*** Function for generating test values
;***
```

```
defun fft.gen-vals ( n )
   (loop (initial result () )
         (incr i from 1 to n)
   (do
       (push result (sin (* (- i 1) (// 3.14159 8.0) ) ) ) )
   (result
       (dreverse result) ) ) )
```

```
;==========================================================================
;
; Natural Log
;
; Using the Maclaurin Series for the natural log of x, we have two
; parameters, x (where 0 < x < 2 ), and k, the number of terms summed.
;
; (SKEX LN1  LN-ARGS)
; (SKEX LN2  LN-ARGS)
; (SKEX LN4  LN-ARGS)
; (SKEX LN20 LN-ARGS)
; (SKEX (LN.UNROLL n) LN-ARGS)
;
;==========================================================================

(:= ln-args '(1.8 30) )

(:= *ln.template* '

(def-block ( x k ) ( log )
    (:= y (-$ 1.0 x) )
    (loop (unroll *unroll*)
          (incr n from 1 to k)
          (step n-real     from  0.0
                           using (+$ n-real 1.0) )
          (step numerator from  -1.0
                           using (*$ numerator y) )
          (step log        from  0.0
                           using (+$ log (//$ numerator n-real) ) ) )
    )
)

(defun ln.unroll ( n )
    (subst n '*unroll* *ln.template*) )

(:= ln1  (ln.unroll 1) )
(:= ln2  (ln.unroll 2) )
(:= ln4  (ln.unroll 4) )
(:= ln20 (ln.unroll 20) )
```

```
;==============================================================================
;
; Matrix Multiply
;
; (SKEX MAMTUL1 MATMUL-ARGS)
; (SKEX MAMTUL2 MATMUL-ARGS)
; (SKEX MAMTUL4 MATMUL-ARGS)
;
;==============================================================================


(:= matmul-args
    '( (1.0 0.0 3.0 4.0
        2.0 2.0 0.0 0.0
        0.0 0.0 3.0 1.0
        0.0 0.0 0.0 0.0
        ) (
        4.0 4.0 4.0 4.0
        3.0 3.0 3.0 3.0
        2.0 2.0 2.0 2.0
        1.0 1.0 1.0 1.0) ) )

(:= *matmul.template* '
(def-block ( (a (1 4) (1 4) )
             (b (1 4) (1 4) ) )
           ( (p (1 4) (1 4) ) )

    (loop (incr u from 1 to 4)
    (do
        (loop (incr v from 1 to 4)
              (unroll *unroll*)
        (do
            (:= sum 0.0)
            (loop (incr i from 1 to 4)
                  (unroll 4 no-back-jump)
            (do
                (:= sum (+$ sum (*$ (a u i) (b i v) ) ) ) ) ) )
            (:= (p u v) sum) ) ) ) )
    )
)


(defun matmul.unroll ( n )
    (subst n '*unroll* *matmul.template*) )

(:= matmul1 (matmul.unroll 1) )
(:= matmul2 (matmul.unroll 2) )
(:= matmul4 (matmul.unroll 4) )
```

```
;============================================================================
;
; Silly Tiny-Lisp test program to find the max of 16 numbers using a tree of
; comparisons.  The 16 is VERY hardwired into this.  This was chosen to get
; a lot of milage out of the bookkeeper and make it likely that paths other
; than the main will be exercised...   This is really pathological and ought
; to stand a good chance of breaking the compactor.
;
;  (SKEX MAX MAX-ARGS)
;
;============================================================================

(:= max-args  '( ( 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1) ) )
(:= max-args1 '( ( 10 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 ) ) )
(:= max-args2 '( ( 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 40 ) ) )
(:= max-args3 '( ( 1 1 1 1 2 2 2 20 20 3 3 3 1 1 1 1 ) ) )

(:= max '
(def-block ( (a 16) )
           ( max )

    (:= max0  (if (> (a 0)  (a 1) )
                 (then (a 0) )
                 (else (a 1) ) )  )

    (:= max1  (if (> (a 2)  (a 3) )
                 (then (a 2) )
                 (else (a 3) ) )  )

    (:= max2  (if (> (a 4)  (a 5) )
                 (then (a 4) )
                 (else (a 5) ) )  )

    (:= max3  (if (> (a 6)  (a 7) )
                 (then (a 6) )
                 (else (a 7) ) )  )

    (:= max4  (if (> (a 8)  (a 9) )
                 (then (a 8) )
                 (else (a 9) ) )  )

    (:= max5  (if (> (a 10) (a 11))
                 (then (a 10))
                 (else (a 11)) )  )

    (:= max6  (if (> (a 12) (a 13))
                 (then (a 12))
                 (else (a 13)) )  )

    (:= max7  (if (> (a 14) (a 15))
                 (then (a 14))
                 (else (a 15)) )  )


    (:= max8  (if (> max0  max1 )
                 (then max0 )
                 (else max1 ) )  )

    (:= max9  (if (> max2  max3 )
                 (then max2 )
                 (else max3 ) )  )

    (:= max10 (if (> max4  max5 )
```

```
                 (then max4 )
                 (else max5 ) )  )

    (:= max11 (if (> max6  max7 )
                 (then max6 )
                 (else max7 ) )  )

    (:= max12 (if (> max8  max9 )
                 (then max8 )
                 (else max9 ) )  )

    (:= max13 (if (> max10 max11)
                 (then max10)
                 (else max11) )  )

    (:= max   (if (> max12 max13)
                 (then max12)
                 (else max13) )  )
    )
)
```

```
;;; Uses sieve to find primes from 1-100.  Should have very little compaction.
;==============================================================================
;
; PRIME
;
; Uses a seive to calculate the primes between 1 and 100.
;
; (SKEX PRIME1)
; (SKEX PRIME2)
; (SKEX PRIME4)
; (SKEX (PRIME.UNROLL N) )
;
;==============================================================================

(:= *prime.template* '

(def-block ()
           ( list-length
             (list (1 30) (initial 0) ) )
    (declare p (1 100) (initial 0) )

    (loop (incr j from 2 to 10)
    (do
        (assert (>= j 2) )
        (assert (<= j 10) )
        (loop (incr k from j)
              (unroll *unroll*)
              (while (<=  (* j k) 100) )
        (do
            (:= (p (* j k) ) 1) ) ) ) ) )

    (:= list-length 0)
    (loop (incr i from 1 to 100)
          (unroll *unroll*)
    (do
        (if (= (p i) 0) (then
            (:= list-length (+ 1 list-length) )
            (:= (list list-length) i) ) ) ) )
    )
)

(defun prime.unroll ( n )
    (subst n '*unroll* *prime.template*) )

(:= prime1 (prime.unroll 1) )
(:= prime2 (prime.unroll 2) )
(:= prime4 (prime.unroll 4) )
```

```
;==============================================================================
;
; Top Level Driver for SKeduling EXperiments.
;
;==============================================================================

(defvar *skex.code* () )            ;*** The most recent tiny-lisp source code.
(defvar *skex.actual-params* () ) ;*** The most recent actual paramaters.
(defvar *skex.seq-naddr-unoptimized* () )
                                   ;*** The most recent sequential naddr
                                   ;***   from *SKEX.CODE*.
(defvar *skex.seq-naddr-optimized*  () )
                                   ;*** Optimized version of the above.
(defvar *skex.par-naddr* () )      ;*** The most recent parallel naddr.
(defvar *skex.timed-functions* () )
                                   ;*** List of functions to be timed.

(defvar *int.operation-count*)     ;*** From INTERPRETER.
(defvar *int.instruction-count*)   ;*** From INTERPRETER.

(defvar *number-of-banks*)         ;*** Number of memory banks.


(declare
    (lexpr time-functions)
    (lexpr options.print)
    (lexpr compile-tiny-lisp)
    (lexpr print-functions-times)
    (lexpr interpret)
    )


;***
;*** Options.  Do (OPTIONS.HELP) for a description of each option and its
;*** current value.
;***

(declare (special                            ;*** From SKEX-OPTIONS
    *skex.time-functions?*
    *skex.eliminate-common-subexpressions?*
    *skex.move-loop-invariants?*
    *skex.disambiguator-tool?*
    *skex.compact?*
    ) )

(options.reset)

(:= *skex.timed-functions* '(

    compile-tiny-lisp
    interpret
    fg.analyze&optimize
    compact

;     bookkeep
;     generate-code
;
;     fg.naddr-to-flow-graph
;     fg.collect-names
;     fg.set-reaching-defs
```

1

PS:<C.S.BULLDOG.EXPERIMENTS>SKEX.LSP.17

```
;    fg.set-live-names
;    fg.set-dominators
;    fg.find-loops
;    fg.set-loop-invariants
;    fg.move-loop-invariants
;    fg.eliminate-common-subexpressions
;    fg.insert-loop-assignments
;    fg.disambiguator-tool
;
;    predecessors
;    stat:index-derivation
;
;    de:possibly-equal?
    ) )

;***
;*** Run a complete experiment on some Tiny Lisp code with some actual
;***    paramaters.
;***

(defun skex ( &optional (code          *skex.code*)
                        (actual-params *skex.actual-params*) )
    (skex.initialize)

    (:= *skex.code*          code)
    (:= *skex.actual-params* actual-params)

    (msg 0 t (e (options.print) ) 0 t)

    (unwind-protect
        (let ()

            (if *skex.time-functions?* (then
                (apply 'time-functions *skex.timed-functions*) ) )

            (:= *skex.seq-naddr-unoptimized*
                (compile-tiny-lisp *skex.code*) )

            (:= *skex.seq-naddr-optimized*
                (fg.analyze&optimize *skex.seq-naddr-unoptimized*) )

            (if *skex.compact?* (then
                (:= *skex.par-naddr*
                    (mis->pnaddr
                        (compact *skex.seq-naddr-optimized*) ) ) )
            (else
                (:= *skex.par-naddr* *skex.seq-naddr-optimized*)
                (:= *skex.seq-naddr-optimized* *skex.seq-naddr-unoptimized*)))

            (skex.run-programs)

            (if *skex.time-functions?* (then
                (msg 0 t (e (print-function-times) ) t) ) )

            () )

        (if *skex.time-functions?* (then
            (apply 'untime-functions *skex.timed-functions*) ) ) )
    () )


;***
;*** Initialize the SKEXing world.  This is also useful for getting rid
```

2

```
;*** of unwanted data structures, pointers, etc.
;***

(defun skex.initialize ()
    (tr.initialize)
    (initialize-code-generator)
    (fg.initialize)
    (de.initialize)

    (:= *skex.code*                   () )
    (:= *skex.actual-params*          () )
    (:= *skex.seq-naddr-unoptimized* () )
    (:= *skex.seq-naddr-optimized*    () )
    (:= *skex.par-naddr*              () )

    (let ( (old (gcgag t) ) )
        (gc)
        (gcgag old) )
    () )


;***
;*** Run the sequential and compacted code from the last experiment but
;*** with new actual paramaters.
;***

(defun skex-again ( actual-params )
    (:= *skex.actual-params* actual-params)
    (skex.run-programs) )


;***
;*** Execute the sequential and compacted code on the actual paramaters
;*** (all saved away in the global variables), and print some pretty
;*** statistics and the results of each execution.
;***

(defun skex.run-programs ()

(let ( (sequential-instructions 0)
       (parallel-instructions   0)
       (parallel-operations     0)
       (formal-params           () )
       (output-vars             () ) )
    (msg 0 t)

    (if (== 'def-block (caar *skex.seq-naddr-optimized*) ) (then
        (desetq (() formal-params output-vars) (car *skex.seq-naddr-optimized*)

    (interpret *skex.seq-naddr-optimized* *skex.actual-params*)
    (:= sequential-instructions *int.operation-count*)

    (msg t t "Uncompacted program results:" t)
    (int.print-program-variables output-vars)

    (interpret *skex.par-naddr* *skex.actual-params*)
    (:= parallel-operations   *int.operation-count*)
    (:= parallel-instructions *int.instruction-count*)

    (msg t "Compacted program results:" t)
    (int.print-program-variables output-vars)

    (msg t "Uncompacted instructions   = " sequential-instructions t)
```

```
    (msg   "Compacted operations       = " parallel-operations     t)
    (msg   "Compacted instructons      = " parallel-instructions   t)
    (msg   "Average # operations//cycle = "
           (// (flonum parallel-operations) (flonum parallel-instructions) )
           t)
    (msg   "Uncompacted//compacted      = "
           (// (flonum sequential-instructions) (flonum parallel-instructions))
           t)
    () ) )


;***
;*** Run a tiny lisp program.
;***

(defun skex.run-tiny-lisp ( code &optional actual-params )

(let ( (formal-params           () )
       (output-vars             () )
       (naddr                   () )
       (sequential-instructions () ) )
    (msg 0 t)

    (:= naddr (compile-tiny-lisp code) )

    (if (== 'def-block (caar naddr) ) (then
        (desetq (() formal-params output-vars) (car naddr) ) ) )

    (interpret naddr actual-params)
    (:= sequential-instructions *int.operation-count*)

    (msg 0  *int.instruction-count* " instructions." t)
    (msg "Uncompacted program results:" t)
    (int.print-program-variables output-vars)
    () ) )
```

```
;===========================================================================
;
; SKEX OPTIONS
;
; This module contains the definitions of top level options dealing with
; the whole compiler (SKEX).
;
;===========================================================================

(eval-when (compile)
    (build '(utilities:options) ) )


(def-option *skex.time-functions?* () experiments: "
If T then all the functions listed in *SKEX.TIME-FUNCTIONS* are timed and
statistics printed out at the end of each SKEX run.
")


(def-option *skex.compact?* t experiments: "
If T then the NADDR program is compiled using trace-scheduling and the
currently loaded codegenerator; the optimized sequential NADDR is compared
with the compacted parallel NADDR.  If () compacting is not invoked, and
the TinyLisp generated NADDR is compared with the optimized sequential
NADDR.
")


(def-option *number-of-banks* 8 experiments: "
This variable contains the number of memory banks to compile for.
")
```

```
;==========================================================================
;
; LU DECOMPOSITION SOLVER.  From Forsythe & Moler.
;
; Given A & b, solves Ax=b by decomposing A into L & U, then solving the
; easy resultant triangular systems Ly=b and Ux=y.  L is lower triangular,
; U upper.
;
; (SKEX SOLVE1 SOLVE-ARGS)
; (SKEX SOLVE2 SOLVE-ARGS)
; (SKEX SOLVE4 SOLVE-ARGS)
; (SKEX (SOLVE.UNROLL 16) SOLVE-ARGS)
;
;==========================================================================

(:= solve-args '(
    (8.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 7.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 9.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 4.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 3.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 9.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 5.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 9.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 4.0 1.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 9.0 1.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 9.0 1.0
     1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0)


    (2.0 7.0 3.0 6.0 6.0 6.0 2.0 3.0 7.0 2.0 1.0 5.0 8.0 4.0 9.0 8.0) ) )


(:= *solve.template* '

(def-block ( (a  (1 16) (1 16) )
             (b  (1 16) ) )

           ( (x  (1 16) )
             (ax (1 16) ) )              ;*** Ax should = b.

    (declare lu     (1 16) (1 16) )
    (declare ps     (1 16) )
    (declare scales (1 16) )


        ;*** Decomposition:  Finds LU, the upper tri & lower tri.
        ;*** decompostion matricies combined.
        ;***
        ;*** Initialize lu to a, scales to 1 over the absolute max of
        ;*** the row, ps(i) to i
        ;
    (loop (incr i from 1 to 16) (do

        (:= (ps i) i)
        (:= normrow 0.0)
        (loop (incr j from 1 to 16)
              (unroll *unroll*)
        (do
```

1

```
            (:= (lu i j) (a i j) )
            (:= normrow (max$ normrow (abs$ (lu i j) ) ) ) ) )

        (if (!=$ normrow 0.0) (then
            (:= (scales i) (//$ 1.0 normrow) ) )
        (else
            (esc (format t "~%Singularity in a row.") ) ) ) ) )

        ;*** Gaussian elimination with partial pivoting:
        ;
    (loop (incr k from 1 to (- 16 1) ) (do

        (:= biggest 0.0)
        (:= pividx  0)
        (loop (incr i from k to 16)
              (unroll *unroll*)
        (do
            (:= size (*$ (abs$ (lu (ps i) k) ) (scales (ps i) ) ) )
            (if ( > size biggest) (then
                (:= biggest size)
                (:= pividx i) ) ) ) )

        (if (=$ 0.0 biggest) (then
            (esc (format t "~%Singular matrix.") ) ) )

        (if (!= pividx k) (then
            (assert (!= pividx k) )
            (:= j            (ps k) )
            (:= (ps k)       (ps pividx) )
            (:= (ps pividx) j) ) )

        (:= pivot (lu (ps k) k) )

        (loop (incr i from (+ k 1) to 16) (do
            (:= mult (//$ (lu (ps i) k) pivot) )
            (:= (lu (ps i) k) mult)
            (if (!=$ mult 0.0) (then
                (assert (!= (ps i) (ps k) ) )
                (loop (incr j from (+ k 1) to 16)
                      (unroll *unroll*)
                (do
                    (:= (lu (ps i) j)
                        (-$ (lu (ps i) j) (*$ mult (lu (ps k) j) ) ) ))))))))


    (if (=$ 0.0 (lu (ps 16) 16) ) (then
        (esc (format t "~%Singular matrix.") ) ) )


        ;*** SOLVE: Using the LU found in DECOMPOSE, solves the linear
        ;*** equation Ax=b.
        ;
    (loop (incr i from 1 to 16) (do
        (:= dot 0.0)
        (loop (incr j from 1 to (- i 1) )
              (unroll *unroll*)
        (do
            (:= dot (+$ dot (*$ (lu (ps i) j) (x j) ) ) ) ) )
        (:= (x i) (-$ (b (ps i) ) dot) ) ) )

    (loop (decr i from 16 to 1) (do
        (:= dot 0.0)
        (loop (incr j from (+ 1 i) to 16)
```

2

```
               (unroll *unroll*)
          (do
              (:= dot (+$ dot (*$ (lu (ps i) j) (x j) ) ) ) ) )
          (:= (x i) (//$ (-$ (x i) dot) (lu (ps i) i) ) ) ) )


          ;*** For a check, we find Ax and see if it's equal to b...
          ;
      (loop (incr i from 1 to 16)
      (do
          (:= tot 0.0)

          (loop (incr j from 1 to 16)
               (unroll *unroll*)
          (do
              (:= tot (+$ tot (*$ (a i j) (x j) ) ) ) ) )

          (:= (ax i) tot) ) )

      ) )


(defun solve.unroll ( n )
    (subst n '*unroll* *solve.template*) )


(:= solve1 (solve.unroll 1) )
(:= solve2 (solve.unroll 2) )
(:= solve4 (solve.unroll 4) )
```

```
;(skex.run-tiny-lisp ins '(1 10 (5 3 4 6 2 1 10 9 8 7) ) )

(:= ins
    '(def-block (h n (l (1 10) ) )
                (  (l (1 10)) )
        (loop (incr j from 2 to n) (do
            (:= key (l j))
            (:= k
                (loop lup (step i from (- j h)  using (- i h)
                                 while (>  i 0) )
                       (unroll 2)
                (do
                    (if (<= (l i) key) (then
                        (:= (l (+ h i) ) (l i) ) )
                    (else
                        (leave lup i) ) ) ) ) )
            (:= (l (+ k h)) key) ) ) ) )
                .

(:= ins
    '(def-block (h n (l (1 10) ) )
                (  (l (1 10)) )
        (loop (incr j from 2 to n) (do
            (:= key (l j))
            (loop lup
                  (step i from (- j h)  using (- i h)
                          while (if (> i 0)
                                (then (<= (l i) key) )
                                (else 0) ) )
                  (result-live i)
                  (unroll 2)
            (do
                (:= (l (+ h i) ) (l i) ) ) )
            (:= (l (+ i h)) key) ) ) ) )
```

```
;==============================================================================
;
; Square Root
;
; Using Newton's method.
;
; (SKEX SQRT1  '(15) )
; (SKEX SQRT2  '(15) )
; (SKEX SQRT4  '(15) )
; (SKEX SQRT10 '(15) )
; (SKEX (SQRT.UNROLL n) '(15) )
;
;==============================================================================

(:= *sqrt.template* '
(def-block ( x ) ( zi )
    (:= zi-init (if (<= x 4.0) (then x) (else (//$ x 2.0) ) ) )
    (loop 1
        (unroll *unroll*)
        (step zi-old from  x
                     using zi)
        (step zi     from  zi-init
                     using (-$ zi (//$ (-$ (*$ zi zi) x)
                                       (+$ zi zi) ) ) )
        (while (>$ (-$ zi-old zi) 0.0000000001) ) )
    )
)

(defun sqrt.unroll ( n )
    (subst n '*unroll* *sqrt.template*) )

(:= sqrt1  (sqrt.unroll 1) )
(:= sqrt2  (sqrt.unroll 2) )
(:= sqrt4  (sqrt.unroll 4) )
(:= sqrt10 (sqrt.unroll 10) )
```

```
;***
;*** This is designed to show off the variable folder.
;***

;;; TRANSPOSE1 doesn't do any unrolling at all


(comment  ;;; Stuff me please.

(skex.set-option 'eliminate-common-subexpressions)

(skex transpose1 '( (
                     1.0 0.0 3.0 4.0    1.0 0.0 4.0 4.0
                     2.0 2.0 0.0 0.0    2.0 2.0 3.0 3.0
                     0.0 0.0 3.0 1.0    0.0 0.0 2.0 2.0
                     0.0 0.0 0.0 0.0    0.0 0.0 1.0 1.0

                     4.0 4.0 4.0 4.0    4.0 4.0 3.0 4.0
                     3.0 3.0 3.0 3.0    3.0 3.0 0.0 0.0
                     2.0 2.0 2.0 2.0    2.0 2.0 3.0 1.0
                     1.0 1.0 1.0 1.0    1.0 1.0 0.0 0.0
                    ) )  )

endcomment)

(:= transpose1 '
(def-block ( (a  (1 8) (1 8) ) )
           ( (at (1 8) (1 8) ) )

    (loop (incr u from 1 to 8)
     (do
         (loop (incr v from 1 to 8)
               (do
                   (:= (at v u) (a u v))))))))

;****************
;**
;**
;** TRANSPOSE2 does the unroll keywords to the hilt.
;**
;**

(comment  ;;; Stuff me please.

(skex.set-option 'eliminate-common-subexpressions)

(skex transpose2 '( (
                     1.0 0.0 3.0 4.0    1.0 0.0 4.0 4.0
                     2.0 2.0 0.0 0.0    2.0 2.0 3.0 3.0
                     0.0 0.0 3.0 1.0    0.0 0.0 2.0 2.0
                     0.0 0.0 0.0 0.0    0.0 0.0 1.0 1.0

                     4.0 4.0 4.0 4.0    4.0 4.0 3.0 4.0
                     3.0 3.0 3.0 3.0    3.0 3.0 0.0 0.0
                     2.0 2.0 2.0 2.0    2.0 2.0 3.0 1.0
                     1.0 1.0 1.0 1.0    1.0 1.0 0.0 0.0
                    ) )  )

endcomment)

(:= transpose2 '
(def-block ( (a  (1 8) (1 8) ) )
```

```
           ( (at (1 8) (1 8) ) )

    (loop (incr u from 1 to 8)
          (unroll 8 completely! no-back-jump fold-step-vars)
          (do
           (loop (incr v from 1 to 8)
                 (unroll 8 completely! no-back-jump fold-step-vars)
                 (do
                  (:= (at v u) (a u v))))))))

;****************
;**
;**
;** TRANSPOSE3 does unrolling without my neat new keywords.  No sense in
;**           unrolling the outer loop, traces won't go beyond.
;**

(comment  ;;; Stuff me please.

(skex.set-option 'eliminate-common-subexpressions)

(skex transpose3 '( (
                     1.0 0.0 3.0 4.0    1.0 0.0 4.0 4.0
                     2.0 2.0 0.0 0.0    2.0 2.0 3.0 3.0
                     0.0 0.0 3.0 1.0    0.0 0.0 2.0 2.0
                     0.0 0.0 0.0 0.0    0.0 0.0 1.0 1.0

                     4.0 4.0 4.0 4.0    4.0 4.0 3.0 4.0
                     3.0 3.0 3.0 3.0    3.0 3.0 0.0 0.0
                     2.0 2.0 2.0 2.0    2.0 2.0 3.0 1.0
                     1.0 1.0 1.0 1.0    1.0 1.0 0.0 0.0
                    ) )  )

endcomment)

(:= transpose3 '
(def-block ( (a  (1 8) (1 8) ) )
           ( (at (1 8) (1 8) ) )

    (loop (incr u from 1 to 8)
          (do
           (loop (incr v from 1 to 8)
                 (unroll 8  )
                 (do
                  (:= (at v u) (a u v))))))))

;****************
;**
;**
;** TRANSPOSE4 is a silly attempt at doing a 12x12 transpose.  What a fool.
;**
;**

(comment  ;;; Stuff me please.

(skex.set-option 'eliminate-common-subexpressions)

(skex transpose4 '( (
                     1.0 0.0 3.0 4.0    1.0 0.0 4.0 4.0    1.0 0.0 4.0 4.0
                     2.0 2.0 0.0 0.0    2.0 2.0 3.0 3.0    2.0 2.0 3.0 3.0
                     0.0 0.0 3.0 1.0    0.0 0.0 2.0 2.0    0.0 0.0 2.0 2.0
                     0.0 0.0 0.0 0.0    0.0 0.0 1.0 1.0    0.0 0.0 1.0 1.0
```

```
                4.0 4.0 4.0 4.0   4.0 4.0 3.0 4.0   4.0 4.0 3.0 4.0
                3.0 3.0 3.0 3.0   3.0 3.0 0.0 0.0   3.0 3.0 0.0 0.0
                2.0 2.0 2.0 2.0   2.0 2.0 3.0 1.0   2.0 2.0 3.0 1.0
                1.0 1.0 1.0 1.0   1.0 1.0 0.0 0.0   1.0 1.0 0.0 0.0

                4.0 4.0 4.0 4.0   4.0 4.0 3.0 4.0   4.0 4.0 3.0 4.0
                3.0 3.0 3.0 3.0   3.0 3.0 0.0 0.0   3.0 3.0 0.0 0.0
                2.0 2.0 2.0 2.0   2.0 2.0 3.0 1.0   2.0 2.0 3.0 1.0
                1.0 1.0 1.0 1.0   1.0 1.0 0.0 0.0   1.0 1.0 0.0 0.0
               ) )  )

endcomment)

(:= transpose4 '
(def-block ( (a  (1 12) (1 12) ) )
           ( (at (1 12) (1 12) ) )

    (loop (incr u from 1 to 12)
          (unroll 12 completely! no-back-jump fold-step-vars)
          (do
           (loop (incr v from 1 to 12)
                 (unroll 12 completely! no-back-jump fold-step-vars)
                 (do
                  (:= (at v u) (a u v))))))))



;****************
;**
;**
;** TRANSPOSE5 does the unroll keywords to the hilt, but not folding vars.
;**
;**

(comment  ;;; Stuff me please.

(skex.set-option 'eliminate-common-subexpressions)

(skex transpose5 '( (
                1.0 0.0 3.0 4.0   1.0 0.0 4.0 4.0
                2.0 2.0 0.0 0.0   2.0 2.0 3.0 3.0
                0.0 0.0 3.0 1.0   0.0 0.0 2.0 2.0
                0.0 0.0 0.0 0.0   0.0 0.0 1.0 1.0

                4.0 4.0 4.0 4.0   4.0 4.0 3.0 4.0
                3.0 3.0 3.0 3.0   3.0 3.0 0.0 0.0
                2.0 2.0 2.0 2.0   2.0 2.0 3.0 1.0
                1.0 1.0 1.0 1.0   1.0 1.0 0.0 0.0
               ) )  )

endcomment)

(:= transpose5 '
(def-block ( (a  (1 8) (1 8) ) )
           ( (at (1 8) (1 8) ) )

    (loop (incr u from 1 to 8)
          (unroll 8 completely! no-back-jump )
          (do
           (loop (incr v from 1 to 8)
                 (unroll 8 completely! no-back-jump )
                 (do
```

```
                  (:= (at v u) (a u v))))))))
```

```
;==============================================================================
;
; Transitive Closure
;
; The result should be all 1's.
;
; (SKEX TRCL TRCL-ARGS)
;
;==============================================================================

(:= trcl-args '( (0 1 0   0 0 1 1 0 0) ) )

(:= trcl '
(def-block ( (a (1 3) (1 3) ) )
           ( (a (1 3) (1 3) ) )

    (loop (incr k from 1 to 3) (do
        (loop  (incr i from 1 to 3)
        (do
            (if (= (a i k) 1) (then
                (loop (incr j from 1 to 3)
                      (unroll 3 no-back-jump)
                (do
                    (if (= (a k j) 1) (then
                        (:= (a i j) 1) ) ) ) ) ) ) ) ) ) )
    )
)
```