(**** (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
;;;
(defun binary-arithmetic-operator:opcode*
  ('( + IADD - ISUB * IMUL // IDIV +$ FADD -$ FSUB •* FMUL //$ FDIV nln IHIN mln$ FHIN (ax IHAX max$ FHAX (bit-reverse BITREV)
  '(- INEG =$ FNEG (exp IEXP =$ FEXP (abs IABS =$ FABS (cos Icos $ sinh $ sinh (sqrt SQRT (float FLOAT (fix FIX)
  ))

(defun unary-arithmetic-operator:opcode*
(def-coapile-functlon (assert) 
(expr dest node t-label f-label prob) 
(tic.syntax-assert expr (= 3 (length (cadr expr)) )) 
(let* ( (assert (operator operandi operand2) ) 
(expr) 
( () value-opcode control-opcode) 
(assoc operator »arlthnetlc-comparison:opcode*)) ) (tic.syntax-assert expr value-opcode) 
(tlc.enlt '(assert .dest .value-opcode) ) 
)

(def-coapile-functlon ! ( expr dest node t-label f-label prob ) 
(tic.syntax-assert expr (= 2 (length expr) ) ) (let ( ( () operand) expr) 
(new-operator) 
(actual-dest) ) 
(caseq node 
(for-control 
(tic.conp-expr operand () node f-label t-label (- 1.0 prob) ) ) 
(for-effect 
(tic.conp-expr operand () aode () ()()) 
(tlc.coap-expr (cadr operand) deat aode t-label f-label prob) ) 
(tlc.conp-expr (cadr (assoc (car operand) *bool-operator:negation*)) ) 
(tic.conp-expr '(.new-operator ,,(cdr operand) ) deat node t-label f-label prob) ) 
(tlc.enlt '(lnot .dest ,actual-dest) ) 
dest) ) ) )

(def-coapile-functlon def-block ( expr dest node t-label f-label prob ) 
(let 
(syntax-assert expr (ftft (<= 4 (length expr) ) (llstp (cadr expr) ) (llstp (caddr expr) ) ) ) 
(( (() paraas llve-vars . block-exprs) 
(case (cadr expr) 
(if (cdar expr) 
(tlc.coap-expr (cadr expr) deat mode t-label f-label prob) ) 
(tlc.coap-expr (car expr) deat mode t-label f-label prob) ) 
(tlc.coap-expr (cadr (assoc (car expr) *bool-operator:negation*)) ) 
(tlc.coap-expr '(.new-operator ,,(cdr expr) ) deat mode t-label f-label prob) ) )

(cdr expr) 
)

(loop 
(Initial rest (cdr expr) ) 
while rest) 
(do 
( ( (! (cdr rest) ) ;*** laet expreaaion in block? 
(tlc.coap-expr (car rest) deat mode t-label f-label prob) 
(tic.conp-expr (car rest) () 'for-effect ()()()))) 
(next rest (cdr rest) ) 
(result dest) ) ) )

(DEF-BLOCK ( paraai paraa2 
(llve-varl Hve-var2 ...) el e2 
en))

Paraai and llve-varl can be either a sym bol (a variable name) or an array name such as (array-name dlnl din2 ...) where the meaning of the array is as in DECLARE...

(cdr expr) 
)

(loop 
(Initial rest (cdr expr) ) 
while rest) 
(do 
( ( (! (cdr rest) ) ;*** laet expreaaion in block? 
(tlc.coap-expr (car rest) deat mode t-label f-label prob) 
(tic.conp-expr (car rest) () 'for-effect ()()()))) 
(next rest (cdr rest) ) 
(result dest) ) ) )

(DEF-BLOCK ( paraai paraa2 
(llve-varl Hve-var2 ...) el e2 
en))

Paraai and llve-varl can be either a sym bol (a variable name) or an array name such as (array-name dlnl din2 ...) where the meaning of the array is as in DECLARE...
(def-compile-function :if-expr (expr dest node t-label f-label prob)
(let ((bool-expr (tic.variable-1-value expr))
(var-ref (car var))
(location (tlc.coap-expr expr2 var-ref 'for-value ()()())))
(if (eq (consp (car rest-if)) (if (consp var))
(then (desetq (if expect-prob) (car rest-if))
(pop rest-if))
(then (desetq (if expect-prob) (car rest-if))
(pop rest-if))
(tlc.syntax-assert expr (== 'flonum (nuatype expect-prob)))))
(tlc.syntax-assert expr (! rest-if))
(tlc.coap-expr bool-expr () 'for-control labell label2 expect-prob)
(tlc.enlt '(label .labell))
(tlc.coap-expr '(block ..exprla)
dest node t-label f-label prob)
(if (!== 'for-control node)
(tlc.enlt '(label .labels) )
else)
(dest) )

(IF-GO bool expect-prob true-label [false-label])
Crocko statement used internally by such as LOOP.
Let's us use IF- addr operators directly without needing
a peephole optlizer for eliminating jumps to jumps).
```
(defun loop-compile-function (loop-name (iterations-guess (loop-name)) (unroll (loop-name)) (decr-var (loop-name)) (step-var (loop-name)) (body-code (loop-name)) (result-code (loop-name)) (expr (loop-name)))
  (let ((loop-label (loop-name))
        (result-label (loop-name))
        (leave-label (loop-name))
        (continue-label (loop-name))
        (final-dest (loop-name)))
    ;; The special expression (UNROLL-INDEX loop-name) expands to the
    ;; unrolling of the named loop (starting with 1).
    (defstruct loop-desc
      (name (loop-name))
      (top-label (loop-name))
      (continue-label (loop-name))
      (leave-label (loop-name))
      (dest (loop-name))
      (mode (loop-name))
      (t-label (loop-name))
      (f-label (loop-name))
      (prob (loop-name)))

(defun loop-compile-function (expr dest node t-label f-label prob)
  (let ((loop-name (loop-name))
        (iterations-guess (loop-name))
        (unroll (loop-name))
        (decr-var (loop-name))
        (step-var (loop-name))
        (body-code (loop-name))
        (result-code (loop-name))
        (top-label (loop-name))
        (result-label (loop-name))
        (leave-label (loop-name))
        (continue-label (loop-name))
        (final-dest (loop-name)))
    ;; The special expression (UNROLL-INDEX loop-name) expands to the
    ;; unrolling of the named loop (starting with 1).
    (defstruct loop-desc
      (name (loop-name))
      (top-label (loop-name))
      (continue-label (loop-name))
      (leave-label (loop-name))
      (dest (loop-name))
      (mode (loop-name))
      (t-label (loop-name))
      (f-label (loop-name))
      (prob (loop-name)))
```

((:= unroll keyword) )
(else
  (caseq keyword
    (no-back-jump
      (:= back-jump? () ) )
    (no-exit-tests
      (:= exit-tests? () ) )
    (no-fold
      (:= fold-step-vars? () )
      (t
        (tcl.syntax-assert clause nil)) )
    (t
      (tcl.syntax-assert clause nil)))
  (iterations-guess
    (tcl.syntax-assert clause (number (cadr clause) )
     (:= iterations-guess (flonum (cadr clause) )
      (:= leave-prob (/ 1.0 iterations-guess) ) )
    (result
      (:= result-code (cdr clause) )
    )
  )
)

(let ( ( (step var from init-expr using using-expr
          while while-expr)
         clause)
        (tcl.syntax-assert clause
          (== 'from from)
          (== 'using using)
          (== 'until while)
          (== 'while while) )
       (push init-code '(:= .var .init-expr) )
       (push step-vars var)
       (caseq while
         (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)
                   (tcl.syntax-assert clause
                     (== 'from from)
                     (== 'using using)
                     (== '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 )))
         (push step-vars var)
         (caseq while
           (while
            (push body-code
              (if-go (! (cadr clause) )
                ,leave-prob ,result-label ))
           (until
            (push body-code
              (if-go ,cadr clause
                ,leave-prob ,result-label )))
         (t
          (tcl.syntax-assert clause nil))
       )
    )
    )
)

(by) (by-expr) (by-temp) )
(tcl.syntax-assert clause (= 'from from)
(if rest (then
  (deseq (to-to-expr . rest) rest)
  (tcl.syntax-assert clause (= 'to to) ) )
(if rest (then
  (deseq (by-expr . rest) rest)
  (tcl.syntax-assert clause (= 'by by) )
  (else
    (= by-expr 1) )
  (tcl.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 (tcl.gensym '%) )
      (push init-code '(:= 'temp ,to-temp) )
    )
    (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 (tcl.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)))))
    )
    (if (numberp by-expr) (then
      (:= by-temp by-expr) )
    (else
      (:= by-temp (tcl.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
      (tcl.syntax-assert clause () )
    )
  )
  (t
    (tcl.syntax-assert clause nil))
  )
)

(increment decr)
(let ( ( (incr var from init-expr . rest)
           clause)
          (to) (to-expr) (to-temp)
        )
)
:: begin of loop proper.
(tlc.exit (label .top-label))
(if back-jump (then
(tlc.exit (loop-start .top-label .iterations-guess) ) )
(loop (initial continue-label-1 ()
(incr i from 1 to unroll)
(next continue-label-1 (atonaconcat continue-label 1) )
(do *** Push a loop descriptor for any LEAVES or CONTINUES
(push *tlc.loop-stack*
(loop-desc: new name
  top-label top-label
  leave-label leave-label
  continue-label continue-label-1
  dest dest
  mode mode
  t-label t-label
  f-label f-label
  prob prob
  back-jump? back-jump?) )
(tlc.enlt '(label continue-label-1) )
;*«« Compile the step code, folding INCR/DECR vars.
(loop (for () var step-expr) in step-code) (do
(if (& & fold-step-vars)
   (> unroll 1)
   (mesq var incr-decr-vars) )
     (let (((- () 1) var-step-expr)
        (var-i (atonaconcat % var 1))
        (var-i+1 (atonaconcat % var (+ 1 1)))
        (if (= i 1) (then
(tlc.comp-expr
  '(:= .var-1 .var)
  () 'for-effect () () )
(tlc.comp-expr
  '(:= .var-i+1 (- .var-1 (1 . delta)) )
))
(if back-jump? (then
(tlc.exit (list 'trace-fence) )
(tlc.exit (goto .top-label) )
))
(tlc.enlt '(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 VARI VARI)
  (GOTO RESULT-LABEL)
  ...
  (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 seemingly useless assignments allow variable renaming
*«« to rename the STEP/INCR/DECR vars in each unrolling.
(loop (incr i from 1 to unroll) (do
(tlc.exit (label .(atonaconcat result-label 1) )
(loop (for var in (union incr-decr-vars step-vars) ) (do
(tlc.exit (assign .var .var) )
(tlc.exit (goto .result-label) )
))
)
(tlc.enlt '(label .result-label) )
(if back-jump? (then
(tlc.exit (loop-end .top-label) )
))
(if (! result-code)
(:= result-code '() )
(tlc.comp-expr
  '(:= .result-code .0)
  () 'for-effect () () )
)
(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) )
))
11
PS: <C.S.BULLDOG.INTERPRETER>COMPILE-FUNCTIONS.LSP.13
12
11
12
(def-compile-function leave (expr dest node t-label f-label prob)
(let ((() loop-name-result-expr)
  expr)
  (tlc.syntax-assert expr (\(\#\#\) (= 3 (length expr))
    (= mode 'for-effect))
  (if (loop-desc:back-jump? loop-desc) (then
    (tlc.exit 'loop-end ,(loop-desc:top-label loop-desc) )
  )
  (tlc.coap-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.exit 'goto ,((loop-desc:leave-label loop-desc) )
  )
  )
)

(*** (CONTINUE loop-name)

(*** Find the loop descriptor on the loop stack that matches \(\text{LOOP-LABEL}\)

(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 addr-code to evaluate expression.

(*** (exp expr dest node t-label f-label prob)

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

(*** (EXPECT number)

(*** (DECLARE array-name dim1 dim2 ... dian [[(INITIAL val1 val2 ...)]]

(*** Each of the \(\text{dimI}\) can be either a pair (lower upper) or else a number,

(*** which is the same as \(0 \text{ number-1}\). The optional \(\text{INITIAL}\) clause

(*** specifies initial values \(\text{vali}\) for the array (stored in row major

(*** order, \(0\)'s used for padding).
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 (car dim) )
            (else
            (:= initial (cdr dim) )
            (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« 0.6) ;** Default expect to use for conditional jumps if none is specified.
(defvar *ctl.naddr-code* () ) ;** Reverse list of naddr code being generated.
(defvar *ctl.loop-stack* ) ;** Stack of descriptors of nested loops.
(defvar *ctl.gensya-counter* ) ;** Our own gensya counter, for repeatable results.
(defvar «tlc.iterations-guess-default» 100.0) ;** Default value for expected number of iterations of a loop
(resolve «tlc.reserved-words» () ) ;** Default expect to use for conditional jumps if none is specified.

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

(defun compile-tiny-lisp ( expr «optional (node 'for-effect) » )
  (let ( (dest (tlc.coap-expr expr «node 'labell » label1 label2 «) » ))
    (tlc.emit (list 'stop )
      (:= *ctl.naddr-code* (dreverse *ctl.naddr-code*) ) ) )

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

(defun ctl ( expr «optional (node 'for-value) » )
  (let ( (dest (tlc.coap-expr expr «node 'labell » label1 label2 » ))
    (tlc.emit (list 'stop )
      (:= *ctl.naddr-code* (dreverse *ctl.naddr-code*) ) ) )

**Our own internal gensya that will generate the same "unique" symbols each time we run the compiler. Always interns the symbol; prefix PS:<C.S.BULLDOG.INTERPRETER>COMPILER.LSP.15**
Array declarations and "symbol" table.

Sample array declaration: (A (1 6) (2 14) (3 30))

Let n be the number of dimensions.
Let $S_i$ be the size of dimension $i$ (counting from 1, left to right)
Let $L_i$ be the lower bound of each dimension.

The index calculation for $(A I 1 I 2 \ldots I n)$ is:
$$I_n + I_{n-1}S_n + I_{n-2}S_nS_{n-1} + \ldots + I_1S_nS_{n-1}\ldots S_2 - 0$$

where
$$0 = (L_n + L_{n-1}S_n + \ldots + L_1S_nS_{n-1}\ldots S_2)$$

The ARRAY: INDEXING-LIST for the array name contains:
$$(S_nS_{n-1}\ldots S_2 \ldots S_nS_{n-1} \ldots S_1)$$

ARRAY: INDEXING-OFFSET contains 0.

(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 (lltatom array-name))
  (push »tie.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
          (numberp (car dimension))
          (numberp (cadr dimension))
          (numberp (cadre dimension))
          (numberp (car dimension))))
        (:= si (+ 1 (- (cadr dimension) (car dimension))))
        (:= li (car dimension))
        (else
          (tlc.syntax-assert bounds-list (numberp dimension))
          (:= si dimension)
          (:= li 0))))
    (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 (lltatom 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)
  (if (numberp expr)
    (lltatom expr)
    (consp expr)
    (litaton (car expr)
      (tlc.array:declared? (car expr)))
  )

* Recursively Compile a Tiny Lisp expression.

(defun tlc.comp-expr (expr dest node t-label f-label prob)
  (let* ( (function () )
    (scalar-de8t () )
    (compiled-location O )
    (if (! prob)
      (= prob »expect-default») )
  (?? ;*** a variable or array reference?
    (tlc.variable? expr)
    (list* (funcion () )
      (scalar-dest () )
      (compiled-location () )
    (if (! prob)
      (:= prob »expect-default») )
    (expr dest mode t-label f-label prob))

PS:<C.S.BULLDOG.INTERPRETER>COMPILER.LSP.15
*** If this is for-value and the dest is a vector reference, compile the variable into the () destination.
(if (= node 'for-value)
  (= scalar-dest
    (if (consp dest) () dest))
)

*** compile the variable
(= compiled-location
  (tic.comp-variable expr scalar-dest node t-label f-label prob))

*** a form?
(if (consp expr)
  (= compiled-location
    (litcons (car expr))
    (= function (reserved-word:compile-function (car expr)))))

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

*** compile the expression
(= compiled-location
  (funcall function expr scalar-dest node 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 (&= (= node 'for-value)
    (consp dest))
  (tlc.emit 'vstore ,(car var-loc) , (cadr var-loc) )
)

PS: <C.S.BULLDOG.INTERPRETER>COMPILER.LSP.15
Declarations of global variables for the Tiny Lisp Compiler. This file should be INCLUDED.

(declare (special
  *expect-default*
  *tlc.naddr-code*
  *tlc.loop-stack*
  *tlc.gensym-counter*
  *iterations-guess-default*
  *tlc.arrays*
  *tlc.reserved-words*
  *tlc.compile-assertions*
) )
Parallel Naddr Interpreter

Warning: this version of NADDR is not yet (and may never be) documented 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 together with its 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*))
       (:= *int.memory-length* (vectorlength *int.memory*))
    (msg 0 "INTERPRETER: Re-initializing memory to be *
       *int.memory-length* "words.")
    (:= *int.memory* (makevector *int.memory-length*))
  )
  (:= *int.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)
  (reprop label 'lnt.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* 0)
  )
)

;*** 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)
  (reprop var 'int.variable:length)
  (reaprop var 'lnt.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 1 from (variable:base var)
      to (+ -1 (+ (variable:base var) (variable:length var) ) )
    )
    (do
      (lf (! rest-initial-list)
        (:= rest-initial-list initial-list) )
      (pop rest-initial-list value)
      (:= (memory 1) value) )
  )
)

(defun variables:initialize ()
  (lf (! (boundp '»Int.variables») ) (then
    (:= »int.variables* () ) )
  )
  (for (var In »int.variables*) (do
    (variable:destroy var) )
  ) (:= *int.variables* O )
)

;;; Operand functions
;;;
(defun operand:1-value ( operand )
  (lf (numberp operand)
    operand
    (variable:base operand) )
)

(defun operand:r-value ( operand )
  (if (numberp operand)
    operand
    (memory (operand:1-value operand) )
  )
)

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

(defun var ( operand )
  'variable:declare operand 1 '(0) )
)

;;; The Instruction queue remembers the last n instructions executed.
;;;
(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 ( 1 )
    "(if (== (++ ,1) *int.instr-q-max-length*) (then
      (:= ,1 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 1 from 1 to *int.instr-q-length*)
    (initial 1 *int.instr-q-head*)
    (save ([]) *int.instr-q-vector* 1) )
    (do (instr-q:incr-index 1) )
  )
)

;;; Instruction operations
;;;
(defun instr:normalize ( instr
  (if (consp (car .instr) )
    instr
    (cons .instr 0 ) )
)

(defun oper:declare ( oper )
  (let ( (function (group:declare-function
    operator:group (oper:operator oper) ) )
  )
    (lf 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
    (lf (!= 0 (operator:cost (oper:operator oper) ) )
      (result () )
    )
  )
)

(defun instr:execute ( instr
  (:= »Int.destinations* ()
  (:= »Int.results* ()
  (if (instr:costsp instr) (then
    (++ »Int.instruction-count*)
    (lnstr-q:add instr)
    (loop
      )))
    (result () )
  )
)

;;; Vector implementing the queue.
(*int.instr-q-vector* ;«** Vector implementing the queue.
 (*int.instr-q-max-length*) )

(*int.instr-q-length* ;** Max allowable length of the queue
 *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)

3
PS:<C.S.BULLDOG.INTERPRETER>INTERPRETER.LSP.29
(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 (listatom 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 1 from (variable:base (car var))
  to (+ -1 (+ (variable:base (car var)) (variable:length (car var))))
  (do
    (msg " ")
    (loop (initial remainder (- 1 (variable:base (car var)))
      (for (lower upper size) in dimensions)
      (do
        (msg " ")
        (loop (initial remainder (- 1 (variable:base (car var))))
          (for (lower upper size) in dimensions)
          (do
            (msg " ")
            (:= remainder (\ remainder size))
          )
        )
      )
    )
  )
)

) ) )

PS:<C.S.BULLDOG.INTERPRETER>INTERPRETER.LSP.29
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
  "int.instruction-count" ;*** encountered a STOP instruction.
  "int.operation-count" ;*** # of instrs executed during the last
  "int.invalid-index-action" ;*** call to interpret
  "int.assertion-failed-action" ;*** # of operations executed
)

*** Options defined elsewhere with DEF-OPTION:

*** Declarations for own variables shared by parts of the Interpreter ***

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

1
PS:<C.S.BULLDOG.INTERPRETER>INTERPRETER-DECLS.LSP.4
This module defines the NADDR operators and operator groups.

(defmacro def-group (group . clauses)
  (loop (initial part-table ()
          declare-function ()
          part-function ()
          code-list ()
          properties ()
          (for clause in clauses)
            (do
              (assert (consp clause)
                  (if (== (car clause) 'declare)
                  (push code-list
                        (atomconcat 'group:declare-function (cadr clause)
                        (caddr clause))
                  (push code-list
                        (defun .declare-function ..(cdr clause) )
                        (if (== (car clause) 'properties)
                  (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) )
                        )))
              )))
  )
  ))

(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)

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

1

PS:<C.S.BULLDOG.INTERPRETER>NADDR.LSP.18

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)
  (let* ((description)
         (littatom description)
         (nuth oper (+ 1 description))
         (nth oper (- 1 description))
         (consop description)
         (for (index in description) (save
              (if (nunp index)
                  (nuth-elt oper (+ 1 index))
                  (nth oper (- 1 index)))))
         ( operate:part-description oper part-name))
)

(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 x y y) 'conditional-jump) => ()
  (oper:property '(vload t2 v 1) 'vector-reference) => t

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

(defun oper:property? (oper property)
  (meq property (group:properties (oper:group oper)))
)

(def simple-loop-clause for-each-oper-operand-read (clause)
  (let*( (for-each-oper-operand-read oper var index-var)
         (indices (gensym))
         (oper-var (gensym))
         (if (! (lt (length clause) 4) (lt (length clause) 3)
               (littatom var)
               (littatom index-var))
           (error (list clause "Invalid FOR-EACH-OPER-OPERAND syntax."))
         (if (! index-var) (then
             (:= index-var (gensym))
           )
         ))
  (for (index in description) (save
      (if (nunp index)
          (nuth-elt oper (+ 1 index))
          (nth oper (- 1 index)))
      (operate:part-description oper index)))
  (while (if (nunp . indices) . oper-var . indices)
    (do
      (if (nunp . indices) (then
        (:= lndex-var (+ 1 ,index-var))
        (:= ,var (pop ,oper-var))
      )
      (else
        (:= ,index-var (pop . indices))
        (:= ,var (nth-elt ,oper-var (+ 1 . index-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 'read1) => (iadd x x y)

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))
         (indices (gensym))
         (oper-var (gensym))
         (if (! (length clause) 4) (if (! (length clause) 3)
           (error (list clause "Invalid FOR-EACH-OPER-OPERAND syntax."))
         )
         (if (! index-var) (then
           (:= index-var (gensym))
         )
         )
  (for (index in description) (save
    (if (nunp index)
        (nuth-elt oper (+ 1 index))
        (nth oper (- 1 index)))
    (operate:part-description oper index)
  )
  (while (if (nunp . indices) . oper-var . indices)
    (do
      (if (nunp . indices) (then
        (:= lndex-var (+ 1 ,index-var))
        (:= ,var (pop ,oper-var))
      )
      (else
        (:= ,index-var (pop . indices))
        (:= ,var (nth-elt ,oper-var (+ 1 . index-var))))
    )
  )
)
(\( ? (\textbf{numberp}\ \textit{part-description}) \))
(\( (> \textit{part-description} \ 0) \))
(loop (for \textit{operand} in \textit{oper})
 incr \textit{pos} from 0)
(save
 (if (\( \textbf{numberp}\ \textit{old-operand}\ \textit{operand}\))
 (new-operand
 \textit{operand}\))
)

\( (\textbf{numberp}\ \textit{part-description}) \)
(\( (< \textit{part-description} \ 0) \))
(loop (for \textit{operand} in \textit{oper})
 incr \textit{pos} from 0)
(save
 (if (\( \textbf{numberp}\ \textit{old-operand}\ \textit{operand}\))
 (new-operand
 \textit{operand}\))
)

\( (\textbf{consp}\ \textit{part-description}) \)
(loop (for \textit{operand} in \textit{oper})
 incr \textit{pos} from 0)
(save
 (if (\( \textbf{consp}\ \textit{old-operand}\ \textit{operand}\))
 (new-operand
 \textit{operand}\))
)

\( (\textbf{t}) \)
(cons (car \textit{oper})
 (top-level-substq new-operand old-operand (cdr \textit{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)
 (initial-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)
 (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 expect
 (probability 1)
 (properties pseudo-op))

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

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

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

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

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

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

PS:<C.S.BULLDOG.INTERPRETER>NADDR.LSP.18
(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))
  (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 (cdr 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))
  (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)
  (read2 3)
  (properties)
  (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)
  (properties)
  (declare ( oper )
    (operand:declare (oper:dest oper))
    (operand:declare (oper:oper1 oper))
    (operand:declare (oper:oper2 oper))
  )

---

(def-operator def-block def-block 0)
(def-operator param param 0)
(def-operator incl incl 0)
(def-operator def def 0)
(def-operator use use 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) ) )
    (:= *int.pc* (label:pc label12) )
    (label12
      (:= *int.pc* (label:pc label12) ) )
  )
)
(def-operator ? cond 1 ( (testa) )
  (:= »int.pc* nil)
  (loop (for test in tests)
    (while (1 *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 (|| (! (lnunp offset) )
             (>= offset length)
             (< offset 0) )
      (if (seqq +int.invalid-index-action* '(break warn) )
        (asm 0 "INTERPRETER: VLOAD invalid index:
             vector " vector " index " = " offset t) )
      (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 (|| (! (lnunp offset) )
             (>= offset length)
             (< offset 0) )
      (if (seqq +int.invalid-index-action* '(break warn) )
        (then
          (asm 0 "INTERPRETER: VSTORE invalid index: 
             vector " vector " Index " = " offset t) )
      (push +int.destinations* (+ vbase offset) )
      (push +int.results* (operand:r-value source) )
    )
  )
)

(def-operator lnot one-in-one-out 1 ( oper )
  (lf (!= oper 0) 0 1) )

(def-operator land two-in-one-out 1 ( oper1 oper2 )
  (lf (!= oper1 0) l= oper2 0 ) )

(def-operator lor 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 lneg one-in-one-out 1 ( oper )
  (- 0 oper) )

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

(def-operator lmul 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-ln-one-out 1 ( oper1 oper2 )
  (// oper1 oper2) )

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

(def-operator ileq two-ln-one-out 1 ( oper1 oper2 )
  (lf (= oper1 oper2) 1 0) )

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

(def-operator lf-leq lf-then-else 1 ( (var1 var2 prob labell label2) )
  (? ( (= (operand:r-value var1)
         (operand:r-value var2) )
       (label:pc labell) )
       (label2
        (= +int.pc* (label:pc label2) ) )
  )
)

(def-operator lf-leq lf-then-else 1 ( (var1 var2 prob labell label2) )
  (? ( (= (operand:r-value var1)
         (operand:r-value var2) )
       (label:pc labell) )
       (label2
        (= +int.pc* (label:pc label2) ) )
  )
)

(def-operator ine two-ln-one-out 1 ( oper1 oper2 )
  (if (!= oper1 oper2) 10) )

(def-operator fne two-ln-one-out 1 ( oper1 oper2 )
  (lf (!= oper1 oper2) 10) )

(def-operator lf-lne lf-then-else 1 ( (var1 var2 prob labell label2) )
  (if (!= oper1 oper2) 1 0) )

PS:<C.S.BULLDOG.INTERPRETER>NADDR.LSP.18
(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 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 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 if-igt two-in-one-out 1 (oper1 oper2)
  (if (> oper1 oper2) 1 0)
)

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

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

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

(def-operator if-llt 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 if-llt 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 ieOnod two-ln-one-out 1 ( x n )
  (if (= 0 (nod x n) ) 1 0)
)

(def-operator if-llt 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 if-llt 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 assert assert 0 (comparison-operator var1 var2)
  (if intASSERTION-FALLEN-ACTION
     (then
       (let*( (function (operator:execute-function comparison-operator) )
             (result (lf function (operand:r-value var1)
                       (operand:r-value var2) )
          )
         (if (= result 0) (then
          (if (= result 0) (then
          (if (= result 0) (then
          )
        )
      )
    )
  )
)
)

(def-operator assert assert 0 (comparison-operator var1 var2)
  (if intASSERTION-FALLEN-ACTION
     (then
       (let*( (function (operator:execute-function comparison-operator) )
             (result (lf function (operand:r-value var1)
                       (operand:r-value var2) )
          )
         (if (= result 0) (then
          (if (= result 0) (then
          (if (= result 0) (then
          )
        )
      )
    )
  )
)
)

(def-operator assert assert 0 (comparison-operator var1 var2)
  (if intASSERTION-FALLEN-ACTION
     (then
       (let*( (function (operator:execute-function comparison-operator) )
             (result (lf function (operand:r-value var1)
                       (operand:r-value var2) )
          )
         (if (= result 0) (then
          (if (= result 0) (then
          (if (= result 0) (then
          )
        )
      )
    )
  )
)
)
(assert-failed (comparison-operator "=" var1 "= (operand:r-value var1) " var2 "= (operand:r-value var2) ") t))

(if (and (= var1 var2) (= var2 3))
    (break-point assertion-failed))

(def-operator labs 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)
    (flicnum (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)
    (flicnum oper1))

(def-operator lmin 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 lmax 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 (bits)
    (loop (initial m1 1 m2 2 j 0)
        (do (if (> m1 m2)
            (j (+ j (// bits m2)))
            (next m1 m2 m2 (+ m2 m2)))
            (result j))))