```
;*** (build *int.build-module-list*)
;*** (build.compile *int.build-module-list*)

(:= *int.build-module-list* '(

    interpreter:naddr
    interpreter:interpreter
    interpreter:compiler
    interpreter:compile-functions
    ) )

(:= *build-module-list* (append *build-module-list* *int.build-module-list*) )
```

```
;===============================================================================
;
;   DEF-COMPILE-FUNCTIONs for the Tiny Lisp Compiler.  (See COMPILER.LSP).
;
;===============================================================================


(eval-when (compile load)
    (include interpreter:compiler-decls) )
(eval-when (compile)
    (build '(interpreter:compiler) ) )

;***
;*** Arithmetic operators
;***

(defvar *binary-arithmetic-operator:opcode*
   '( (+           IADD)
      (-           ISUB)
      (*           IMUL)
      (//          IDIV)
      (+$          FADD)
      (-$          FSUB)
      (*$          FMUL)
      (//$         FDIV)
      (min         IMIN)
      (min$        FMIN)
      (max         IMAX)
      (max$        FMAX)
      (bit-reverse BITREV)
      ) )

(defvar *unary-arithmetic-operator:opcode*
    '( (-     INEG)
       (-$    FNEG)
       (exp   IEXP)
       (exp$  FEXP)
       (abs   IABS)
       (abs$  FABS)
       (cos   COS)
       (sin   SIN)
       (tan   TAN)
       (sqrt  SQRT)
       (float FLOAT)
       (fix   FIX)
       ) )


(def-compile-function (+ - // * +$ -$ *$ //$
                       min min$ max max$ bit-reverse
                       exp exp$ abs abs$ cos sin tan
                       sqrt float fix )
                      ( expr dest mode t-label f-label prob )
(let ( ( (operator operand1 operand2)  expr)
       ( length (length expr) )
       ( opcode  () ) )

     (caseq mode
        (for-effect)
        (for-control
            (error (list expr "Internal Tiny Lisp error." expr) ) )
        (for-value
            (? ( (&& (:= opcode (cadr (assoc operator
                                                *unary-arithmetic-operator:opcode*)
                       (= 2 length) )
                  (tlc.emit
                     '(,opcode
                       ,dest
                       ,(tlc.comp-expr operand1 () 'for-value () () () ) ) ) )

               ( (&& (:= opcode (cadr (assoc operator
                                                *binary-arithmetic-operator:opcode*)
                       (= 3 length) )
                  (tlc.emit
                     '(,opcode
                       ,dest
                       ,(tlc.comp-expr operand1 () 'for-value () () () )
                       ,(tlc.comp-expr operand2 () 'for-value () () () ) ) ) )

               ( t
                 (tlc.syntax-assert expr () ) ) ) ) )
    dest) )



;***
;*** Arithmetic comparison operators
;***

(defvar *arithmetic-comparison:opcode*
   '( (<       ILT    IF-ILT)
      (<=      ILE    IF-ILE)
      (=       IEQ    IF-IEQ)
      (!=      INE    IF-INE)
      (>=      IGE    IF-IGE)
      (>       IGT    IF-IGT)
      (<$      FLT    IF-FLT)
      (<=$     FLE    IF-FLE)
      (=$      FEQ    IF-FEQ)
      (!=$     FNE    IF-FNE)
      (>=$     FGE    IF-FGE)
      (>$      FGT    IF-FGT)
      (=0-mod  IEOMOD IF-IEOMOD)
      ) )

(def-compile-function (< <= = != >= > <$ <=$ =$ !=$ >=$ >$ =0-mod)
                      ( expr dest mode t-label f-label prob )

(tlc.syntax-assert expr (= 3 (length expr) ) )
(let* ( ( (operator operand1 operand2)
          expr)
        ( ( () value-opcode control-opcode)
          (assoc operator *arithmetic-comparison:opcode*) ) )

    (caseq mode
        (for-effect)
        (for-value
            (tlc.emit
                '(,value-opcode
                  ,dest
                  ,(tlc.comp-expr operand1 () 'for-value () () () )
                  ,(tlc.comp-expr operand2 () 'for-value () () () ) ) )
            dest)
        (for-control
            (tlc.emit
                '(,control-opcode
```

```
                ,(tlc.comp-expr operand1 () 'for-value () () () )
                ,(tlc.comp-expr operand2 () 'for-value () () () )
                ,prob
                ,t-label
                ,f-label) )
            () ) ) ) )



;***
;*** Assertion operators
;***

(def-compile-function ( assert )
                      ( expr dest mode t-label f-label prob )

(tlc.syntax-assert expr
    (&& (consp (cadr expr) )
        (= 3 (length (cadr expr) ) )
        (== 'for-effect mode) ) )

(let* ( ( (assert (operator operand1 operand2) )
          expr)
        ( ( () value-opcode control-opcode)
          (assoc operator *arithmetic-comparison:opcode*) ) )
    (tlc.syntax-assert expr value-opcode)

    (tlc.emit
        '(assert
          ,value-opcode
          ,(tlc.comp-expr operand1 () 'for-value () () () )
          ,(tlc.comp-expr operand2 () 'for-value () () () ) ) )
    () ) )


;***
;*** (! expr)
;***
;***    Notice the tiny amount of optimization for such as (! (> x y) ).
;***

(defvar *bool-operator:negation*
        '( (<   >=)
           (<=  >)
           (=   !=)
           (!=  =)
           (>=  <)
           (>   <=)
           (<$  >=$)
           (<=$ >$)
           (=$  !=$)
           (!=$ =$)
           (>=$ <$)
           (>$  <=$)
           ) )

(def-compile-function ! ( expr dest mode t-label f-label prob )

(tlc.syntax-assert expr (= 2 (length expr) ) )
(let ( ( ( () operand)  expr)
       (new-operator)
       (actual-dest) )
```



```
    (caseq mode
        (for-control
            (tlc.comp-expr operand () mode f-label t-label (- 1.0 prob) ) )
        (for-effect
            (tlc.comp-expr operand () mode () () () ) )
        (for-value
            (? ( (&& (consp operand)
                     (== '! (car operand) ) )
                 (tlc.comp-expr (cadr operand)
                                dest mode t-label f-label prob) )

               ( (&& (consp operand)
                     (:= new-operator
                         (cadr (assoc (car operand) *bool-operator:negation*))))
                 (tlc.comp-expr `(,new-operator ,,(cdr operand) )
                                dest mode t-label f-label prob) )

               ( t
                 (:= actual-dest (tlc.comp-expr operand
                                                () 'for-value () () () ) )
                 (tlc.emit `(inot ,dest ,actual-dest) )
                 dest) ) ) ) ) )




;***
;*** (BLOCK e1 e2 ... en)
;***

(def-compile-function block ( expr dest mode t-label f-label prob )
    (tlc.syntax-assert expr (<= 2 (length expr) ) )
    (loop
        (initial rest (cdr expr) )
        (while rest)
        (do
            (if (! (cdr rest) )         ;*** last expression in block?
                (tlc.comp-expr (car rest) dest mode t-label f-label prob)
                (tlc.comp-expr (car rest) ()   'for-effect () () () ) ) )
        (next rest (cdr rest) )
        (result dest) ) )


;***
;*** (DEF-BLOCK ( param1 param2 ... ) (live-var1 live-var2 ...) e1 e2 ... en)
;***
;***  Param1 and live-var1 can be either a symbol (a variable name) or
;***  a list of the the form:
;***
;***        (array-name dim1 dim2 ...)
;***
;***  where the meaning of the dimi are as in (DECLARE...
;***

(def-compile-function def-block ( expr dest mode t-label f-label prob )

(tlc.syntax-assert expr (&& (<= 4  (length expr) )
                            (listp (cadr  expr) )
                            (listp (caddr expr) ) ) )
(let ( ( (() params live-vars . block-exprs)
```

```
        expr)
      ( array-decls )
      ( array-decl  )
      ( actual-dest ) )

   (for (var in '(,,params ,,live-vars) ) (do
       (? ( (consp var)
            (:= array-decl '(declare ,,var) )
            (if (! (member array-decl array-decls) )
                (push array-decls array-decl) ) )
          ( (litatom var) )
          ( t
            (tlc.syntax-assert expr () ) ) ) ) )

   (tlc.emit '(def-block ,params ,live-vars) )
   (for (var in params) (do
       (tlc.emit '(param ,(if (litatom var)
                              var
                              (car var) ) ) ) ) )
   (tlc.emit '(def ,,(for (var in params)
                          (when (litatom var) )
                          (save '(,var ,var) ) ) ) )

   (:= actual-dest
       (tlc.comp-expr '(block ,,array-decls ,,block-exprs )
                      dest mode t-label f-label prob) )

   (tlc.emit '(use  ,,(for (var in live-vars)
                           (when (litatom var) )
                           (save '(,var ,var) ) ) ) )
   (tlc.emit '(live ,,(for (var in live-vars)
                           (save (if (litatom var)
                                     var
                                     (car var) ) ) ) ) )
   actual-dest) )



;***
;*** (:= e1 e2)
;***

(def-compile-function := ( expr dest mode t-label f-label prob )

(tlc.syntax-assert expr (= 3 (length expr) ) )
(let* ( ( ( () expr1 expr2)  expr)
        ( var-ref  (tlc.variable-l-value expr1) )
        ( location () ) )

    (caseq mode
        (for-value
            (if dest (then
                (:= location (tlc.comp-expr expr2 dest 'for-value () () ()))
                (tlc.comp-expr location var-ref 'for-value () () () )
                dest)
            (else
                (tlc.comp-expr expr2 var-ref 'for-value () () () ) ) ) )
        (for-control
            (:= location (tlc.comp-expr expr2 var-ref 'for-value () () () ) )
            (tlc.emit '(truego ,location ,prob ,t-label ,f-label) ) )
        (for-effect
            (tlc.comp-expr expr2 var-ref 'for-value () () () ) ) ) ) )
```

5

```
;***
;*** (IF bool (THEN c1 c2 ... cn) (ELSE a1 a2 ... an) (EXPECT probability) )
;***
;***    ELSE and EXPECT clauses are optional.
;***

(def-compile-function if ( expr dest mode t-label f-label prob )

(let ( ( ( () bool-expr (then . expris) . rest-if)
         expr)
       ( expr2s )
       ( expect-prob )
       ( label1 (tlc.generate-label) )
       ( label2 (tlc.generate-label) )
       ( label3 (tlc.generate-label) ) )

    (tlc.syntax-assert expr (== 'then then) )
    (if (&& (consp (car rest-if) )
            (== 'else (caar rest-if) ) )
    (then
        (desetq ( () . expr2s) (car rest-if) )
        (pop rest-if) ) )
    (if (&& (consp (car rest-if) )
            (== 'expect (caar rest-if) ) )
    (then
        (desetq ( () expect-prob) (car rest-if) )
        (pop rest-if)
        (tlc.syntax-assert expr (== 'flonum (numtype expect-prob) ) ) ) )
    (tlc.syntax-assert expr (! rest-if) )

    (if (! expr2s)
        (:= expr2s '(0) ) )

    (tlc.comp-expr bool-expr () 'for-control label1 label2 expect-prob)

    (tlc.emit '(label ,label1) )
    (tlc.comp-expr '(block ,,expris)
                   dest mode t-label f-label prob)
    (if (!== 'for-control mode)
        (tlc.emit '(goto ,label3) ) )

    (tlc.emit '(label ,label2) )
    (tlc.comp-expr '(block ,,expr2s)
                   dest mode t-label f-label prob)

    (if (!== 'for-control mode)
        (tlc.emit '(label ,label3) ) )

    dest) )


;***
;*** (IF-GO bool expect-prob true-label [false-label])
;***
;***    Crocko statement used internally by such as LOOP.
;***    Let's us use IF- naddr operators directly without needing
;***    a peephole optimizer for eliminating jumps to jumps).
;***

(def-compile-function if-go ( expr dest mode t-label f-label prob )
```

6

```
:t ( ( ( () bool-expr expect-prob t-label f-label)  expr)
     ( follow-label () ) )

    (tlc.syntax-assert expr
        (&& (litatom t-label)
            (== 'flonum (numtype expect-prob) )
            (|| (! f-label)
                (litatom f-label) )
            (== 'for-effect mode) ) )

    (if (! f-label)
        (:= follow-label (tlc.generate-label) )
        (:= follow-label f-label) )

    (tlc.comp-expr bool-expr () 'for-control t-label follow-label expect-prob)

    (if (!== follow-label f-label)
        (tlc.emit '(label ,follow-label) ) )
    ) )



;***
;*** (LABEL label)
;*** (GOTO  label)
;***

(def-compile-function (label goto) ( expr dest mode t-label f-label prob )

    (tlc.syntax-assert expr (&& (= 2 (length expr) )
                                (litatom (cadr expr) ) ) )
    (tlc.emit expr) )



;***
;*** (LOOP [loop-name]
;***     (ITERATIONS-GUESS integer)
;***     (UNROLL [constant] [NO-BACK-JUMP] [NO-EXIT-TESTS] [NO-FOLD])
;***     (INCR var FROM expr [TO expr] )
;***     (DECR var FROM expr [TO expr] )
;***     (STEP var FROM expr USING expr [WHILE|UNTIL expr] )
;***     (WHILE expr)
;***     (UNTIL expr)
;***     (DO e1
;***         ...
;***         (CONTINUE loop-name)  or (LEAVE loop-name expr)
;***     (RESULT e1 e2 ... en) )
;***
;*** The special expression (UNROLL-INDEX loop-name) expands to the
;*** unrolling of the named loop (starting with 1).
;***

(def-struct loop-desc   ;*** Describes a current nested loop
    name                    ;*** Name of the loop (given on LEAVE and CONTINUE)
    top-label               ;*** Top of the loop label
    continue-label          ;*** Label to CONTINUE to
    leave-label             ;*** Label to LEAVE to
    dest                    ;*** Tlc.COMP-EXPR paramaters to compile
    mode                    ;***    LEAVE results with
    t-label                 ;***
    f-label                 ;***
    prob                    ;***
```



```
    back-jump?              ;*** True if this loop wants a back jump
    )



(def-compile-function loop ( expr dest mode t-label f-label prob )

(let ( (name           () ) ;*** The name of the loop
       (label          () ) ;*** <name>-genynsed-number
       (unroll         1  ) ;*** # of times to unroll the loop
       (iterations-guess
                       *iterations-guess-default*)
                           ;*** # of times programmer guesses loop will exec.
       (leave-prob     (// 1.0 (flonum *iterations-guess-default*) ) )
                           ;*** Prob. of exiting loop
       (back-jump?      t ) ;*** Generate a backjump to the top of the loop
       (exit-tests?     t ) ;*** Generate if-tests for exiting the loop
       (fold-step-vars? t ) ;*** Constant fold successive index calculations
       (incr-decr-vars () ) ;*** Vars in INCR and DECR clauses.
       (step-vars      () ) ;*** Vars in STEP clauses.
       (init-code      () ) ;*** Code for initializing loop variables
       (step-code      () ) ;*** Code for stepping loop variables
       (body-code      () ) ;*** Code for DO body and exit tests
       (result-code    () ) ;*** Code for the result
       (top-label      () ) ;*** Goto label for top of loop
       (result-label   () ) ;*** Goto label for result code (loop end)
       (leave-label    () ) ;*** Goto label for LEAVEing loop
       (continue-label () ) ;*** Goto label for CONTINUEing loop
       (final-dest     () ) ;*** Where the result of LOOP was compiled.
       )

        ;*** get the loop label, if any
        ;
    (if (litatom (cadr expr) ) (then
        (:= name  (cadr expr) )
        (:= label (atomconcat name (tlc.gensym '-) ) )
        (:= expr  (cddr expr) ) )
    (else
        (:= name (:= label (tlc.gensym 'loop) ) )
        (:= expr  (cdr expr) ) ) )

        ;*** build the internal goto labels
        ;
    (:= top-label      (atomconcat label '-top) )
    (:= result-label   (atomconcat label '-result) )
    (:= leave-label    (atomconcat label '-leave) )
    (:= continue-label (atomconcat label '-continue) )


        ;*** enumerate through the clauses, checking syntax, and
        ;*** building up the different code sections
        ;
    (for (clause in expr) (do
        (? ( (litatom clause)
             (:= label clause) )

           ( (consp clause)
             (caseq (car clause)

                (unroll
                    (tlc.syntax-assert clause (cdr clause) )
                    (for (keyword in (cdr clause) ) (do
                        (if (inump keyword) (then
```

```
                (:= unroll keyword) )
            (else
                (caseq keyword
                    (no-back-jump
                        (:= back-jump? () ) )
                    (no-exit-tests
                        (:= exit-tests? () ) )
                    (no-fold
                        (:= fold-step-vars? () ) )
                    (t
                        (tlc.syntax-assert clause nil) ) ) ) ))))

        (iterations-guess
            (tlc.syntax-assert clause (numberp (cadr clause) ) )
            (:= iterations-guess (flonum (cadr clause) ) )
            (:= leave-prob (// 1.0 iterations-guess) ) )

        (result
            (:= result-code (cdr clause) ) )

        (do
            (for (expr in (cdr clause) ) (do
                (push body-code expr) ) ) )

        (while
            (push body-code
                  '(if-go (! ,(cadr clause) )
                          ,leave-prob ,result-label) ) )

        (until
            (push body-code
                  '(if-go ,(cadr clause) ,leave-prob ,result-label)))

        (step
            (let ( ( (step var from init-expr using using-expr
                           while while-expr)
                     clause) )
                (tlc.syntax-assert clause
                    (&& (== 'from from)
                        (== 'using using)
                        (|| (! while)
                            (== 'until while)
                            (== 'while while) ) ) )
                (push init-code '(:= ,var ,init-expr) )
                (push step-vars var)

                (caseq while
                    (while
                        (push body-code
                              '(if-go (! ,while-expr)
                                      ,leave-prob ,result-label) ) )
                    (until
                        (push body-code
                              '(if-go ,while-expr
                                      ,leave-prob ,result-label) ) )
                    ( t ) )

                (push step-code '(:= ,var ,using-expr) ) ) )

        ( (incr decr)
            (let ( ( (incr var from init-expr . rest)
                     clause)
                   (to)  (to-expr) (to-temp)
```



```
                   (by)  (by-expr) (by-temp) )
                (tlc.syntax-assert clause (== 'from from) )
                (if rest (then
                    (desetq (to to-expr . rest) rest)
                    (tlc.syntax-assert clause (== 'to to) ) ) )
                (if rest (then
                    (desetq (by by-expr . rest) rest)
                    (tlc.syntax-assert clause (== 'by by) ) )
                (else
                    (:= by-expr 1) ) )
                (tlc.syntax-assert clause (! rest) )

                (push incr-decr-vars var)
                (push init-code '(:= ,var ,init-expr) )

                (if to (then
                    (if (numberp to-expr) (then
                        (:= to-temp to-expr) )
                    (else
                        (:= to-temp (tlc.gensym '%) )
                        (push init-code '(:= ,to-temp ,to-expr) ) ) )

                    (if (== 'incr incr) (then
                        (push body-code
                              '(possible-if-go (> ,var ,to-temp)
                                               ,leave-prob ,result-label) ) )
                    (else
                        (push body-code
                              '(possible-if-go (< ,var ,to-temp)
                                               ,leave-prob ,result-label) )))))

                (if (numberp by-expr) (then
                    (:= by-temp by-expr) )
                (else
                    (:= by-temp (tlc.gensym '%) )
                    (push init-code '(:= ,by-temp ,by-expr) ) ) )

                (if (== 'incr incr) (then
                    (push step-code '(:= ,var (+ ,var ,by-temp) ) ) )
                (else
                    (push step-code '(:= ,var (- ,var ,by-temp) ))))))

            ( t
                (tlc.syntax-assert clause () ) ) ) )

        ( t
            (tlc.syntax-assert clause () ) ) ) ) )


(:= init-code (dreverse init-code) )
(:= body-code (dreverse body-code) )
(:= step-code (dreverse step-code) )

    ;***
    ;*** Compile the initial code
    ;***
    ;
(for (expr in init-code) (do
    (tlc.comp-expr expr () 'for-effect () () () ) ) )

    ;***
    ;*** Unroll and compile the loop body, including the stepping
    ;***
```

```
    ;*** beginning of loop proper.
    ;
(tlc.emit '(label ,top-label) )
(if back-jump? (then
    (tlc.emit '(loop-start ,top-label ,iterations-guess) ) ) )

(loop (initial continue-label-i () )
      (incr i from 1 to unroll)
      (next continue-label-i (atomconcat continue-label i) )
(do
        ;*** Push a loop descriptor for any LEAVEs or CONTINUEs
        ;
    (push *tlc.loop-stack*
          (loop-desc:new name           name
                         top-label      top-label
                         leave-label    leave-label
                         continue-label continue-label-i
                         dest           dest
                         mode           mode
                         t-label        t-label
                         f-label        f-label
                         prob           prob
                         back-jump?     back-jump?) )

        ;*** Compile the body code.  Get rid of the step tests if
        ;*** we don't want them.
        ;
    (for (expr in (sublis '( ( (unroll-index ,name) ,,i)
                             (,result-label ,,(atomconcat result-label i)))
                          body-code) )
    (do
        (? ( (!== (car expr) 'possible-if-go)
             (tlc.comp-expr expr () 'for-effect () () () ) )

           ( exit-tests?            ;*** we've got an if-possible-go,
                                    ;*** do it if we want it.
             (tlc.comp-expr '(if-go ,,(cdr expr) )
                            () 'for-effect () () () ) ) ) )

        ;*** Pop the loop descriptor (insuring that LEAVEs and
        ;*** and CONTINUEs can only be done in the loop body).
        ;
    (pop *tlc.loop-stack*)

    (tlc.emit '(label ,continue-label-i) )

        ;*** Compile the step code, folding INCR/DECR vars.
        ;
    (loop (for ( () var step-expr) in step-code) (do
        (if (&& fold-step-vars?
                (> unroll i)
                (memq var incr-decr-vars) )
        (then
            (let ( ((+- () delta) step-expr)
                   (var-i       (atomconcat '% var i) )
                   (var-i+1     (atomconcat '% var (+ i 1) ) ) )
                (if (== i 1) (then
                    (tlc.comp-expr
                        '(:= ,var-i ,var)
                        () 'for-effect () () () ) ) )
                (tlc.comp-expr
                    '(:= ,var-i+1 (,+- ,var-i (* ,i ,delta) ) )
                    () 'for-effect () () () )
                (tlc.comp-expr
                    '(:= ,var ,var-i+1)
                    () 'for-effect () () () ) ) )
            (else
                (tlc.comp-expr
                    '(:= ,var ,step-expr)
                    () 'for-effect () () () ) ) ) ) )

    ) )

    ;***
    ;*** Compile the backjump to the top of the loop, if wanted
    ;***
    ;
(if back-jump? (then
    (tlc.emit (list 'trace-fence) )
    (tlc.emit '(goto ,top-label) ) )
(else
    (tlc.emit '(goto ,result-label) ) ) )

    ;***
    ;*** Compile the exit code for each of the loop unrollings.  The
    ;*** exit code for unrolling #3 looks like:
    ;***
    ;***     (LABEL RESULT-LABEL3)
    ;***     (ASSIGN VAR1 VAR1)
    ;***     ...
    ;***     (ASSIGN VARn VARn)
    ;***     (GOTO RESULT-LABEL)
    ;***
    ;*** The VARi are the vars from the STEP, INCR, and DECR clauses.
    ;*** By having each unrolling exit to its own copy of such code,
    ;*** the seeminly useless assignemnts allow variable renaming
    ;*** to rename the STEP/INCR/DECR vars in each unrolling.
    ;
(loop (incr i from 1 to unroll) (do
    (tlc.emit '(label ,(atomconcat result-label i) ) )
    (loop (for var in (unionq incr-decr-vars step-vars) ) (do
        (tlc.emit '(assign ,var ,var) ) ) )
    (tlc.emit '(goto ,result-label) ) ) )

    ;***
    ;*** Compile the result code.
    ;***
    ;
(tlc.emit '(label ,result-label) )
(if back-jump? (then
    (tlc.emit '(loop-end ,top-label) ) ) )
(if (! result-code)
    (:= result-code '(0) ) )

(loop (initial rest-code result-code)
      (while rest-code)
(do
    (if (! (cdr rest-code) ) (then
        (:= final-dest
            (tlc.comp-expr (car rest-code)
                           dest mode t-label f-label prob) ) )
    (else
        (:= final-dest
            (tlc.comp-expr (car rest-code) () 'for-effect () () ()))))))
(next rest-code (cdr rest-code) ) )
```

```
    (tlc.emit '(label ,leave-label) )

    final-dest) )



;***
;*** (LEAVE loop-name result-expr)
;***

(def-compile-function leave ( expr dest mode t-label f-label prob )

(let ( ( (() loop-name result-expr)
         expr)
       ( loop-desc ) )
    (tlc.syntax-assert expr (&& (= 3 (length expr) )
                                (litatom loop-name)
                                (== mode 'for-effect) ) )
    (:= loop-desc (tlc.find-loop-desc loop-name expr) )
    (if (loop-desc:back-jump? loop-desc) (then
        (tlc.emit '(loop-end ,(loop-desc:top-label loop-desc) ) ) ) )
    (tlc.comp-expr result-expr
                   (loop-desc:dest    loop-desc)
                   (loop-desc:mode    loop-desc)
                   (loop-desc:t-label loop-desc)
                   (loop-desc:f-label loop-desc)
                   (loop-desc:prob    loop-desc) )
    (tlc.emit '(goto ,(loop-desc:leave-label loop-desc) ) )
    () ) )



;***
;*** (CONTINUE loop-name)
;***

(def-compile-function continue ( expr dest mode t-label f-label prob )

(let ( ( (() loop-name)
         expr)
       ( loop-desc ) )
    (tlc.syntax-assert expr (&& (= 2 (length expr) )
                                (litatom loop-name)
                                (== mode 'for-effect) ) )
    (:= loop-desc (tlc.find-loop-desc loop-name expr) )
    (tlc.emit '(goto ,(loop-desc:continue-label loop-desc) ) )
    () ) )



;***
;*** Find the loop descriptor on the loop stack that matches LOOP-LABEL
;***

(defun tlc.find-loop-desc ( loop-name leave-continue-expr )
    (loop
        (for desc in *tlc.loop-stack*)
        (do (if (== loop-name (loop-desc:name desc) )
                (return desc) ) )
        (result
            (error (list leave-continue-expr
```



```
                "There is no loop with this LEAVE//CONTINUE label") ) ) ) )



;***
;*** (ESC lisp-expression)
;***
;***    Replace occurrences of (VAR expression) within lisp-expression
;***    (VAR temp-var), generating naddr-code to evaluate expression.
;***

(def-compile-function esc ( expr dest mode t-label f-label prob )
    (tlc.syntax-assert expr (= 2 (length expr) ) )
    (tlc.emit '(esc ,(tlc.comp-esc (cadr expr) ) ) ) )

(defun tlc.comp-esc ( expr )
    (? ( (&& (consp expr)
             (== 'var (car expr) ) )
         (let ( (location (tlc.comp-expr (cadr expr)
                                         () 'for-value () () () ) ) )
           '(var ,location) ) )
       ( (consp expr)
         (cons (tlc.comp-esc (car expr) )
               (tlc.comp-esc (cdr expr) ) ) )
       ( t
         expr) ))


;***
;*** (LIVE (var1 var2 ... varn) expr1 expr2 ... exprn)
;***

(def-compile-function live ( expr dest mode t-label f-label prob )
    (tlc.syntax-assert expr (<= 3 (length expr) ) )
    (let ( ( actual-dest
             (tlc.comp-expr '(block ,,(cddr expr) )
                            dest mode t-label f-label prob) ) )
         (tlc.emit '(live ,,(cadr expr) ) )
         actual-dest) )



;***
;*** (EXPECT number)
;***

(def-compile-function expect ( expr dest mode t-label f-label prob )
    (tlc.syntax-assert expr (&& (= 2 (length expr) )
                                (== 'for-effect mode) ) )
    (tlc.emit expr) )


;***
;*** (DECLARE array-name dim1 dim2 ... dimn [(INITIAL val1 val2 ...)] )
;***
;*** Each of the dimI can be either a pair (lower upper) or else a number,
;*** which is the same as (0 number-1).  The optional INITIAL clause
;*** specifies initial values vali for the array (stored in row major
;*** order, 0's used for padding).
;***

(def-compile-function declare ( expr dest mode t-label f-label prob )
```

```
(tlc.syntax-assert expr (&& (> (length expr) 2)
                            (litatom (cadr expr) )
                            (== 'for-effect mode) ) )
(let ( (dimensions () )
       (initial    () )
       (var        (cadr expr) )
       (size       0) )

    (for (dim in (cddr expr) ) (do
        (if (&& (consp dim)
                (== 'initial (car dim) ) )
        (then
            (:= initial (cdr dim) ) )
        (else
            (push dimensions dim) ) ) ) )
    (:= dimensions (dreverse dimensions) )

    (:= size (tlc.array:declare var dimensions) )
    (tlc.emit '(dcl ,var ,size ,initial) ) ) )
```

```
;===============================================================================
;
; Convert a tiny subset of a numeric Lisp-like language into NADDR.
;
;===============================================================================

;***
;*** Global variable initialization.
;***

(eval-when (compile load)
    (include interpreter:compiler-decls) )


(defvar *expect-default* 0.5)    ;*** Default expect to use for conditional
                                 ;***   jumps if none is specified.
(defvar *tlc.naddr-code* () )    ;*** Reverse list of naddr code being
                                 ;***   generated.
(defvar *tlc.loop-stack*)        ;*** Stack of descriptors of nested loops,
                                 ;***   for compiling LEAVE and CONTINUE
(defvar *tlc.gensym-counter*)    ;*** Our own gensym counter, for repeatable
                                 ;***   results.
(defvar *iterations-guess-default* 100.0)
                                 ;*** Default value for expected number
                                 ;***   of iterations of a loop
(:= *tlc.reserved-words* () )


;***
;*** Compile a Tiny Lisp expression for one of modes FOR-VALUE, FOR-CONTROL,
;***    or FOR-EFFECT, and return the NADDR code.
;***

(defun compile-tiny-lisp ( expr &optional (mode 'for-effect) )
    (:= *tlc.naddr-code*     () )
    (:= *tlc.loop-stack*     () )
    (:= *tlc.gensym-counter* 0)
    (tlc.arrays:initialize)

    (let ( (dest (tlc.comp-expr expr () mode 'label1 'label2 () ) ) )
        (tlc.emit (list 'stop) )
        (:= *tlc.naddr-code* (dreverse *tlc.naddr-code*) ) ) )


;***
;*** Same as COMPILE-TINY-LISP, except just print out the results pretty.
;***

(defun ctl ( expr &optional (mode 'for-value) )
    (:= *tlc.naddr-code*     () )
    (:= *tlc.loop-stack*     () )
    (:= *tlc.gensym-counter* 0)
    (tlc.arrays:initialize)

    (let ( (dest (tlc.comp-expr expr () mode 'label1 'label2 () ) ) )
        (tlc.emit (list 'stop) )
        (:= *tlc.naddr-code* (dreverse *tlc.naddr-code*) )
        (msg 0 "Result = " dest t (h *tlc.naddr-code*) ) ) )


;***
;*** Our own internal gensym that will generate the same "unique" symbols
;*** each time we run the compiler.  Always interns the symbol;  prefix
```



```
;*** can be more than a single character.
;***

(defun tlc.gensym ( prefix )
    (atomconcat prefix (++ *tlc.gensym-counter*) ) )


;***
;*** T if symbol is any reserved word that can occur in the car of a form.
;***

(defun tlc.reserved-word? ( symbol )
    (memq symbol *tlc.reserved-words*) )


;***
;*** Mapping from reserved-word onto a function that compiles forms whose
;*** car is the reserved-word.  :='able.
;***

(defmacro reserved-word:compile-function ( symbol )
    `(get ,symbol 'reserved-word:compile-function) )



;***
;*** (DEF-COMPILE-FUNCTION <symbol or list of symbols> <args> . <body>)
;***

(defmacro def-compile-function ( symbols args . body )
    (if (! (consp symbols) )
        (:= symbols (list symbols) ) )
    (let ( (compile-function (atomconcat (car symbols) '-compile-function) ) )
        `(eval-when (eval compile load)
            (defun ,compile-function ,args ,,body)
            ,,(for (symbol in symbols)
                   (splice `( (push *tlc.reserved-words* ',symbol)
                              (:= (reserved-word:compile-function ',symbol)
                                  ',compile-function) ) ) ) ) ) )

;***
;*** Raise an error if VALUE is false.
;***

(defun tlc.syntax-assert ( expr value )
    (if (! value)
        (error (list expr "Invalid Tiny Lisp expression.") ) ) )


;***
;*** Emit a single NADDR instruction.  Always returns NIL.
;***
;
(defun tlc.emit ( naddr-instr )
    (push *tlc.naddr-code* naddr-instr)
    () )

;***
;*** Make a new label (but don't emit it.
;***

(defun tlc.generate-label ()
    (tlc.gensym 'l) )
```

```
;***
;*** Array declarations and "symbol" table.
;***    Sample array declaration:  (A (1 5) (2 14) (3 30) )
;***
;***    Let n be the number of dimensions.
;***    Let Si be the size of dimension i (counting from 1, left to right)
;***    Let Li be the lower bound of each dimension.
;***
;***    The index calculation for (A I1 I2 ... In) is:
;***
;***    In + In-1*Sn + In-2*Sn*Sn-1 + ... + I1*Sn*Sn-1*...*S2
;***       - O
;***
;***    where
;***
;***       O = (Ln + Ln-1*Sn + ... + L1*Sn*Sn-1*...*S2)
;***
;***    The ARRAY:INDEXING-LIST for the array name contains:
;***
;***    (Sn*Sn-1*...*S2 ... Sn*Sn-1 Sn 1)
;***
;***    ARRAY:INDEXING-OFFSET contains O.
;***

(defmacro tlc.array:indexing-list ( array-name )
    `(prop 'tlc.array:indexing-list ,array-name) )

(defmacro tlc.array:indexing-offset ( array-name )
    `(prop 'tlc.array:indexing-offset ,array-name) )


    ;*** declare array-name (e.g. A) to be an array with dimensions given
    ;*** by bounds-list (e.g. ( (1 4) (0 6) ) ), storing the indexing list
    ;*** and offset with the array-name in our "symbol table".  Return
    ;*** the total size of the array.
    ;
(defun tlc.array:declare ( array-name bounds-list )
    (assert (litatom array-name) )
    (push *tlc.arrays* array-name)

    (loop
        (initial indexing-list (cons 1 () )
                 si            0
                 li            0
                 offset        0
                 si-product    1)
        (for dimension in (reverse bounds-list) )
        (do
            (if (consp dimension) (then
                (tlc.syntax-assert bounds-list
                    (&& (= 2 (length dimension) )
                        (numberp (car  dimension) )
                        (numberp (cadr dimension) ) ) )
                (:= si (+ 1 (- (cadr dimension) (car dimension) ) ) )
                (:= li (car dimension) ) )
            (else
                (tlc.syntax-assert bounds-list (numberp dimension) )
                (:= si dimension)
                (:= li 0) ) )
```



```
            (:= offset (+ offset (* li si-product) ) )
            (:= si-product (* si-product si) )
            (push indexing-list si-product) )
        (result
            (:= (tlc.array:indexing-list   array-name) (cdr indexing-list) )
            (:= (tlc.array:indexing-offset array-name) (- 0 offset) )
            si-product) ) )


(defun tlc.array:declared? ( array-name )
    (assert (litatom array-name) )
    (prop 'tlc.array:indexing-list array-name) )

(defun tlc.arrays:initialize ()
    (if (! (boundp '*tlc.arrays*) )
        (:= *tlc.arrays* () ) )
    (for (array-name in *tlc.arrays*) (do
         (remprop array-name 'tlc.array:indexing-list) ) )
    (:= *tlc.arrays* () ) )


;***
;*** T if EXPR is a variable or (array index) reference or constant number.
;***

(defun tlc.variable? ( expr )
    (|| (numberp expr)
        (litatom expr)
        (&& (consp expr)
            (litatom (car expr) )
            (tlc.array:declared? (car expr) ) ) ) )

;***
;*** Recursively Compile a Tiny Lisp expression.
;***
;*** EXPR    - the expression to be compiled
;*** DEST    - Variable where to put the result (FOR-VALUE only).
;***           This may be a vector location (V T), a scalar, or
;***           () in which case a temporary is gensymed.
;*** MODE    - one of FOR-VALUE -  evaluate the expression for its value
;***                  FOR-EFFECT - evaluate for side effects only
;***                  FOR-CONTROL- evaluate for control flow only
;*** T-LABEL - for FOR-CONTROL exprs only, where to jump if true.
;*** F-LABEL - for FOR-CONTROL exprs only, where to jump if false.
;*** PROB    - for FOR-CONTROL exprs only, the probabability of T-LABEL
;***           being taken (if (), use *expect-default*).
;***
;*** For FOR-VALUE, returns a scalar location containing the expression.
;*** Note that this is the only function for which DEST may be a vector
;*** reference; all other compile functions are guarranteed of receiving
;*** a scalar destination.
;***

(defun tlc.comp-expr ( expr dest mode t-label f-label prob )
(let* ( (function          () )
        (scalar-dest       () )
        (compiled-location () ) )

    (if (! prob)
        (:= prob *expect-default*) )

    (?  ;*** a variable or array reference?
       ( (tlc.variable? expr)
```

```
        ;*** If this is for-value and the dest is a vector reference,
        ;*** compile the variable into the () destination.
      (if (== mode 'for-value)
          (:= scalar-dest
              (if (consp dest) () dest) ) )

        ;*** compile the variable
      (:= compiled-location
          (tlc.comp-variable expr scalar-dest
                             mode t-label f-label prob) ) )
     ;*** a form?
    ( (&& (consp expr)
          (litatom (car expr) )
          (:= function (reserved-word:compile-function (car expr))))

        ;*** If this is for-value, get a scalar destination that we
        ;*** compile into.  Gensym up a temporary if we weren't given
        ;*** one or if we have an vector reference as a destination.
      (if (== mode 'for-value)
          (:= scalar-dest
              (? ( (consp dest)
                   (tlc.gensym '%) )
                 ( (== ':= (car expr) )
                   dest)
                 ( (! dest)
                   (tlc.gensym '%) )
                 ( t
                   dest) ) ) )

        ;*** compile the expression
      (:= compiled-location
          (funcall function expr scalar-dest mode t-label f-label prob)))

     ;*** a syntax error
    ( t
      (tlc.syntax-assert expr () ) ) )

     ;*** if for-value and the actual destination was an array operand,
     ;*** generate a VSTORE
  (if (&& (== mode 'for-value)
          (consp dest) )
      (tlc.emit '(vstore ,(car dest) ,(cadr dest) ,compiled-location) ) )

  compiled-location) )



;***
;*** Compile a variable or array reference, returning the location of
;*** it's value.  DEST is either a scalar or ().
;***

(defun tlc.comp-variable ( expr dest mode t-label f-label prob )
    (caseq mode
        (for-effect)
        (for-control
            (let ( (loc (tlc.comp-variable expr () 'for-value () () () ) ) )
                (if (numberp loc) (then
                    (tlc.emit '(goto ,(if (= loc 0) f-label t-label) ) ) )
                (else
                    (tlc.emit '(truego ,loc ,prob ,t-label ,f-label) ) ) ) ) )
        (for-value
```



```
            (let ( (var-loc (tlc.variable-l-value expr) ) )
                (? ( (consp var-loc)
                     (if (! dest)
                         (:= dest (tlc.gensym '%) ) )
                     (tlc.emit '(vload ,dest ,(car var-loc) ,(cadr var-loc)))
                     dest)
                   ( dest
                     (tlc.emit '(assign ,dest ,var-loc) )
                     dest)
                   ( t
                     var-loc) ) ) ) ) )


;***
;*** Return the "l-value" of a variable or array reference.  The l-value
;*** of a simple variable is just the variable itself.  For an array element
;*** reference (A I1 I2 ... In), it is (A T), where T is the temporary
;*** containing the 0-based indexing calculation for array A.   As a side
;*** effect, the code for doing the indexing is emitted.
;***

(defun tlc.variable-l-value ( expr )
    (tlc.syntax-assert expr (tlc.variable? expr) )
    (if (consp expr)
        (tlc.comp-array-ref expr)
        expr) )


;***
;*** Compile an array-reference, e.g. (A (+I J) K) into a single "vector"
;*** reference, such as (A T5), with code to do the index calculation into
;*** T5.
;***

(defun tlc.comp-array-ref ( (array-name . indices) )
    (let ( (indexing-list (tlc.array:indexing-list   array-name) )
           (expr          (tlc.array:indexing-offset array-name) )
           (term          () ) )
        (tlc.syntax-assert indices (= (length indices)
                                      (length indexing-list) ) )
        (for (index in indices)
             (si    in indexing-list)
        (do
            (if (= si 1)
                (:= term index)
                (:= term '(* ,index ,si) ) )
            (if (&& (numberp expr)
                    (= expr 0) )
                (:= expr term)
                (:= expr '(+ ,expr ,term) ) ) ) )

        '(,array-name ,(tlc.comp-expr expr () 'for-value () () () ) ) ) )
```

```
;===============================================================================
;
; Declarations of global variables for the Tiny Lisp Compiler.  This file
; should be INCLUDEd.
;
;===============================================================================

(declare (special
    *expect-default*            ;*** Default expect to use for conditional
                                ;***   jumps if none is specified.
    *tlc.naddr-code*            ;*** Reverse list of naddr code being
                                ;***   generated.
    *tlc.loop-stack*            ;*** Stack of descriptors of nested loops,
                                ;***   for compiling LEAVE and CONTINUE
    *tlc.gensym-counter*        ;*** Our own gensym counter, for repeatable
                                ;***   results.
    *iterations-guess-default*  ;*** Default value for expected number
                                ;***   of iterations of a loop
    *tlc.arrays*                ;*** List of declared array names (symbols)
    *tlc.reserved-words*        ;*** List of "reserved words" in Tiny Lisp
    *tlc.compile-assertions*    ;*** True if assertions are to be compiled
                                ;***   in-line.
    ) )
```

```
;==============================================================================
;
; Parallel Naddr Interpreter
;
; Warning:  this version of NADDR is not yet (and may never be)
;           documentated anywhere except here in the source.  It is
;           currently a moving target.
;
; Some Terminology:
;       operator          -- a single atomic function like IADD.
;       operation, oper   -- an operator togethor with it's operands.
;                            e.g. (IADD X Y Z)
;       instruction, instr -- a list of operations, all to be executed
;                            in parallel, e.g. ( (IADD X Y Z) (IMUL
;                            A B C) )
;
;==============================================================================

(eval-when (compile load)
    (include interpreter:interpreter-decls.lsp) )

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

;***
;*** Miscellaneous options.
;***

(def-option *int.invalid-index-action* 'break interpreter: "
Specifies what action to take when a VLOAD//VSTORE index is out of bounds:
    ()     -- do nothing.
    'WARN  -- print an informative message and continue execution.
    'BREAK -- print an informative message and break.
")


(def-option *int.assertion-failed-action* 'break interpreter: "
Specifies what action to take when an ASSERT fails:
    ()     -- do nothing.
    'WARN  -- print an informative message and continue execution.
    'BREAK -- print an informative message and break.
")


;***
;*** Memory
;***

(declare (special
    *int.memory*        ;*** Array of Lisp values, either integers or floats.
    *int.memory-length* ;*** Size of memory.
    *int.memory-free*   ;*** Index of next unallocated word in memory.
    ) )

(defmacro memory ( index )
    '([] *int.memory* ,index) )

(defun memory:initialize ()
    (if (! (boundp '*int.memory-length*) ) (then
        (:= *int.memory-length* 8000) ) )

    (if (|| (! (boundp '*int.memory*) )
```



```
            (!= *int.memory-length* (vectorlength *int.memory*) ) )
    (then
        (msg 0 "INTERPRETER: Re-initializing memory to be "
             *int.memory-length*
             " words." t)
        (:= *int.memory* (makevector *int.memory-length*) ) ) )

    (vector:initialize *int.memory* 0)
    (:= *int.memory-free* 0)
    () )


;***
;*** Labels
;***

(declare (special
    *int.labels*        ;*** list of symbols that are labels
    ) )

(defun label:pc ( label )
    (get label 'int.label:pc) )

(defun label:destroy ( label )
    (remprop label 'int.label:pc) )

(defun label:declare ( label pc )
    (push *int.labels* label)
    (:= (get label 'int.label:pc) pc) )

(defun labels:initialize ()
    (if (! (boundp '*int.labels*) )
        (:= *int.labels* () ) )
    (for (label in *int.labels*) (do
        (label:destroy label) ) )
    (:= *int.labels* () ) )


;***
;*** Variables
;***

(declare (special
    *int.variables*     ;*** list of symbols that are variable names
    ) )

(defmacro variable:base ( var )
    '(get ,var 'int.variable:base) )

(defmacro variable:length ( var )
    '(get ,var 'int.variable:length) )

(defmacro variable:declared? ( var )
    '(variable:base ,var) )

(defun variable:destroy ( var )
    (remprop var 'int.variable:length)
    (remprop var 'int.variable:base) )

(defun variable:declare ( var length init-value-list )
    (if (! (variable:declared? var) ) (then
        (push *int.variables* var)
        (:= (variable:base   var) *int.memory-free*)
```

```
        (:= (variable:length var) length)
        (:= *int.memory-free* (+ *int.memory-free* length) )
        (variable:initialize var init-value-list) ) ) )

(defun variable:initialize ( var initial-list ) (prog ()
    (if (! initial-list)
        (return () ) )
    (loop (initial value             0
                   rest-initial-list initial-list)
          (incr i from (variable:base var)
                to   (+ -1 (+ (variable:base   var)
                              (variable:length var) ) ) )
    (do
        (if (! rest-initial-list)
            (:= rest-initial-list initial-list) )
        (pop rest-initial-list value)
        (:= (memory i) value) ) ) ) )

(defun variables:initialize ()
    (if (! (boundp '*int.variables*) ) (then
        (:= *int.variables* () ) ) )
    (for (var in *int.variables*) (do
        (variable:destroy var) ) )
    (:= *int.variables* () ) )



;***
;*** Operand functions
;***

(defun operand:l-value ( operand )
    (if (numberp operand)
         operand
         (variable:base operand) ) )

(defun operand:r-value ( operand )
    (if (numberp operand)
        operand
        (memory (operand:l-value operand) ) ) )

(defun operand:declare ( operand )
    (if (! (numberp operand) )
        (variable:declare operand 1 '(0) ) ) )

(defmacro var ( operand )
    '(operand:r-value ',operand) )


;***
;*** The instruction queue remembers the last n instructions executed.
;***

(declare (special
    *int.instr-q-vector* ;*** Vector implementing the queue.
    *int.instr-q-max-length*
                         ;*** Max allowable length of the queue
    *int.instr-q-length* ;*** Current length.
    *int.instr-q-head*   ;*** Index of the oldest item on the queue.
    *int.instr-q-tail*   ;*** Index of the newest item on the queue.
    ) )
(:= *int.instr-q-max-length* 200)
```



```
(defun instr-q:initialize ()
    (:= *int.instr-q-vector* (makevector *int.instr-q-max-length*) )
    (:= *int.instr-q-length* 0)
    (:= *int.instr-q-head*   0)
    (:= *int.instr-q-tail*   (+ -1 *int.instr-q-max-length*) )
    () )

(eval-when (eval compile)
    (defmacro instr-q:incr-index ( i )
        '(if (== (++ ,i) *int.instr-q-max-length*) (then
            (:= ,i 0) ) ) ) )

(defun instr-q:add ( instr )
    (instr-q:incr-index *int.instr-q-tail*)
    (if (== *int.instr-q-length* *int.instr-q-max-length*) (then
        (instr-q:incr-index *int.instr-q-head*) )
    (else
        (++ *int.instr-q-length*) ) )
    (:= ([] *int.instr-q-vector* *int.instr-q-tail*) instr) )


(defun instr-q:list ()
    (loop (incr l from 1 to *int.instr-q-length*)
          (initial i *int.instr-q-head*)
          (save ([] *int.instr-q-vector* i) )
          (do (instr-q:incr-index i) ) ) )


;***
;*** Instruction operations
;***

(defmacro instr:normalize ( instr )
    '(if (consp (car ,instr) )
         ,instr
         (cons ,instr () ) ) )

(defun oper:declare ( oper )
    (let ( (function (group:declare-function
                          (operator:group (oper:operator oper) ) ) ) )
         (if function
             (funcall function oper) ) ) )

(defun instr:declare ( instr )
    (for (oper in instr) (do
        (oper:declare oper) ) ) )


(defun instr:costsp ( instr )
    (loop (for oper in instr) (do
        (if (!= 0 (operator:cost (oper:operator oper) ) )
            (return t) ) )
    (result () ) ) )



(defun instr:execute ( instr )
    (:= *int.destinations* () )
    (:= *int.results*      () )
    (if (instr:costsp instr) (then
        (++ *int.instruction-count*) ))
    (instr-q:add instr)
    (loop
```

```
                (variable:declared? var) ) )
            (! (&& (consp var)
                (variable:declared? (car var) ) ) ) )
    (then
        (msg 0 "Variable " var " isn't used." t)
        (go continue) ) )


        ;*** a scalar, print it simply and return
    (if (litatom var) (then
        (msg 0 "Variable " var " = " (operand:r-value var) t)
        (go continue) ) )

        ;*** an array, print out each element of the array, labeled
        ;*** with its index.

        ;*** first make a list of triples of the form (LOWER UPPER SIZE)
        ;*** corresponding to the dimensions of the array.
        ;
    (:= dimensions () )
    (loop (initial size  1
                   lower 0
                   upper 0)
          continue
          (for dimension in (reverse (cdr var) ) )
    (do
        (if (&& (consp dimension)
                (== 'initial (car dimension) ) )
            (go continue) )
        (if (consp dimension) (then
            (desetq (lower upper) dimension) )
        (else
            (:= lower 0)
            (:= upper (+ -1 dimension) ) ) )
        (push dimensions '(,lower ,upper ,size) )
        (:= size (* size (+ 1 (- upper lower) ) ) ) ) )

        ;*** now print each element of the array, labeled with the index
        ;
    (msg "Variable " var " = " t)
    (loop (incr i from (variable:base (car var) )
                to   (+ -1 (+ (variable:base (car var) )
                              (variable:length (car var) ) ) ) )
    (do
        (msg "    ")
        (loop (initial remainder (- i (variable:base (car var) ) ) )
              (for (lower upper size) in dimensions)
        (do
            (msg " " (+ lower (// remainder size) ) )
            (:= remainder (\ remainder size) ) ) )

        (msg " = " (memory i) t) ) )
    ) ) )
```

```
;===============================================================================
;
; Declarations for the Interpreter -- this file is INCLUDEd.
;
;===============================================================================

;***
;*** Declarations for global variables exported by INTERPRETER
;***

(declare (special
    *int.pc*                    ;*** the next instr to execute (nil to stop).
    *int.running?*              ;*** true as long as the interpreter hasn't
                                ;*** encountered a STOP instruction.
    *int.instruction-count*     ;*** # of instrs executed during the last
                                ;*** call to interpret
    *int.operation-count*       ;*** # of operations executed

                                ;*** Options defined elsehwere with DEF-OPTION:
    *int.invalid-index-action*
    *int.assertion-failed-action*
    ) )


;***
;*** Declarations for own variables shared by parts of the interpreter
;***

(declare (special
    *int.destinations*  ;*** list of memory addresses to store results of curren
                        ;*** instr
    *int.results*       ;*** list of results of instrs of current instr,
                        ;*** corresponding to *int.destinations*
                        ;*** These two lists "buffer" all the memory writes of
                        ;*** the current instr being executed.
    *int.memory*
    ) )
```

```
;===============================================================================
;
; NADDR
;
; This module defines the NADDR operators and operator groups.
;
;===============================================================================

(eval-when (compile load)
    (include interpreter:interpreter-decls) )

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


;***===========================================================================
;***
;*** Operator functions
;***
;***===========================================================================

(defmacro operator:group ( op )
    `(get ,op 'int.operator:group) )

(defmacro operator:execute-function ( op )
    `(get ,op 'int.operator:execute-function) )

(defmacro operator:cost ( op )
    `(get ,op 'int.operator:cost) )

(defmacro def-operator ( op group cost . execute-code )
    (let ( (execute-name
              (if execute-code
                  (then (atomconcat 'operator:execute- op) )
                  (else () ) ) ) )
        (assert (litatom op) )
        (assert (litatom group) )
        (assert (inump cost) )
        `(eval-when (eval compile load)
            (:= (operator:group ',op)             ',group)
            (:= (operator:execute-function ',op) ',execute-name)
            (:= (operator:cost ',op)              ,cost)
            ,(if execute-code
                 (then `(defun ,execute-name ,,execute-code) )
                 (else () ) )
        ) ) )


;***===========================================================================
;***
;*** Operator Group functions
;***
;***===========================================================================

(defmacro group:declare-function ( group )
    `(get ,group 'int.group:declare-function) )

(defmacro group:part-table ( group )
    `(get ,group 'int.group:part-table) )

(defmacro group:properties ( group )
    `(get ,group 'int.group:properties) )
```



```
(defmacro def-group ( group . clauses )
    (loop (initial part-table       ()
                   declare-function ()
                   part-function    ()
                   code-list        ()
                   properties       () )
    (for clause in clauses)
    (do
        (assert (consp clause) )
        (? ( (== 'declare (car clause) )
             (:= declare-function (atomconcat 'group:declare- group) )
             (push code-list
                   `(:= (group:declare-function ',group) ',declare-function) )
             (push code-list
                   `(defun ,declare-function ,,(cdr clause) ) ) )
           ( (== 'properties (car clause) )
             (:= properties (append properties (cdr clause) ) ) )
           ( (&& (= 2 (length clause) )
                 (|| (numberp (cadr clause) )
                     (&& (consp (cadr clause) )
                         (for-every (x in (cadr clause) )
                            (numberp x) ) ) ) )
             (push part-table clause) )
           ( t
             (error (list clause "Invalid DEF-GROUP syntax.") ) ) ) )
    (result
        `(eval-when (eval compile load)
            (:= (group:part-table ',group) ',part-table)
            (:= (group:properties ',group) ',properties)
            ,,code-list) ) ) )


;***===========================================================================
;***
;*** Operation operations
;***
;***===========================================================================

(defmacro oper:operator  ( oper ) `(car    ,oper) )
(defmacro oper:operands  ( oper ) `(cdr    ,oper) )
(defmacro oper:dest      ( oper ) `(cadr   ,oper) )
(defmacro oper:oper1     ( oper ) `(caddr  ,oper) )
(defmacro oper:oper2     ( oper ) `(cadddr ,oper) )
(defmacro oper:group     ( oper )
    `(operator:group (oper:operator ,oper) ) )


(defun oper:part-description ( oper part-name )
    (cadr (assoc part-name (group:part-table (oper:group oper) ) ) ) )


;***===========================================================================
;***
;*** (OPER:PART OPER PART-NAME)
;***
;*** Returns the given part of an operation, e.g.
;***
;***    (oper:part '(iadd a b c) 'read)             => (b c)
;***    (oper:part '(if-ige x y 0.5 11 12) 'labels) => (11 12)
;***
;*** See the DEF-GROUP definitions of operator groups for the parts of
;*** each operator group.  Some parts common across groups are:
```

```
;***
;***    read        -- list of input operands to the operation (a vector is
;***                   considered an input operand).
;***    read1       -- the 1st input operand
;***    read2       -- the 2nd input operand
;***    written     -- the output operand of the operation (scalars only)
;***    labels      -- list of label operands (1 for goto, 2 for cond-jumps).
;***    label1      -- the first label operand
;***    label2      -- the second label operand
;***    probability -- the probability operand (EXPECT and cond jumps).
;***    bank        -- the optional memory bank of a VLOAD or VSTORE.
;***
;*** If there is no part of the given name, () is returned.
;***
;;;=============================================================================

(defun oper:part ( oper part-name )
    (oper:description:part oper (oper:part-description oper part-name) ) )

(defun oper:description:part ( oper description )
    (? ( (! description)
         () )
       ( (litatom description)
         (funcall description oper) )
       ( (inump description)
         (if (> description 0)
             (nth-elt oper (+ 1 description) )
             (nth     oper (- 1 description) ) ) )
       ( (consp description)
         (for (index in description) (save
            (if (numberp index)
                (nth-elt oper (+ 1 index) )
                (oper:description:part oper index) ) ) ) )
       ( t
         () ) ) )


;;;=============================================================================
;***
;*** (OPER:PROPERTY?  GROUP PROPERTY)
;*** (GROUP:PROPERTY? GROUP PROPERTY)
;***
;*** GROUP:PROPERTY returns true if PROPERTY was defined as one of the
;*** properties of GROUP in its DEF-GROUP.  OPER:PROPERTY?  returns true
;*** if PROPERTY is a defined property of the operation operator's group.
;*** Currently defined properties are:
;***
;*** conditional-jump -- the group defines conditional jumps.
;*** vector-reference -- the group defines vector references.
;*** pseudo-op        -- the group defines pseudo-operations that don't
;***                     affect state or flow of control.
;*** Examples:
;***
;***    (oper:property '(iadd a b c)   'conditional-jump) => ()
;***    (oper:property '(vload t2 v i) 'vector-reference) => t
;***
;;;=============================================================================

(defun group:property? ( group property )
    (memq property (group:properties group) ) )

(defun oper:property? ( oper property )
    (memq property (group:properties (oper:group oper) ) ) )
```



```
;;;=============================================================================
;***
;*** (LOOP (FOR-EACH-OPER-OPERAND-READ OPER NAME [INDEX]) ...)
;***
;*** Enumerates NAME through each scalar and vector name read by OPER.  If
;*** supplied, INDEX takes on the operand number of NAME.   Look Ma, no
;*** consing.
;***
;;;=============================================================================

(def-simple-loop-clause for-each-oper-operand-read ( clause )
    (let*( ( (for-each-oper-operand oper var index-var) clause)
           ( indices  (gensym) )
           ( oper-var (gensym) ) )

       (if (! (&& (<= (length clause) 4)
                  (>= (length clause) 3)
                  (litatom var)
                  (litatom index-var) ) )
           (error (list clause "Invalid FOR-EACH-OPER-OPERAND syntax.") ) )

       (if (! index-var) (then
           (:= index-var (gensym) ) ) )

       '( (initial ,var       ()
                   ,index-var ()
                   ,oper-var  ,oper
                   ,indices   (oper:part-description ,oper-var 'read) )
          (begin
               (if (inump ,indices) (then
                   (:= ,index-var (- -1 ,indices ) )
                   (:= ,oper-var  (nth ,oper-var (- 1 ,indices) ) ) ) ) )
          (while (if (inump ,indices) ,oper-var ,indices) )
          (do
               (if (inump ,indices) (then
                   (:= ,index-var (+ 1 ,index-var) )
                   (:= ,var       (pop ,oper-var) ) )
               (else
                   (:= ,index-var (pop ,indices) )
                   (:= ,var       (nth-elt ,oper-var (+ 1 ,index-var))))))
          (when (litatom ,var) ) ) ) )


;;;=============================================================================
;***
;*** (OPER:SUBSTITUTE-OPERAND OPER NEW-OPERAND OLD-OPERAND PART)
;***
;*** Substitutes NEW-OPERAND for OLD-OPERAND wherever OLD-OPERAND occurs
;*** in the operand positions described by PART:
;***
;***    (oper:substitute-operand '(iadd x y y) 'z 'read1) => (iadd x z y)
;***    (oper:substitute-operand '(iadd x y y) 'z 'read)  => (iadd x z z)
;***
;*** If PART is (), then all operand positions of OPER are examined for
;*** possible substitution.
;***
;;;=============================================================================

(defun oper:substitute-operand ( oper new-operand old-operand part )
    (let*( (part-description (oper:part-description oper part) ) )
```

```
      (?( (&& (numberp part-description)
              (>= part-description 0) )
          (loop (for operand in oper)
                (incr pos from 0)
          (save
              (if (&& (== old-operand operand)
                      (== pos part-description) )
                  new-operand
                  operand) ) ) )

        ( (&& (numberp part-description)
              (< part-description 0) )
          (loop (for operand in oper)
                (incr pos from 0)
          (save
              (if (&& (== old-operand operand)
                      (>= pos (minus part-description) ) )
                  new-operand
                  operand) ) ) )

        ( (consp part-description)
          (loop (for operand in oper)
                (incr pos from 0)
          (save
              (if (&& (== old-operand operand)
                      (memq pos part-description) )
                  new-operand
                  operand) ) ) )

        ( t
          (cons (car oper)
                (top-level-substq new-operand old-operand (cdr oper)))))))


;*****************************************************************************
;***
;*** Definitions of operator groups.
;***
;*****************************************************************************

(def-group def-block
    (in-variables  1)
    (out-variables 2)
    (properties    pseudo-op) )

(def-group param
    (written    1)
    (properties pseudo-op) )


(def-group dcl
    (variable    1)
    (length      2)
    (intial-list 3)
    (declare ( (dcl var length initial-list) )
        (variable:declare var length initial-list) ) )

(def-group live
    (read       -1)
    (properties pseudo-op) )

(def-group def
    (body       -1)
```

5

```
    (properties pseudo-op) )

(def-group use
    (body       -1)
    (properties pseudo-op) )

(def-group esc
    (properties pseudo-op) )

(def-group stop)

(def-group end
    (properties pseudo-op) )

(def-group noop)

(def-group expect
    (probability 1)
    (properties  pseudo-op) )

(def-group trace-fence
    (properties  pseudo-op) )

(def-group loop-start
    (labels              (1) )
    (label1              1)
    (expected-iterations 2)
    (properties          pseudo-op) )

(def-group loop-end
    (labels     (1) )
    (label1     1)
    (properties pseudo-op) )


(def-group loop-assign
    (read       -1)
    (read1      1)
    (read2      2)
    (written    1)
    (properties pseudo-op) )

(def-group label
    (labels     (1) )
    (label1     1)
    (properties pseudo-op)
    (declare ( oper )
        (label:declare (oper:dest oper) *int.pc*) ) )

(def-group goto
    (labels (1) )
    (label1 1) )

(def-group assert
    (read       -2)
    (read1      2)
    (read2      3)
    (compare-op 1)
    (properties pseudo-op)
    (declare ( (assert-op compare-op var1 var2) )
        (operand:declare var1)
        (operand:declare var2) ) )
```

6

```
(def-group if-then-else
    (read        (1 2) )
    (read1       1)
    (read2       2)
    (probability 3)
    (labels      (4 5) )
    (label1      4)
    (label2      5)
    (properties  conditional-jump)
    (declare ( (if-op var1 var2 prob label1 label2) )
        (operand:declare var1)
        (operand:declare var2) ) )

(def-group cond-jump
    (read        (1) )
    (read1       1)
    (probability 2)
    (labels      (3 4) )
    (label1      3)
    (label2      4)
    (properties  conditional-jump)
    (declare ( (cond-op var1 prob label1 label2) )
        (operand:declare var1) ) )

(def-group cond
    (properties  conditional-jump)
    (declare ( oper )
        (for (test in (cadr oper) ) (do
            (oper:declare test) ) ) ) )


(def-group vload
    (written 1)
    (read    (2 3) )
    (read1   2)
    (read2   3)
    (vector  2)
    (index   3)
    (bank    4)
    (properties vector-reference)
    (declare ( (vload dest vector index) )
        (operand:declare dest)
        (operand:declare index) ) )

(def-group vstore
    (written 1)
    (read    (1 2 3) )  ;*** Reading the vector prevents one VSTORE from
                        ;*** killing a previous VSTORE in dead-code removal.
                        ;*** This is like LOOP-ASSIGN.
    (read1   2)
    (read2   3)
    (vector  1)
    (index   2)
    (bank    4)
    (properties vector-reference)
    (declare ( (vstore vector index source) )
        (operand:declare index)
        (operand:declare source) ) )

(def-group one-in-one-out
    (written 1)
    (read    -2)
    (read1   2)
```



```
    (declare ( oper )
        (operand:declare (oper:dest  oper) )
        (operand:declare (oper:oper1 oper) ) ) )

(def-group two-in-one-out
    (written 1)
    (read    -2)
    (read1   2)
    (read2   3)
    (declare ( oper )
        (operand:declare (oper:dest  oper) )
        (operand:declare (oper:oper1 oper) )
        (operand:declare (oper:oper2 oper) ) ) )


;***============================================================================
;***
;*** Operator defintions
;***
;***============================================================================

(def-operator def-block         def-block         0)
(def-operator param             param             0)
(def-operator dcl               dcl               0)
(def-operator live              live              0)
(def-operator def               def               0)
(def-operator use               use               0)
(def-operator noop              noop              0)
(def-operator expect            expect            0)
(def-operator loop-start        loop-start        0)
(def-operator trace-fence       trace-fence       0)
(def-operator loop-end          loop-end          0)
(def-operator loop-assign       loop-assign       0)
(def-operator label             label             0)

(def-operator esc esc 0 ( expr )
    (eval `(progn ,,expr) ) )

(def-operator stop stop 0 ( operands )
    (:= *int.running?* () ) )

(def-operator goto goto 0 ( (label) )
    (:= *int.pc* (label:pc label) ) )

(def-operator truego cond-jump 1 ( (var prob label1 label2) )
    (? ( (!= 0 (operand:r-value var) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )

(def-operator falsego cond-jump 1 ( (var prob label1 label2) )
    (? ( (= 0 (operand:r-value var) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )

(def-operator ? cond 1 ( (tests) )
    (:= *int.pc* nil)
    (loop (for test in tests)
          (while (! *int.pc*) )
    (do
        (let ( (function (operator:execute-function
                           (oper:operator test) ) ) )
```

```
            (if function
                (funcall function (oper:operands test) ) ) ) ) ) )

(def-operator vload vload 1 ( (dest vector index) )
    (let ( (vbase  (variable:base   vector) )
           (length (variable:length vector) )
           (offset (operand:r-value index) ) )
        (if (|| (! (inump offset) )
                (>= offset length)
                (<  offset 0) )
        (then
            (if (memq *int.invalid-index-action* '(break warn) ) (then
                (msg 0 "INTERPRETER: VLOAD invalid index: "
                       vector " " index " = " offset t) ) )
            (if (== 'break *int.invalid-index-action*) (then
                (break-point vload-index) ) ) ) )
        (push *int.destinations* (operand:l-value dest)  )
        (push *int.results*      (memory (+ vbase offset)) ) ) )

(def-operator vstore vstore 1 ( (vector index source) )
    (let ( (vbase  (variable:base   vector) )
           (length (variable:length vector) )
           (offset (operand:r-value index) ) )
        (if (|| (! (inump offset) )
                (>= offset length)
                (<  offset 0) )
        (then
            (if (memq *int.invalid-index-action* '(break warn) ) (then
                (msg 0 "INTERPRETER: VSTORE invalid index: "
                       vector " " index " = " offset t) ) )
            (if (== 'break *int.invalid-index-action*) (then
                (break-point vload-index) ) ) ) )
        (push *int.destinations* (+ vbase offset) )
        (push *int.results*      (operand:r-value source) ) ) )



(def-operator inot one-in-one-out 1 ( oper )
    (if (!= oper 0) 0 1) )

(def-operator iand two-in-one-out 1 ( oper1 oper2 )
    (if (&& (!= oper1 0)
            (!= oper2 0) )
        1
        0) )

(def-operator ior two-in-one-out 1 ( oper1 oper2 )
    (if (|| (!= oper1 0)
            (!= oper2 0) )
        1
        0) )


(def-operator assign one-in-one-out 0 ( oper1 )
    oper1)



(def-operator iadd two-in-one-out 1 ( oper1 oper2 )
    (+ oper1 oper2) )

(def-operator fadd two-in-one-out 1 ( oper1 oper2 )
    (+ oper1 oper2) )
```



```
(def-operator isub two-in-one-out 1 ( oper1 oper2 )
    (- oper1 oper2) )

(def-operator fsub two-in-one-out 1 ( oper1 oper2 )
    (- oper1 oper2) )


(def-operator ineg one-in-one-out 1 ( oper )
    (- 0 oper) )

(def-operator fneg one-in-one-out 1 ( oper )
    (- 0 oper) )



(def-operator imul two-in-one-out 1 ( oper1 oper2 )
    (* oper1 oper2) )

(def-operator fmul two-in-one-out 1 ( oper1 oper2 )
    (* oper1 oper2) )



(def-operator idiv two-in-one-out 1 ( oper1 oper2 )
    (// oper1 oper2) )

(def-operator fdiv two-in-one-out 1 ( oper1 oper2 )
    (// oper1 oper2) )



(def-operator ieq two-in-one-out 1 ( oper1 oper2 )
    (if (= oper1 oper2) 1 0) )

(def-operator feq two-in-one-out 1 ( oper1 oper2 )
    (if (= oper1 oper2) 1 0) )

(def-operator if-ieq if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (= (operand:r-value var1)
            (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )

(def-operator if-feq if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (= (operand:r-value var1)
            (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )



(def-operator ine two-in-one-out 1 ( oper1 oper2 )
    (if (!= oper1 oper2) 1 0) )

(def-operator fne two-in-one-out 1 ( oper1 oper2 )
    (if (!= oper1 oper2) 1 0) )

(def-operator if-ine if-then-else 1 ( (var1 var2 prob label1 label2) )
```

```
    (? ( (!= (operand:r-value var1)
             (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )


(def-operator if-fne if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (!= (operand:r-value var1)
             (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )




(def-operator igt two-in-one-out 1 ( oper1 oper2 )
    (if (> oper1 oper2) 1 0) )

(def-operator fgt two-in-one-out 1 ( oper1 oper2 )
    (if (> oper1 oper2) 1 0) )

(def-operator if-igt if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (> (operand:r-value var1)
            (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )

(def-operator if-fgt if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (> (operand:r-value var1)
            (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )



(def-operator ige two-in-one-out 1 ( oper1 oper2 )
    (if (>= oper1 oper2) 1 0) )

(def-operator fge two-in-one-out 1 ( oper1 oper2 )
    (if (>= oper1 oper2) 1 0) )

(def-operator if-ige if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (>= (operand:r-value var1)
             (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )

(def-operator if-fge if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (>= (operand:r-value var1)
             (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )



(def-operator ile two-in-one-out 1 ( oper1 oper2 )
    (if (<= oper1 oper2) 1 0) )
```



```
(def-operator fle two-in-one-out 1 ( oper1 oper2 )
    (if (<= oper1 oper2) 1 0) )

(def-operator if-ile if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (<= (operand:r-value var1)
             (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )

(def-operator if-fle if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (<= (operand:r-value var1)
             (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )


(def-operator ilt two-in-one-out 1 ( oper1 oper2 )
    (if (< oper1 oper2) 1 0) )

(def-operator flt two-in-one-out 1 ( oper1 oper2 )
    (if (< oper1 oper2) 1 0) )

(def-operator if-ilt if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (< (operand:r-value var1)
            (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )

(def-operator if-flt if-then-else 1 ( (var1 var2 prob label1 label2) )
    (? ( (< (operand:r-value var1)
            (operand:r-value var2) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )



(def-operator ie0mod two-in-one-out 1 ( x m )
    (if (== 0 (mod x m) ) 1 0) )

(def-operator if-ie0mod if-then-else 1 ( (x m prob label1 label2) )
    (? ( (== 0 (mod (operand:r-value x)
                    (operand:r-value m) ) )
         (:= *int.pc* (label:pc label1) ) )
       ( label2
         (:= *int.pc* (label:pc label2) ) ) ) )




(def-operator assert assert 0 ( (comparison-operator var1 var2) )
    (if *int.assertion-failed-action* (then
        (let*( (function (operator:execute-function comparison-operator) )
               (result   (if function
                             (funcall function (operand:r-value var1)
                                               (operand:r-value var2) )
                             0) ) )
          (if (= result 0) (then
```

```
            (msg 0 "NADDR: ASSERT failed. ("
                comparison-operator " "
                var1 " = " (operand:r-value var1) " "
                var2 " = " (operand:r-value var2) ")" t)
            (if (== 'break *int.assertion-failed-action*) (then
                (break-point assertion-failed) ) ) ) ) ) ) )


(def-operator iabs one-in-one-out 1 ( oper1 )
    (if (>= oper1 0)
        oper1
        (- 0 oper1) ) )

(def-operator fabs one-in-one-out 1 ( oper1 )
    (if (>= oper1 0.0)
        oper1
        (- 0 oper1) ) )



(def-operator iexp two-in-two-out 1 ( oper1 oper2 )
    (^ oper1 oper2) )

(def-operator fexp two-in-two-out 1 ( oper1 oper2 )
    (flonum (expt oper1 oper2) ) )



(def-operator sqrt one-in-one-out 1 ( oper1 )
    (sqrt oper1) )

(def-operator sin one-in-one-out 1 ( oper1 )
    (sin oper1) )

(def-operator cos one-in-one-out 1 ( oper1 )
    (cos oper1) )

(def-operator tan one-in-one-out 1 ( oper1 )
    (tan oper1) )



(def-operator fix one-in-one-out 1 ( oper1 )
    (fix oper1) )

(def-operator float one-in-one-out 1 ( oper1 )
    (flonum oper1) )


(def-operator imin two-in-one-out 1 ( oper1 oper2 )
    (min oper1 oper2) )

(def-operator fmin two-in-one-out 1 ( oper1 oper2 )
    (min oper1 oper2) )



(def-operator imax two-in-one-out 1 ( oper1 oper2 )
    (max oper1 oper2) )

(def-operator fmax two-in-one-out 1 ( oper1 oper2 )
    (max oper1 oper2) )
```



```
(def-operator bitrev two-in-one-out 1 ( i bits )
    (loop (initial m1 1
                   m2 2
                   j  0)
    (do
        (if (>= (- i (* m2 (// i m2) ) ) m1) (then
            (:= j (+ j (// bits m2) ) ) ) ) )
    (while (< m2 bits) )
    (next m1 m2
          m2 (+ m2 m2) )
    (result j) ) )
```