```
;===========================================================================
;
; (FG.DISAMBIGUATE-BANKS)
;
; This module implements the bank disambiguation.  It assumes that the
; assertions have already been processed and that loop assignments have
; already been inserted for derivations to work ok.
;
; Each vector reference is examined in turn.  The residue of the derivation
; of the index is calculated, and if it is non-nil (known) the NADDR source
; operation is destructively modified to include the bank.
;
; Warning:  If we ever change this to do non-destructive modification, then
; remember to recalculate the hash table used in the disambiguator.
;
;===========================================================================

(include flow-analysis:flow-analysis-decls)


(defun fg.disambiguate-banks ()

    (loop (for-each-stat stat)
          (when (stat:property? stat 'vector-reference) )
          (initial heading-printed? () )
    (do
        (if-let ( (bank (de:residue (stat:index-derivation stat)
                                    *number-of-banks*
                                    stat) ) )
        (then
            (:= (cdr (nth (stat:source stat) 4) )
                `(,bank) ) )

        (else (if *fg.show-unknown-bank-references?* (then
            (if (! heading-printed?) (then
                (msg 0 t "Vector references to unknown banks:" t)
                (:= heading-printed? t) ) )
            (msg t (h stat) t) ) ) ) ) ) )

    () )
```

```
;==============================================================================
;
; BBLOCK
;
; Every basic block in the flow graph is represented by a BBLOCK.  All
; BBLOCKs are numbered and stored in a global array that maps the
; numbers onto the corresponding BBLOCKs.
;
;===============================
(def-struct bblock                  ;
    number                          ; Number of this block.
    dfo-number                      ; Depth-first-order number of this block.
                                    ;
    first-stat                      ; First STATement in this block.
    last-stat                       ; The last statement.
    (preds        () suppress)      ; List of predecessor blocks.
    (succs        () suppress)      ; List of successor blocks.
                                    ;
    (gen          () suppress)      ; STAT-SET used for flow analysis algorithms.
    (kill         () suppress)      ; STAT-SET used for flow analysis algorithms.
                                    ;
    (reaching-in  () suppress)      ; STAT-SET of definitions reaching this block
    (reaching-out () suppress)      ; on entry and exit from this block.
                                    ;
    (reaching-copies-in             ; STAT-SET of definitions copy-reaching
                  () suppress)      ; this block on entry and exit.
    (reaching-copies-out            ;
                  () suppress)      ;
                                    ;
    (dominators   () suppress)      ; BBLOCK-SET of dominators of this block.
                                    ;
    (live-in      () suppress)      ; NAME-SET of names live on entry and exit
    (live-out     () suppress)      ; from this block.
    )                               ;
;===============================
;
; (BBLOCK:CREATE)
;     Creates a new BBLOCK and records it in the array
;     of BBLOCKs.  This is the only was a BBLOCK should
;     be created.
;
; (BBLOCK:DELETE BBLOCK)
;     Unsplices BBLOCK from the flow graph, forgetting it and all of
;     its STATs.  A BBLOCK may be deleted only if it satisfies the conditions
;     of BBLOCK:UNSPLICE.
;
; (BBLOCK:SPLICE BBLOCK NEW-BBLOCK PREDS)
;     Splices NEW-BBLOCK between BBLOCK and each block in PREDS.
;     PREDS is assumed to be a subset of the preds of BBLOCK.
;
; (BBLOCK:UNSPLICE BBLOCK)
;     Unsplices BBLOCK from the flow graph.  A BBLOCK may be unspliced
;     only if it has 1 successor and 0, 1, or 2 predecessors; or 2
;     successors and 0 or 1 predecessor.
;
; (BBLOCK:APPEND-STAT BBLOCK STAT)
;     Appends STAT to the end of BBLOCK.
;
; (BBLOCK:DELETE-STATS BBLOCK)
;     Deletes all of the STATs in the BBLOCK.
;
; (BBLOCK:MERGE-WITH-SUCCESSOR BBLOCK)
;     Tries to merge BBLOCK with its successor.  The merge can take place
```



```
;     only if BBLOCK has one successor, and that successor has only one
;     predecessor.  If they are mergeable, the successor BBLOCK is deleted
;     and true is returned; otherwise nothing happens and false is
;     returned.
;
; (BBLOCK:DOMINATES? BBLOCK BBLOCK1)
;     Returns true if BBLOCK dominates BBLOCK1.
;
; (BBLOCK:EMPTY? BBLOCK)
;     Returns true if BBLOCK has no STATs in it.
;
; (NUMBER:BBLOCK NUMBER)
;     Returns the BBLOCK of a given number.
;
; ##B <number>
;     Easy, interactive syntax for (NUMBER:BBLOCK <number>).
;
; (LOOP (FOR-EACH-BBLOCK BBLOCK) ...)
;     This is the only public way for enumerating through all the
;     BBLOCKs of the flow graph.  The enumeration is in the original
;     order of the NADDR source.  It is defined via LOOP's
;     DEF-SIMPLE-LOOP-CLAUSE.
;
; (LOOP (FOR-EACH-BBLOCK-STAT BBLOCK STAT) ...)
;     This is the recommended way for enumerating through all the STAT's
;     of a BBLOCK.  The enumeration goes from first stat to last.
;
; (LOOP (FOR-EACH-BBLOCK-STAT^ BBLOCK STAT) ...)
;     Enumerates through the STATs of a BBLOCK in reverse order.
;
;==============================================================================

(include flow-analysis:flow-analysis-decls)

(declare (special
    *fg.total-bblocks*       ;*** Current number of BBLOCKs
    *fg.number:bblock*       ;*** Array for mapping BBLOCK numbers onto
                             ;*** BBLOCKs.
    *fg.entry-bblock*        ;*** Entry block (one with no predecessors)
    ) )


(defun fg.initialize-bblocks ()
    (vector-map:initialize '*fg.number:bblock* '*fg.total-bblocks* 50 t) )

(defun bblock:create ()
    (vector-map:add-element '*fg.number:bblock* '*fg.total-bblocks*
        (bblock:new number *fg.total-bblocks*)
        50
        t) )


(defun bblock:delete ( bblock )
    (assert (bblock:is bblock) )

            ;*** Delete each STAT that is part of the BBLOCK.
            ;
    (bblock:delete-stats bblock)

            ;*** Remove the BBLOCK from the array of BBLOCKs.
            ;
    (:= ([] *fg.number:bblock* (bblock:number bblock) ) () )
```

```
            ;*** Remove BBLOCK from the flow graph.
            ;
    (bblock:unsplice bblock)
    () )


(defun bblock:delete-stats ( bblock )
    (assert (bblock:is bblock) )

    (loop (initial stat      (bblock:first-stat bblock)
                   succ-stat () )
    (while stat)
    (next succ-stat (stat:succ stat) )
    (do
        (stat:delete stat) )
    (next stat succ-stat) )

    (:= (bblock:first-stat bblock) () )
    (:= (bblock:last-stat  bblock) () )

    bblock)


(defun bblock:splice ( bblock new-bblock preds )
    (assert (bblock:is bblock) )
    (assert (bblock:is new-bblock) )

            ;*** Make each predecessor point at NEW-BBLOCK, and remove
            ;*** that predecessor from the preds of BBLOCK.
            ;
    (for (pred-bblock in preds) (do
        (top-level-dsubstq new-bblock bblock (bblock:succs pred-bblock) )
        (:= (bblock:preds bblock)
            (top-level-removeq pred-bblock (bblock:preds bblock) ) ) ) )

            ;*** Add NEW-BBLOCK to the preds of BBLOCK
            ;
    (push (bblock:preds bblock) new-bblock)

            ;*** The preds of NEW-BBLOCK are PREDS, and the succs are
            ;*** are just BBLOCK.
            ;
    (:= (bblock:succs new-bblock) (list bblock) )
    (:= (bblock:preds new-bblock) preds)

    new-bblock)


(defun bblock:unsplice ( bblock )
    (assert (bblock:is bblock) )

    (let ( (pred-bblocks (bblock:preds bblock) )
           (succ-bblocks (bblock:succs bblock) ) )

            ;*** BBLOCK has either 1 successor  and 0, 1, or 2 predecessors;
            ;*** or 2 successors and 0 or 1 predecessors;
            ;*** or 0 successors and 1 predecessor.
            ;
        (assert (|| (&& (== 1 (length succ-bblocks) )
                        (>= 2 (length pred-bblocks) ) )
                    (&& (== 2 (length succ-bblocks) )
                        (>= 1 (length pred-bblocks) ) )
                    (&& (== 0 (length succ-bblocks) )
                        (== 1 (length pred-bblocks) ) ) ) )


            ;*** Make the successors of BBLOCK point back at its predecessors.
            ;
        (for (succ-bblock in succ-bblocks) (do
            (:= (bblock:preds succ-bblock)
                (unionq (top-level-removeq bblock &&&)
                        pred-bblocks) ) ) )

            ;*** Make the predecessors of BBLOCK point at the successors.
            ;
        (caseq (length succ-bblocks)

            (2      ;*** 2 successors and 0 or 1 predecessors.
                    ;
                (if-let ( (pred-bblock (car pred-bblocks) ) ) (then
                    (assert (== 1 (length (bblock:succs pred-bblock) ) ) )
                    (:= (bblock:succs pred-bblock) succ-bblocks) ) ) )

            (1      ;*** 1 successor and 0, 1, or 2 predecessors.
                    ;
                (let ( (succ-bblock (car succ-bblocks) ) )
                    (for (pred-bblock in pred-bblocks) (do
                        (:= (bblock:succs pred-bblock)
                            (noduplesq (top-level-substq succ-bblock bblock
                                                         &&&) ) ) ) ) ) ) )

            (0      ;*** 0 successors, 1 predeccessor.
                    ;
                (:= (bblock:succs (car pred-bblocks) ) () ) ) )


            ;*** Clear all pointers in BBLOCK.
            ;
        (:= (bblock:preds bblock) () )
        (:= (bblock:succs bblock) () )

        () ) )


(defun bblock:append-stat ( bblock stat )
    (assert (bblock:is bblock) )
    (assert (stat:is   stat) )

    (:= (stat:bblock stat) bblock)
    (if (! (bblock:first-stat bblock) ) (then
        (:= (bblock:first-stat bblock) stat)
        (:= (bblock:last-stat  bblock) stat) )
    (else
        (:= (stat:pred stat) (bblock:last-stat bblock) )
        (:= (stat:succ (bblock:last-stat bblock) )
            stat)
        (:= (bblock:last-stat bblock) stat) ) )
    stat)


(defun bblock:merge-with-successor ( bblock )
    (if (&& (== 1 (length (bblock:succs bblock) ) )
            (== 1 (length (bblock:preds (car (bblock:succs bblock) ) ) ) ) )
```

```
    (then
        (let*( (succ-bblock (car (bblock:succs bblock) ) )
               (succ-bblock-stats
                    (loop (for-each-bblock-stat succ-bblock stat) (save
                        stat) ) ) )

            (loop (for stat in succ-bblock-stats) (do
                (stat:extract stat)
                (bblock:append-stat bblock stat) ) )

            (bblock:delete succ-bblock) )
            t)

    (else
        () ) ) )


(defun bblock:dominates? ( bblock bblock1 )
    (assert (bblock:is bblock) )
    (assert (bblock:is bblock1) )
    (bblock-set:member? (bblock:dominators bblock1) bblock) )


(defun bblock:empty? ( bblock )
    (assert (bblock:is bblock) )
    (&& (! (bblock:first-stat bblock) )
        (! (bblock:last-stat  bblock) ) ) )


(defmacro number:bblock ( number )
    '([] *fg.number:bblock* ,number) )

(def-sharp-sharp b
    '([] *fg.number:bblock* ,(read) ) )


(def-simple-loop-clause for-each-bblock ( clause )
    (let ( ( (for-each-bblock var) clause)
           (index (intern (gensym) ) ) )

        (if (! (&& (= 2 (length clause) )
                   (litatom var) ) )
            (error (list clause "Invalid FOR-EACH-BBLOCK syntax.") ) )

        '( (initial ,var () )
           (incr ,index from 0 to (+ -1 *fg.total-bblocks*) )
           (next ,var ([] *fg.number:bblock* ,index) )
           (when ,var) ) ) )


(eval-when (eval compile load)
(def-simple-loop-clause for-each-bblock-stat ( clause )
    (let ( ( (for-each-bblock-stat bblock stat-var) clause)
           (bblock-var (intern (gensym) ) ) )

        (if (! (&& (= 3 (length clause) )
                   (litatom stat-var) ) )
            (error (list clause "Invalid FOR-EACH-BBLOCK-STAT syntax.") ) )

        '( (initial ,bblock-var ,bblock
                    ,stat-var   () )
           (next ,stat-var
                 (if (! ,stat-var)
```



```
                     (bblock:first-stat ,bblock-var)
                     (stat:succ ,stat-var) ) )
           (while ,stat-var) ) ) )
)

(def-simple-loop-clause for-each-bblock-stat^ ( clause )
    (let ( ( (for-each-bblock-stat bblock stat-var) clause)
           (bblock-var (intern (gensym) ) ) )

        (if (! (&& (= 3 (length clause) )
                   (litatom stat-var) ) )
            (error (list clause "Invalid FOR-EACH-BBLOCK-STAT^ syntax.") ) )

        '( (initial ,bblock-var ,bblock
                    ,stat-var   () )
           (next ,stat-var
                 (if (! ,stat-var)
                     (bblock:last-stat ,bblock-var)
                     (stat:pred ,stat-var) ) )
           (while ,stat-var) ) ) )


(defun bblock:print ( bblock &optional bblock-fields )
    (assert (bblock:is bblock) )

    (msg 0 (bblock:number bblock) ": succs: "
        (for (succ-bblock in (bblock:succs bblock) ) (save
            (bblock:number succ-bblock) ) )
        " preds: "
        (for (pred-bblock in (bblock:preds bblock) ) (save
            (bblock:number pred-bblock) ) )
        t)

    (loop (for field in bblock-fields) (do
        (caseq field
            (gen
                (msg "    Gen: "
                     (e (stat-set:print (bblock:gen bblock) ) )
                     t) )
            (kill
                (msg "    Kill: "
                     (e (stat-set:print (bblock:kill bblock) ) )
                     t) )
            (reaching-in
                (msg "    Reaching-in: "
                     (e (stat-set:print (bblock:reaching-in bblock) ) )
                     t) )
            (reaching-out
                (msg "    Reaching-out: "
                     (e (stat-set:print (bblock:reaching-out bblock) ) )
                     t) )
            (reaching-copies-in
                (msg "    Reaching-copies-in: "
                     (e (stat-set:print (bblock:reaching-copies-in bblock) ) )
                     t) )
            (reaching-copies-out
                (msg "    Reaching-copies-out: "
                     (e (stat-set:print (bblock:reaching-copies-out bblock) ) )
                     t) )
            (dominators
                (msg "    Dominators: "
                     (e (bblock-set:print (bblock:dominators bblock) ) )
                     t) )
```

```
        (live-in
            (msg "    Live-in: "
                 (e (name-set:print (bblock:live-in bblock) ) )
                 t) )
        (live-out
            (msg "    Live-out: "
                 (e (name-set:print (bblock:live-out bblock) ) )
                 t) )
        ) ) )

(loop (for-each-bblock-stat bblock stat) (do
    (msg "    " (j (stat:number stat) 3) ": " (stat:source stat) )
    (if (memq 'reaching-uses bblock-fields) (then
        (msg (t 45) " "
             (for (use-stat in (stat:reaching-uses stat) ) (save
                (stat:number use-stat) ) ) ) ) )
    (msg t) ) ) )
```

```
;===========================================================================
; BBLOCK-SETS
;
; Sets of BBLOCKs are represented using BBLOCK-SETs, currently implemented
; as BIT-SETs.
;
; *FG.EMPTY-BBLOCK-SET*
;     The empty BBLOCK-SET.
;
; (BBLOCK-SET:UNIVERSE)
;     Set of all BBLOCKs.
;
; (BBLOCK-SET:SINGLETON BBLOCK)
;     Creates a new set containing BBLOCK.
;
; (BBLOCK-SET:MEMBER? SET BBLOCK)
;     Returns true if BBLOCK is a member of SET.
;
; (BBLOCK-SET:INTERSECTION SET1 SET2 ...)
;     Returns a new set that is the intersection of all the given sets.
;
; (BBLOCK-SET:UNION SET1 SET2 ...)
;     Returns a new set that is the union of all the given sets.
;
; (BBLOCK-SET:UNION1 SET BBLOCK)
;     Unions a single BBLOCK into SET.
;
; (BBLOCK-SET:DIFFERENCE SET1 SET2)
;     Returns a new set that contains all elements in SET1 not in SET2.
;
; (BBLOCK-SET:= SET1 SET2)
;     Returns true if the two sets are equal.
;
; (BBLOCK-SET:SIZE SET)
;     Returns the number of elements in the set.
;
; (LOOP (FOR-EACH-BBLOCK-SET-ELEMENT SET BBLOCK)
;     Enumerates BBLOCK through each element in SET.  Uses
;     DEF-SIMPLE-LOOP-CLAUSE.
;
; (BBLOCK-SET:PRINT SET)
;    Prints SET by printing out the statement numbers.
;
;===========================================================================

(include flow-analysis:flow-analysis-decls)

(declare (special *fg.total-bblocks*) )

(defvar *fg.empty-bblock-set* () )    ;*** the empty BBLOCK-SET.

(defun bblock-set:universe ()
    (bit-set:universe *fg.total-bblocks*) )

(defun bblock-set:singleton ( bblock )
    (bit-set:singleton (bblock:number bblock) ) )

(defun bblock-set:member? ( set bblock )
    (bit-set:member? set (bblock:number bblock) ) )

(defun bblock-set:intersection args
    (apply 'bit-set:intersection (listify-lexpr-args args) ) )
```



```
(defun bblock-set:union args
    (apply 'bit-set:union (listify-lexpr-args args) ) )

(defun bblock-set:union1 ( set bblock )
    (bit-set:union1 set (bblock:number bblock) ) )

(defun bblock-set:difference ( set1 set2 )
    (bit-set:difference set1 set2) )

(defun bblock-set:= ( set1 set2 )
    (bit-set:= set1 set2) )

(defun bblock-set:size ( set )
    (bit-set:size set) )

(def-simple-loop-clause for-each-bblock-set-element ( clause )
    (let ( ( (for-each-bblock-set-element set bblock) clause)
           (index (intern (gensym) ) ) )
        (if (! (&& (= 3 (length clause) )
                   (litatom bblock) ) )
            (error (list clause
                         "Invalid FOR-EACH-BBLOCK-SET-ELEMENT syntax.") ) )
        `( (initial ,bblock () )
           (for-each-bit-set-element ,set ,index)
           (next ,bblock (number:bblock ,index) )
           (when ,bblock) ) ) )

(defun bblock-set:print ( set )
    (bit-set:print set) )
```

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

(:= *flow.build-module-list* '(
    flow-analysis:stat
    flow-analysis:stat-set
    flow-analysis:bblock
    flow-analysis:bblock-set
    flow-analysis:name
    flow-analysis:name-set
    flow-analysis:loop

    flow-analysis:naddr-to-flow-graph
    flow-analysis:remove-dead-code
    flow-analysis:flow-graph-to-naddr

    flow-analysis:temporary-name
    flow-analysis:collect-names
    flow-analysis:depth-first-order
    flow-analysis:reaching-defs
    flow-analysis:reaching-uses
    flow-analysis:reaching-copies
    flow-analysis:live-names
    flow-analysis:dominators
    flow-analysis:find-loops

    flow-analysis:loop-invariant-motion
    flow-analysis:induction-variable-removal
    flow-analysis:cse-node
    flow-analysis:cse-hash-tables
    flow-analysis:common-subexpression-elimination
    flow-analysis:copy-propagation
    flow-analysis:constant-folding
    flow-analysis:variable-renaming

    flow-analysis:derivations
    flow-analysis:disambiguator
    flow-analysis:disambiguator-tool
    flow-analysis:bank-disambiguator

    flow-analysis:statistics
    flow-analysis:dependencies
    flow-analysis:flow-analysis-options
    flow-analysis:flow-analysis
    ) )

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

```
;***=========================================================================
;***
;*** (FG.COLLECT-NAMES)
;***    Collects all the scalar names defined in the flow graph, doing
;***    NAME:CREATE on each one.  Sets :DEFINING-STATS for each name to
;***    be the set of STATs defining that name, and :USING-STATS to be
;***    the set of STATs using a name.
;***
;***=========================================================================

(include flow-analysis:flow-analysis-decls)


(defun fg.collect-names ()
    (fg.initialize-names)

    (loop (for-each-stat stat) (do
        (caseq (stat:operator stat)
            (def-block
                (for (name in (stat:part stat 'in-variables) ) (do
                    (if (litatom name) (then
                        (name:create name 'scalar 0)
                        (name:add-defining-stat name stat) ) ) ) ) ) )
            (dcl
                (push *fg.all-vector-names* (stat:part stat 'variable) )
                (name:create (stat:part stat 'variable)
                             'vector
                             (stat:part stat 'length) ) )

            (t
                (let ( (defined-name (stat:part stat 'written) ) )
                    (if defined-name (then
                        (name:create defined-name 'scalar 0)
                        (name:add-defining-stat defined-name stat) ) ) )))))

    (loop (for-each-stat stat) (do
        (loop (for-each-stat-operand-read stat read-name) (do
            (name:add-using-stat read-name stat) ) ) ) )

    () )
```

```
;***----------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for   ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                              ********
;***----------------------------------------------------------------********
;==========================================================================
;
; COMMON SUBEXPRESSION ELIMINATION
;
; This module implements basic block common subexpression elimination.
; Currently, a VSTORE into a vector kills all previous values loaded from
; that vector.
;
; (FA.INITIALIZE-CSE)
;     Initializes this module for the current flow graph.
;
; (FA.ELIMINATE-COMMON-SUBEXPRESSIONS)
;     Eliminates the common subexpressions in each BBLOCK of the flow
;     graph, destroying the old STATs of the BBLOCKs and replacing them
;     with new STATs.  This invalidates all the flow information except
;     the live variable info.
;
; (CSE.TEST OPER-LIST LIVE-LIST)
;     A testing function that takes OPER-LIST, a list of NADDR, converts
;     it into a BBLOCK, eliminates the CSEs in the BBLOCK, and then prints
;     the new BBLOCK.  LIVE-LIST is a list of names that are live on exit
;     from the block.
;
; (CSE.PRINT)
;     Prints out the current DAG of CSE-NODEs.
;
;==========================================================================


(eval-when (compile load)
    (include flow-analysis:flow-analysis-decls) )

(eval-when (compile)
    (build '(flow-analysis:cse-node) ) )

(declare (special
    *cse.all-cse-nodes*
    ) )

(defun fa.initialize-cse ()
    (cse.initialize)
    () )


(defun fa.eliminate-common-subexpressions ()
    (fa.initialize-cse)
    (loop (for-each-bblock bblock) (do
        (bblock:eliminate-common-subexpressions bblock) ) )
    () )

(defun cse.test ( oper-list live-list )
    (fa.initialize)

    (let ( (bblock (bblock:create) ) )
        (loop (for name in live-list) (do
            (name:create name 'scalar 0)
            (:= (bblock:live-out bblock)
                (name-set:union1 (bblock:live-out bblock)
```

```
                                   name) ) ) )
        (loop (for oper in oper-list) (do
            (loop (for name in (oper:part oper 'read) ) (do
                (name:create name 'scalar 0) ) )
            (if (oper:part oper 'written)
                (name:create (oper:part oper 'written) 'scalar 0) )
            (bblock:append-stat bblock (stat:create oper) ) ) )

        (bblock:eliminate-common-subexpressions bblock)

        (bblock:print bblock)
        () ) )


;***=======================================================================
;***
;*** (BBLOCK:ELIMINATE-COMMON-SUBEXPRESSIONS BBLOCK)
;***
;*** Eliminates the common subexpressions of a BBLOCK.  All the old STATs
;*** in the BBLOCK are deleted and replaced by new ones.  If BBLOCK
;*** is now empty (because of dead code that was removed), delete it.
;***
;***=======================================================================

(defun bblock:eliminate-common-subexpressions ( bblock )
    (cse.initialize)

    (loop (for-each-bblock-stat bblock stat) (do
        (cse.process-oper (stat:source stat) ) ) )

    (cse.remove-useless-labels (bblock:live-out bblock) )
    (cse.make-copy-nodes-for-initial-nodes)
    (cse.assign-names-and-add-pseudo-edges)
    (cse.set-remaining-counts)

    (bblock:delete-stats bblock)
    (loop (for oper in (cse.generate-code) ) (do
        (bblock:append-stat bblock (stat:create oper) ) ) )

    (if (bblock:empty? bblock) (then
        (bblock:delete bblock) ) )

    () )

;***=======================================================================
;***
;*** (CSE.INITIALIZE)
;***
;*** Initializes for the building of the next basic block DAG.
;***
;***=======================================================================

(defun cse.initialize ()
    (cse.initialize-cse-nodes)
    (cse.initialize-hash-tables)
    () )


;***=======================================================================
;***
;*** (CSE.PROCESS-OPER OPER)
;***
```

```
;*** Adds a CSE-NODE representing operation OPER to the DAG being built.
;***
;***==========================================================================

(defun cse.process-oper ( oper )
    (caseq (oper:group oper)
        (vload
            (cse.process-vload oper) )

        (vstore
            (cse.process-vstore oper) )

        ( (one-in-one-out two-in-one-out three-in-one-out if-compare
                          if-boolean assert loop-assign vbase)
            (caseq (oper:operator oper)
                ( (fassign iassign)
                    (cse.process-assign oper) )
                (t
                    (cse.process-n-in-n-out oper) ) ) )

        ( t
          (cse.process-miscellaneous oper) ) )
    () )


;***==========================================================================
;***
;*** (CSE.PROCESS-N-IN-N-OUT OPER)
;***
;*** Adds a CSE-NODE representing operation OPER to the DAG.  OPER should
;*** be an operation that reads one or two variables and optionally writes
;*** a result.
;***
;*** First, the CSE-NODEs representing the operands of OPER are
;*** found/created.  Then, the hash tables are examined to see if a node
;*** computing OPER already exists.  If so, then the variable defined
;*** by OPER is just added to the list of names on the found CSE-NODE.
;*** If there isn't any previous CSE-NODE, then one is created.
;***
;***==========================================================================

(defun cse.process-n-in-n-out ( oper )
    (let*( (children
                (if (== 'vbase (oper:operator oper) ) (then
                    `(,(cse.name:cse-node (oper:part oper 'vector) ) ) )
                (else
                    (for (name in (oper:part oper 'read) ) (save
                        (cse.name:cse-node name) ) ) ) ) )
           (expr-key
                `(,(oper:operator oper)
                  ,,children) )
           (cse-node
                (cse.expr:cse-node expr-key) ) )

        (if (! cse-node) (then
            (:= cse-node
                (cse-node:create-interior oper children) )
            (cse.expr:define-cse-node expr-key cse-node) ) )

        (if (oper:part oper 'written) (then
            (cse-node:assign-to-name cse-node (oper:part oper 'written) ) ) )
        () ) )
```



```
;***==========================================================================
;***
;*** (CSE.PROCESS-MISCELLANEOUS OPER)
;***
;*** Adds a CSE-NODE representing operation OPER to the DAG.  OPER is any
;*** any miscellaneous NADDR operation that doesn't compute a value.  The
;*** CSE-NODE created for OPER is an "interior" node but it has no children.
;***
;***==========================================================================

(defun cse.process-miscellaneous ( oper )
    (let ( (cse-node (cse-node:create-interior oper () ) ) )
        (:= (cse-node:miscellaneous? cse-node) t)
        () ) )


;***==========================================================================
;***
;*** (CSE.PROCESS-ASSIGN OPER)
;***
;*** Adds a CSE-NODE representing operation OPER to the DAG.  OPER should be
;*** be an xASSIGN.
;***
;*** The CSE-NODE containing the value being assigned is found, and that
;*** node is assigned to the name being assigned by OPER (the name is added
;*** to the list of labels of the CSE-NODE).
;***
;***==========================================================================

(defun cse.process-assign ( oper )
    (let ( (cse-node (cse.name:cse-node (oper:part oper 'read1) ) ) )
        (cse-node:assign-to-name cse-node (oper:part oper 'written) )
        (:= (cse-node:datatype cse-node) (oper:dest-datatype oper) )
        () ) )


;***==========================================================================
;***
;*** (CSE.PROCESS-VLOAD OPER)
;***
;*** Adds a CSE-NODE representing operation OPER to the DAG.  OPER should be
;*** a VLOAD.
;***
;*** If an unkilled CSE-NODE already exists that VLOADs from the same
;*** vector and index, that CSE-NODE is used for OPER -- the destination
;*** of OPER is just added to the labels of the CSE-NODE.
;***
;*** If there is no such previous VLOAD, but there is an unkilled VSTORE
;*** into the same vector and index, that CSE-NODE node is used.  The
;*** destination of OPER is added to the labels of the CSE-NODE that
;*** represents the value being VSTOREd, not to the VSTORE CSE-NODE itself.
;***
;*** If there is no previous VLOAD or VSTORE cse, then a new CSE-NODE is
;*** created.  A pseudo-edge is added between the new CSE-NODE and any
;*** previous unkilled VSTORE into the same vector.  This ensures proper
;*** evaluation order.  The new CSE-NODE is added to the list of VSTOREs
;*** and VLOADs of the initial node representing the vector.
;***
;***==========================================================================

(defun cse.process-vload ( oper )
    (let*( (vector-cse-node (cse.name:cse-node (oper:part oper 'vector) ) )
```

```
        (index-cse-node  (cse.name:cse-node (oper:part oper 'index ) ) )
        (children        '(,vector-cse-node ,index-cse-node) )
        (expr-key        '(vload ,,children) )
        (cse-node        (cse.expr:cse-node expr-key) ) )

     (? ( (|| (! cse-node)
              (cse-node:killed? cse-node) )
          (:= cse-node (cse-node:create-interior oper children) )
          (cse.expr:define-cse-node expr-key cse-node)
          (cse-node:assign-to-name cse-node (oper:part oper 'written) )

          (loop (for vload-vstore-cse-node in
                     (cse-node:vloads-vstores vector-cse-node) )
          (when (! (cse-node:killed? vload-vstore-cse-node) ) )
          (when (== 'vstore (oper:group
                               (cse-node:source vload-vstore-cse-node))))
          (do
             (cse-node:add-pseudo-child cse-node vload-vstore-cse-node) ) )

          (push (cse-node:vloads-vstores vector-cse-node) cse-node) )


        ( (== 'vstore (oper:group (cse-node:source cse-node) ) )
          (cse-node:assign-to-name (caddr (cse-node:children cse-node) )
                                   (oper:part oper 'written) ) )

        ( t
          (cse-node:assign-to-name cse-node (oper:part oper 'written) ) ) )

     () ) )


;***======================================================================
;***
;*** (CSE.PROCESS-VSTORE OPER)
;***
;*** Adds a CSE-NODE representing operation OPER to the DAG.  OPER should be
;*** a VSTORE.
;***
;*** A new CSE-NODE is always created for a VSTORE.  All previous VLOADs
;*** and VSTOREs into the same vector are killed.   The new CSE-NODE is
;*** added to the list of VLOADs and VSTOREs stored in the initial node
;*** for the vector.  Note the trickiness of hashing the new CSE-NODE
;*** under VLOAD instead of VSTORE -- this lets future VLOADs find this
;*** VSTORE as a cse.
;***
;*** We can make VSTOREs smarter by having them kill only those previous
;*** VLOADs and VSTOREs that could possibly have the same index as this
;*** VSTORE.  We would call the disambiguator oracle for this info.
;***
;***======================================================================

(defun cse.process-vstore ( oper )
    (let*( (vector-cse-node (cse.name:cse-node (oper:part oper 'vector) ) )
           (index-cse-node  (cse.name:cse-node (oper:part oper 'index ) ) )
           (value-cse-node  (cse.name:cse-node (oper:part oper 'read2) ) )
           (children        '(,vector-cse-node ,index-cse-node
                                               ,value-cse-node) )
           (expr-key        '(vload ,vector-cse-node ,index-cse-node) )
           (cse-node        (cse-node:create-interior oper children) ) )

       (loop (for vload-vstore-cse-node in
                  (cse-node:vloads-vstores vector-cse-node) )
```



```
        (when (! (cse-node:killed? vload-vstore-cse-node) ) )
        (do
           (cse-node:add-pseudo-child cse-node vload-vstore-cse-node)
           (:= (cse-node:killed? vload-vstore-cse-node) t) ) )

        (push (cse-node:vloads-vstores vector-cse-node) cse-node)
        (cse.expr:define-cse-node expr-key cse-node)
        () ) )


;***======================================================================
;***
;*** (CSE.REMOVE-USELESS-LABELS)
;***
;*** For each CSE-NODE, removes all labels (names) that are not live on
;*** exit from the BBLOCK.   This guarrantees we will not do any useless
;*** assignments.
;***
;***======================================================================

(defun cse.remove-useless-labels ( live-out )
    (loop (for cse-node in *cse.all-cse-nodes*) (do
        (loop (for name in (cse-node:labels cse-node) )
              (initial new-labels () )
        (do
            (if (&& (name-set:member? live-out name)
                    (! (memq name new-labels) ) )
            (then
                (push new-labels name) ) ) )
        (result
            (:= (cse-node:labels cse-node) new-labels) ) ) ) )
    () )


;***======================================================================
;***
;*** (CSE.MAKE-COPY-NODES-FOR-INITIAL-NODES)
;***
;*** ?
;***
;***======================================================================

(defun cse.make-copy-nodes-for-initial-nodes ()
    (loop (for cse-node in *cse.all-cse-nodes*) (do
        (if (cse-node:initial? cse-node)
            (cse-node:make-copy-cse-nodes cse-node) ) ) ) )


;***======================================================================
;***
;*** (CSE.ASSIGN-NAMES-AND-ADD-PSEUDO-EDGES)
;***
;*** For each CSE-NODE that produces a value:
;***
;*** 1.  One of the names in :LABELS is assigned as the :NAME of this
;***     node;  this name will be used to hold the value of the node.
;***     If there is no suitable name, a temporary is created.
;***
;*** 2.  A pseudo-edge is added between this node and all uses of the
;***     initial node of the name picked in step 1.  This guarrantees
;***     that the name won't be assigned into until all uses of its
;***     previous value have been evaluated.
```

```
;***
;*** 3.  ASSIGN nodes are created for all other names that label the node;
;***     each ASSIGN assigns its name with the value of this node.  Pseudo
;***     edges are added between each new ASSIGN node and the uses of
;***     the initial node of its name (as in step 2).
;***
;***=========================================================================


(defun cse.assign-names-and-add-pseudo-edges ()
    (loop (initial rest-cse-nodes *cse.all-cse-nodes*
                   name           () )
          (while rest-cse-nodes)
          (bind cse-node (pop rest-cse-nodes)
                oper     (cse-node:source cse-node) )
          (when (|| (cse-node:initial? cse-node)
                    (&& (! (cse-node:miscellaneous? cse-node) )
                        (!== 'vstore (oper:group oper) )
                        (oper:part oper 'written) ) ) )
    (do
        (:= name (cse-node:pick-a-good-name cse-node) )
        (:= (cse-node:name cse-node) name)
        (:= (cse-node:labels cse-node)
            (top-level-removeq name (cse-node:labels cse-node) ) )

        (cse-node:add-initial-name-edges cse-node)
        (loop (for copy-cse-node in (cse-node:make-copy-cse-nodes cse-node) )
        (do
            (push rest-cse-nodes copy-cse-node)
            (cse-node:add-initial-name-edges copy-cse-node) ) ) ) )
    () )


;***=========================================================================
;***
;*** (CSE-NODE:PICK-A-GOOD-NAME CSE-NODE)
;***
;*** Picks one of the names labelling CSE-NODE (:LABELS) as the name that
;*** will be used to hold the value of CSE-NODE.  For initial nodes, the
;*** initial name is returned.
;***
;*** For interior nodes, one of the names of :LABELS is used if there
;*** is an eligible name.  A name is eligible if every use of the initial
;*** value node of that name is not a ancestor of this CSE-NODE (if one
;*** of the uses of the initial value of the name were an ancestor,
;*** computing CSE-NODE into the name would destroy its initial value
;*** before all uses of it were evaluated).  If there is no eligible name,
;*** a new temporary is created.
;***
;*** If CSE-NODE does not compute a result (e.g. it is a cond jump), then
;*** () is returned.
;***
;***=========================================================================


(defun cse-node:pick-a-good-name ( cse-node )
    (assert (cse-node:is cse-node) )

    (? ( (cse-node:initial? cse-node)
         (cse-node:initial-name cse-node) )

       ( (oper:part (cse-node:source cse-node) 'written)
         (loop (for name in (cse-node:labels cse-node) )
```


```
                (initial all-initial-uses-ok? () )
          (do
             (:= all-initial-uses-ok? t)

             (loop (for initial-use-cse-node in (cse.name:initial-uses name) )
             (do
                 (:= all-initial-uses-ok?
                     (&& all-initial-uses-ok?
                         (! (cse-node:ancestor? initial-use-cse-node
                                                cse-node) ) ) ) ) )

             (if all-initial-uses-ok?
                 (return name) ) )

          (result
             (if (cse-node:source cse-node)
                 (fa.temporary-name (oper:part (cse-node:source cse-node)
                                               'written) )
                 (fa.temporary-name () ) ) ) ) )
       ( t
         () ) ) )


;***=========================================================================
;***
;*** (CSE-NODE:ADD-INITIAL-NAME-EDGES CSE-NODE)
;***
;*** Adds a pseudo-edge between CSE-NODE and all nodes that use the initial
;*** value of the :NAME of CSE-NODE.  This guarantees that CSE-NODE will
;*** be evaluated (and will destroy the initial value of :NAME) only after
;*** all uses of :NAME have been evaluated.
;***
;***=========================================================================

(defun cse-node:add-initial-name-edges ( cse-node )
    (if (! (cse-node:initial? cse-node) )
        (loop (for initial-use-cse-node in
                   (cse.name:initial-uses (cse-node:name cse-node) ) )
        (when (!== initial-use-cse-node cse-node) )
        (do
            (cse-node:add-pseudo-child cse-node initial-use-cse-node) ) ) ) )



;***=========================================================================
;***
;*** (CSE.SET-REMAINING-COUNTS)
;***
;*** Sets the :REMAINING-COUNT of each CSE-NODE to be the sum of the number
;*** of children and pseudo-children.  This count indicates the number of
;*** children of a node remaining to be evaluated.
;***
;***=========================================================================

(defun cse.set-remaining-counts ()
    (loop (for cse-node in *cse.all-cse-nodes*) (do
        (:= (cse-node:remaining-count cse-node)
            (+ (length (cse-node:children        cse-node) )
               (length (cse-node:pseudo-children cse-node) ) ) ) ) ) )


;***=========================================================================
;***
```

```
;*** (CSE.GENERATE-CODE)
;***
;*** Traverses the DAG in topological order (from the leaves up),
;*** converting DAG nodes back into NADDR operations.  A list of NADDR
;*** is returned.
;***
;*** There is one slightly tricky thing going on here --
;*** *CSE.ALL-CSE-NODES* is in reverse order of creation.  When we
;*** enumerate through looking for all leaf nodes, a list of those leaf
;*** nodes is created (reversed again) containing those nodes in original
;*** source order.  Those nodes are processed in depth-first manner.  This
;*** all guarrantees that miscellaneous pseudo-ops at the beginning or
;*** end of block maintain their relative position.  Neat how that falls
;*** out.
;***
;*** But it is slightly more tricky than that.  Induction variable removal
;*** works best when the induction statments (I := I +/-/* C) are evaluated
;*** as late as possible with respect to statements that use the older
;*** value of I.  So when we are picking the next node off of the to-do
;*** stack, if the first node on the stack is a possible induction
;*** statement, we ignore it and look deeper into the stack for a non-
;*** induction node.
;***
;***=========================================================================

(:= *cse.beginning-pseudo-operators*
    '(def-block dcl) )

(defun cse.generate-code ()

(let ( (to-do-list () )
       (code       () )
       (cond-code  () ) )

        ;*** Build the TO-DO-LIST of nodes with no children.   Put all
        ;*** DEF-BLOCKs and DCLs at the very beginning of the list.
        ;***
    (loop (for cse-node in *cse.all-cse-nodes*)
          (when (== 0 (cse-node:remaining-count cse-node) ) )
          (when (! (memq (oper:operator (cse-node:source cse-node) )
                         *cse.beginning-pseudo-operators*) ) )
    (do
        (push to-do-list cse-node) ) )

    (loop (for cse-node in *cse.all-cse-nodes*)
          (when (== 0 (cse-node:remaining-count cse-node) ) )
          (when (memq (oper:operator (cse-node:source cse-node) )
                      *cse.beginning-pseudo-operators*) )
    (do
        (push to-do-list cse-node) ) )


    (loop (while to-do-list)
          (initial cse-node () )
    (do
            ;*** Pick the next node.  If the first node in the remaining list
            ;*** is not a possible induction statement, use it.  Otherwise
            ;*** see if there is a non-induction and non-pseudo-op node
            ;*** later on in the list;  if not, then just use the first
            ;*** element of the list.
            ;***
        (if (! (cse-node:possible-induction? (car to-do-list) ) ) (then
            (:= cse-node (pop to-do-list) ) )
```



```
        (else
            (:= cse-node
                (loop (for cse-node in to-do-list)
                      (unless (&& (cse-node:source cse-node)
                                  (|| (oper:property?
                                          (cse-node:source cse-node) 'pseudo-op)
                                      (cse-node:possible-induction? cse-node)
                                      ) ) )
                (do
                    (return cse-node) )
                (result (car to-do-list) ) ) )
            (:= to-do-list (top-level-removeq cse-node &&&) ) ) )

            ;*** Now generate NADDR for the node.
            ;***
        (if (! (cse-node:initial? cse-node) ) (then
            (caseq (oper:group (cse-node:source cse-node) )
                ( (if-true if-compare goto stop)
                    (push cond-code (cse-node:generate cse-node) ) )
                ( t
                    (push code      (cse-node:generate cse-node) ) ) ) ) )

        (loop (for parent-cse-node in (cse-node:parents cse-node) ) (do
            (if (= 0 (-- (cse-node:remaining-count parent-cse-node) ) )
                (push to-do-list parent-cse-node) ) ) ) )
    (result
        (if cond-code (then
            (assert (= 1 (length cond-code) ) )
            (push code (car cond-code) ) ) )
        (dreverse code) ) ) ) )


;***=========================================================================
;***
;*** (CSE-NODE:POSSIBLE-INDUCTION? CSE-NODE)
;***
;*** Returns true if CSE-NODE describes an operation of the form:
;***
;***    I := I +/-/* C   or   I := C +/-/* I
;***
;*** for any C.
;***
;***=========================================================================

(defun cse-node:possible-induction? ( cse-node )
    (let*( (name     (cse-node:name     cse-node) )
           (children (cse-node:children cse-node) )
           (oper     (cse-node:source   cse-node) ) )

        (&& (memq (oper:operator oper) '(iadd isub imul) )
            (|| (== name (cse-node:name (car  children) ) )
                (== name (cse-node:name (cadr children) ) ) ) ) ) ) )

;***=========================================================================
;***
;*** (CSE-NODE:GENERATE CSE-NODE)
;***
;*** Constructs a NADDR operation for CSE-NODE, substituting in the name
;*** of this node and its children for the old variable names.
;***
;***=========================================================================

(defun cse-node:generate ( cse-node )
```

```
(let ( (oper (cse-node:source cse-node) ) )

    (?( (== 'vbase (oper:operator oper) )
        `(vbase ,(cse-node:name cse-node) ,(oper:part oper 'vector) ) )

      ( (cse-node:children cse-node)
        (:= oper
            (oper:substitute-for-part
                oper
                (loop (for child-cse-node in (cse-node:children cse-node) )
                (save
                    (cse-node:name child-cse-node) ) )
                'read) )

        (if (cse-node:name cse-node) (then
            (:= oper (oper:substitute-for-part oper
                                            (cse-node:name cse-node)
                                            'written) ) ) )
        oper)

      ( t
        oper) ) ) )


;***=========================================================================
;***
;*** (CSE.PRINT)
;***
;*** Prints out the current DAG.
;***
;***=========================================================================

(defun cse.print ()

(loop (for cse-node in *cse.all-cse-nodes*) (do
    (msg 0 (cse-node:number cse-node) ":" t)

    (if (cse-node:name cse-node)
        (msg "  name:    " (cse-node:name cse-node) t) )
    (if (&& (! (cse-node:name cse-node) )
            (cse-node:initial-name cse-node) )
        (msg "  initial-name: " (cse-node:initial-name cse-node) t) )
    (if (cse-node:labels cse-node)
        (msg "  labels:  " (cse-node:labels cse-node) t) )
    (if (cse-node:source cse-node)
        (msg "  source:           " (cse-node:source cse-node) t) )
    (if (cse-node:parents cse-node)
        (msg "  parents: "
            (cse-node-list:numbers (cse-node:parents cse-node) ) t) )
    (if (cse-node:children cse-node)
        (msg "  children:           "
            (cse-node-list:numbers (cse-node:children cse-node) ) t) )
    (if (cse-node:pseudo-children cse-node)
        (msg "  p-children:         "
            (cse-node-list:numbers (cse-node:pseudo-children cse-node) ) t))
    (if (cse-node:remaining-count cse-node)
        (msg "  remaining-count: " (cse-node:remaining-count cse-node) t) )
    (if (! (bit-set:= *bit-set.empty-set* (cse-node:descendants cse-node)))
    (then
        (msg "  descendants:      "
            (e (bit-set:print (cse-node:descendants cse-node) ) )
            t) ) ) ) )
```



```
() )



(defun cse-node-list:numbers ( list )
    (for (cse-node in list) (save
        (cse-node:number cse-node) ) ) )
```

```
;===========================================================================
;
; (FG.FOLD-CONSTANTS)
;
; Does constant folding, using an iterative algorithm similar to that
; in in the Dragon Book.  Given a constant assignment S: A := C (where
; C is a constant number), we substitute C for A in all uses of S that
; have S as the only reaching definition of A.  If the use now has all
; constant operands, we evaluate the right hand side and replace it by
; an assignment statement.  We keep iterating over the flow graph until
; we can't do any more substitutions.
;
; Constant folding also includes the following transformations (for both
; integer and real arithmetic):
;
;    A {* / AND} 1 ==> A
;    A {+ - OR}  0 ==> A
;    0 {* / AND} A ==> 0
;    A /         A ==> 1
;    A -         A ==> 0
;
; Eventually, we'll want to do constant folding on conditional jumps also.
;
; Unintuitively, it is faster to keep iterating over the flow graph in
; in depth-first order, rather than using a statment-propagation algorithm.
; The reason is that enumerating in depth-first order causes
; STAT:REACHING-DEFS to be evaluated very efficienty (since it is
; calculated on the fly from the basic block reaching defs).  Whereas
; a statement- propagation algorithm, keeping a stack of statements that
; have changed, evaluates STAT:REACHING-DEFS in random order, causing
; essentially N**2 behaviour with lots of consing.
;
;===========================================================================

(eval-when (compile load)
    (include flow-analysis:flow-analysis-decls) )


(defun fg.fold-constants ()

    (loop (initial change? t)
          (while change?)
          (incr pass from 1)
    (do
        (:= change? () )

        (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward) )
        (do
            (loop (for-each-bblock-stat bblock stat)
                  (when (!== 'live (stat:operator stat) ) )
                  (bind any-substitutions? () )
            (do
                (loop (for-each-stat-operand-read stat operand)
                      (bind operand-reaching-defs
                                (stat:operand-reaching-defs stat operand) )
                      (when (== 1 (stat-set:size operand-reaching-defs) ) )
                      (bind def-stat (stat-set:choose operand-reaching-defs) )
                      (when (== 'assign (stat:operator def-stat) ) )
                      (bind constant (stat:part def-stat 'read1) )
                      (when (numberp constant) )
                (do
                    (:= (stat:source stat)
                        (oper:substitute-operand &&&
```



```
                                                 constant
                                                 operand
                                                 'read) )
                    (:= any-substitutions? t) ) )

                (if (|| any-substitutions?
                        (== 1 pass) )
                (then
                    (if (cf.stat:simplify stat) (then
                        (:= change? t) ) ) ) ) ) ) ) ) ) )

    () )


;***========================================================================
;***
;*** (CF.STAT:SIMPLIFY STAT)
;***
;*** Destructively simplifies the source of STAT using the rules described
;*** above.  Returns true if STAT is now a constant assignment (or was
;*** already).
;***
;*** This code relies on (EQUAL 0 0.0) returning true.
;***
;***========================================================================

(defun cf.stat:simplify ( stat )
    (let*( (operator (stat:operator stat) )
           (read     (stat:part     stat 'read) )
           (written  (stat:part     stat 'written) )
           (identity (cf.operator:identity operator) ) )

        (?( (&& (== 'assign operator)
                (numberp (stat:part stat 'read1) ) )
            t)

          ( (&& (memq (stat:group stat) '(one-in-one-out two-in-one-out) )
                (for-every (operand in read)
                    (numberp operand) ) )
            (:= (stat:source stat)
                `(assign ,written
                         ,(apply (operator:execute-function operator) read) ) )
            t)

          ( (&& (memq operator '(imul fmul iand) )
                (member 0 read) )
            (:= (stat:source stat)
                `(assign ,written 0) )
            t)

          ( (&& (memq operator '(idiv fdiv isub fsub) )
                (== (car read) (cadr read) ) )
            (:= (stat:source stat)
                `(assign ,written ,identity) )
            t)

          ( (&& (memq operator '(imul fmul iadd fadd iand ior) )
                (member identity read) )
            (:= (stat:source stat)
                `(assign ,written ,,(top-level-remove identity read) ) )
            () )

          ( (&& (memq operator '(idiv fdiv isub fsub) )
```

```
              (= identity (stat:part stat 'read2) ) )
          (:= (stat:source stat)
              '(assign ,written ,(stat:part stat 'read1) ) )
          () )

        ( t
          () ) ) ) ) )


;***=======================================================================
;***
;*** (CF.OPERATOR:IDENTITY OPERATOR)
;***
;*** Returns the identity constant for the given operator.
;***
;***=======================================================================

(defun cf.operator:identity ( operator )
    (caseq operator

        ( (imul idiv iand)
          1)

        ( (fmul fdiv)
          1.0)

        ( (iadd isub ior)
          0)

        ( (fadd fsub)
          0.0)

        ( t
          () ) ) )
```

```
;==============================================================================
;
; (FG.PROPAGATE-COPIES)
;
; Propagates copies produced by assignment statements through the flow
; graph.  For each statement S: A := B, all the uses of S are examined;
; if S copy reaches a use, then B can be substituted for A in the use.
; The assignment statements are processed in depth-first order so that
; copies are propagated through chains of assignments.
;
; This algorithm differs slightly from that of the Dragon Book.   The
; Dragon Book requires that every an assignment copy-reaches every use
; before we substitute in any of the uses;  thus, after substitution, the
; assignment can be deleted since it is useless.
;
; This algorithm subsitutes into each use of the assignment independently
; of whether it copy-reaches every use; we count on dead-code removal
; to remove any useless assignments so created.  Doing the substitution
; wherever possible can potentially reduce the depth of the data precedence
; graph and eliminate needless write-after-conditional-read conflicts,
; even if we can't substitute an assignment into ALL of its uses.
;
;==============================================================================

(eval-when (compile load)
    (include flow-analysis:flow-analysis-decls) )


(defun fg.propagate-copies ()
    (let ( (assign-stats () ) )

            ;*** Set ASSIGN-STATS to be all assignments.  The list is
            ;*** created in depth-first order, so that we can propagate
            ;*** copies through long chains of assignments.
            ;
        (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward) )
        (do
            (loop (for-each-bblock-stat bblock stat)
                  (when (== 'assign (stat:operator stat) ) )
            (do
                (push assign-stats stat) ) ) ) )
        (:= assign-stats (dreverse assign-stats) )

            ;*** For each assignment A := B, substitute B into each use
            ;*** of the assignment that has it as a reaching copy.  We
            ;*** destructivel change STAT:SOURCE instead of creating a
            ;*** new STAT so that we preserve the reaching-uses of the
            ;*** changed STAT; otherwise, we couldn't propagate through
            ;*** chains of assignments.
            ;
        (loop (for assign-stat in assign-stats) (do
            (loop (for use-stat in (stat:reaching-uses assign-stat) )
                  (when (!== 'live (stat:operator use-stat) ) )
                  (when (stat-set:member? (stat:reaching-copies use-stat)
                                          assign-stat) )
            (do
                (:= (stat:source use-stat)
                    (oper:substitute-operand (stat:source use-stat)
                                             (stat:part assign-stat 'read1)
                                             (stat:part assign-stat 'written)
                                             'read) ) ) ) ) )
        () ) )
```

```
;***-------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for     ********
;*** educational and research purposes only, by the faculty, students, ********
;*** and staff of Yale University only.                                ********
;***-------------------------------------------------------------------********
;==========================================================================
;
; CSE-HASH-TABLES
;
; This module implements the hash tables used by basic block common
; subexpression elimination.  There are three such tables,
;
;     one for mapping operators and operands onto previously found
;     CSE-NODEs with the same operator and operands;
;
;     one that maps a NAME onto the CSE-NODEs currently holding the value
;     of the NAME;
;
;     one that maps a NAME onto the initial (leaf) CSE-NODE whose
;     :INITIAL-NAME is NAME (the CSE-NODE that represents the NAMEs value
;     on entry into the basic block).
;
;
; (CSE.INITIALIZE-HASH-TABLES)
;     Initializes the hash tables.
;
;
; (CSE.EXPR:CSE-NODE KEY)
;     Maps KEY, a list of the form (OPERATOR OPERAND1 OPERAND2) onto the
;     CSE-NODE in the CSE DAG that has that operator and operands.  Returns
;     () if there is no such node.
;
; (CSE.EXPR:DEFINE-CSE-NODE KEY CSE-NODE)
;     Associates the CSE-NODE with KEY (see previous function) in the
;     hash table.
;
;
; (CSE.NAME:CSE-NODE NAME)
;     Returns the CSE-NODE currently representing the value of NAME (NAME
;     is in the :LABELS of CSE-NODE).  A new initial node representing
;     NAME is created if there is no node currently holding NAME.
;
; (CSE.NAME:DEFINE-CSE-NODE NAME CSE-NODE)
;     Associates CSE-NODE with NAME in the hash table that maps NAMEs
;     to the CSE-NODEs currently holding those NAMEs.
;
;
; (CSE.NAME:INITIAL-CSE-NODE NAME)
;     Returns the initial CSE-NODE that represents the value of NAME on
;     entry to the basic block.  Returns () if there is no such node.
;
; (CSE.NAME:DEFINE-INITIAL-CSE-NODE NAME CSE-NODE)
;     Associates CSE-NODE as the initial node of NAME.
;
; (CSE.NAME:INITIAL-USES NAME)
;     Returns the parents of the initial CSE-NODE of NAME; that is, all
;     the direct uses of the initial value of NAME.
;
;
; (CSE-NODE:ASSIGN-TO-NAME CSE-NODE NAME)
;     Moves the label NAME to CSE-NODE; if a previous node contained NAME
;     as a label, NAME is removed from that node.  After calling this
;     function, (CSE.NAME:CSE-NODE NAME) will return CSE-NODE.
```



```
;
; *** Note:  In the conversion to ELISP, we had to sacrifice some efficiency
; *** in hashing.  See CSE.EXPR:CSE-NODE-HASH below.
;
;==========================================================================

(eval-when (compile load)
    (include flow-analysis:flow-analysis-decls) )

(eval-when (compile)
    (build '(flow-analysis:cse-node) ) )


(declare (special
    *cse.name:cse-node*          ;*** a HASH-TABLE mapping a NAME to its
                                 ;*** current node

    *cse.name:initial-cse-node*  ;*** a HASH-TABLE mapping a NAME to its
                                 ;*** initial node.  NAME in this special
                                 ;*** case may a constant number.

    *cse.expr:cse-node*          ;*** a HASH-TABLE mapping expressions onto
                                 ;*** CSE-NODEs.

    *hash-table.not-found*
    ) )


(defun cse.initialize-hash-tables ()
    (:= *cse.name:cse-node*                          ;*** These two use EQUALT
        (hash-table:create 'equalt 'sxhash) )        ;*** because we want to
                                                     ;*** distinguish 1.0 from 1.
    (:= *cse.name:initial-cse-node*                  ;***
        (hash-table:create 'equalt 'sxhash) )        ;***


    (:= *cse.expr:cse-node*
        (hash-table:create 'cse.expr:cse-node-compare
                           'cse.expr:cse-node-hash) )
    () )


(defun cse.expr:cse-node-hash ( (operator operanda operandb) )
    (sxhash operator) )
        ;
        ;*** Note:  In MACLISP, we did MAKNUMs on the three items here
        ;*** to hash.  But since ELISP doesn't have MAKNUM (it's a copying
        ;*** GC), we just hash on the operator.  This isn't so nice,
        ;*** but will probably suffice for now;  eventually, we could put
        ;*** unique numbers in each node and use that as the hash.

(defun cse.expr:cse-node-compare ( (operator1 operand1a operand1b)
                                   (operator2 operand2a operand2b) )
    (&& (== operator1 operator2)
        (== operand1a operand2a)
        (== operand1b operand2b) ) )


(defun cse.expr:cse-node ( key )
    (let ( (cse-node (hash-table:get *cse.expr:cse-node* key) ) )
        (if (== *hash-table.not-found* cse-node)
            ()
```

```
            cse-node) ) )


(defun cse.expr:define-cse-node ( key cse-node )
    (assert (cse-node:is cse-node) )
    (hash-table:put *cse.expr:cse-node* key cse-node) )



(defun cse.name:cse-node ( name )
    (let ( (cse-node (hash-table:get *cse.name:cse-node* name) ) )

        (if (== *hash-table.not-found* cse-node) (then
            (:= cse-node (cse-node:create-initial name) )
            (hash-table:put *cse.name:initial-cse-node* name cse-node)
            (hash-table:put *cse.name:cse-node*         name cse-node)
            cse-node)

        (else
            cse-node) ) ) )


(defun cse.name:define-cse-node ( name cse-node )
    (assert (cse-node:is cse-node) )
    (hash-table:put *cse.name:cse-node* name cse-node) )




(defun cse.name:initial-cse-node ( name )
    (let ( (cse-node (hash-table:get *cse.name:initial-cse-node* name) ) )
        (if (== *hash-table.not-found* cse-node)
            ()
            cse-node) ) )


(defun cse.name:define-initial-cse-node ( name cse-node )
    (assert (cse-node:is cse-node) )
    (hash-table:put *cse.name:initial-cse-node* name cse-node) )



(defun cse.name:initial-uses ( name )
    (let ( (initial-cse-node (hash-table:get *cse.name:initial-cse-node*
                                                   name) ) )
        (if (== *hash-table.not-found* initial-cse-node) (then
            () )
        (else
            (cse-node:parents initial-cse-node) ) ) ) )


(defun cse-node:assign-to-name ( cse-node name )
    (assert (cse-node:is cse-node) )
    (let ( (old-cse-node (hash-table:get *cse.name:cse-node* name) ) )

        (if (!== *hash-table.not-found* old-cse-node) (then
            (:= (cse-node:labels old-cse-node)
                (top-level-removeq name (cse-node:labels old-cse-node) ) ) ) )
        (push (cse-node:labels cse-node) name)
        (hash-table:put *cse.name:cse-node* name cse-node) ) )
```

```lisp
;***-----------------------------------------------------------------------********
;*** Copyright (C) 1983 John R. Ellis.  This file may be used for          ********
;*** educational and research purposes only, by the faculty, students,    ********
;*** and staff of Yale University only.                                    ********
;***-----------------------------------------------------------------------********
;===========================================================================
;
; CSE-NODE
;
; This module implements the nodes of the DAG built for basic block common
; subexpression elimination.  Everything here is private to the few
; modules implementing CSE.
;
;=======================

(def-struct cse-node    ;
                        ;*** Fields used by each variant of CSE-NODE
                        ;
    name                ; The final symbolic name given to this node.
                        ;   This may be a constant number for initial nodes.
    number              ; Unique number identifying this node.
    labels              ; List of scalar NAMEs whose value is this node.
    (children           ; List of CSE-NODEs representing the operands.
        () suppress)    ;
    (pseudo-children    ; List of nodes which must precede this node
        () suppress)    ;   in any linear order but which aren't in
                        ;   :CHILDREN.
    (parents            ; List of nodes for which this node is in
        () suppress)    ;   :CHILDREN or :PSEUDO-CHILDREN.
    remaining-count     ; Number of children (including both :CHILDREN
                        ;   and :PSEUDO-CHILDREN) that remain to be
                        ;   evaluated before this one can be.
   (descendants         ; Bit set of descendants (via :CHILDREN
    *bit-set.empty-set*);   and :PSEUDO-CHILDREN) of this node.
                        ;
    datatype            ; The datatype of this node, INTEGER, FLOAT, or
                        ; () for unknown.
                        ;
                        ;*** For interior nodes
                        ;
    source              ; The source operation for this node; () if not
                        ;   an interior node.
    miscellaneous?      ; True if this node doesn't do anything useful.
    killed?             ; True if this node is a VLOAD or VSTORE and
                        ;   its value is no longer available (because
                        ;   a subsequent VSTORE invalidated it).
                        ;
                        ;*** For initial nodes
                        ;
    initial-name        ; For initial nodes, the NAME;  () otherwise.
                        ;   This may be a constant number.
    (vloads-vstores     ; For a vector name initial node, the list
        () suppress)    ;   of CSE-NODEs that are VLOADs or VSTOREs from
                        ;   the vector.
    )                   ;
;=======================
;
; *CSE.ALL-CSE-NODES*
;     A list of all CSE-NODEs in the current DAG.
;
; (CSE.INITIALIZE-CSE-NODES)
;     Forgets the previous DAG of CSE-NODEs and prepares for the building
;     of a new one.
;
```



```lisp
;
; (CSE-NODE:CREATE)
;     Builds a new CSE-NODE.
;
; (CSE-NODE:CREATE-INITIAL INITIAL-NAME)
;     Creates a leaf node in the DAG with name INITIAL-NAME (either a
;     variable name or a constant).
;
; (CSE-NODE:CREATE-INTERIOR SOURCE CHILDREN)
;     Creates an interior node in the DAG with NADDR operation SOURCE
;     and with CHILDREN, a list of child CSE-NODEs that form the operands.
;
; (CSE-NODE:ADD-PSEUDO-CHILD CSE-NODE CHILD-CSE-NODE)
;     Makes CHILD-CSE-NODE a pseudo-child of CSE-NODE (a pseudo child
;     is one that must be evaluated before the parent, but is not an
;     operand of the parent).
;
; (CSE-NODE:MAKE-COPY-CSE-NODES CSE-NODE)
;     For each label of CSE-NODE, builds a parent xASSIGN node that copies
;     the value of CSE-NODE into the label.  :LABELS of CSE-NODE is set
;     to ().
;
; (CSE-NODE:INITIAL? CSE-NODE)
;     True if CSE-NODE is an initial node (leaf node).
;
; (CSE-NODE:ANCESTOR? CSE-NODE1 CSE-NODE2)
;     True if CSE-NODE1 is an ancestor of CSE-NODE2.
;
;===========================================================================


(eval-when (compile load)
    (include flow-analysis:flow-analysis-decls) )


(declare (special
    *cse.all-cse-nodes*          ;*** A list of all CSE-NODEs created.
    *cse.total-cse-nodes*        ;*** Total number of CSE-NODEs created.
    ) )


(defun cse.initialize-cse-nodes ()
    (:= *cse.all-cse-nodes*   () )
    (:= *cse.total-cse-nodes* 0)
    () )


(defun cse-node:create ()
    (let ( (cse-node (cse-node:new) ))
        (push *cse.all-cse-nodes* cse-node)
        (++ *cse.total-cse-nodes*)
        (:= (cse-node:number cse-node) *cse.total-cse-nodes*)
        cse-node) )


(defun cse-node:create-initial ( initial-name )
    (let ( (cse-node (cse-node:create) ) )
        (:= (cse-node:initial-name cse-node) initial-name)
        cse-node) )


(defun cse-node:create-interior ( source children )
    (assert (consp source) )
    (let ( (cse-node (cse-node:create) ) )
```

```
        (:= (cse-node:source   cse-node) source)
        (:= (cse-node:datatype cse-node) (oper:dest-datatype source) )
        (:= (cse-node:children cse-node) children)
        (:= (cse-node:descendants cse-node)
            (cse-node:calculate-descendants cse-node) )

        (loop (for child-cse-node in children) (do
            (push (cse-node:parents child-cse-node) cse-node) ) )

        cse-node) )


(defun cse-node:calculate-descendants ( cse-node )
    (assert (cse-node:is cse-node) )
    (let ( (descendants *bit-set.empty-set*) )

        (loop (for child-cse-node in (cse-node:children cse-node) ) (do
            (:= descendants
                (bit-set:union
                    (bit-set:union1 (cse-node:descendants child-cse-node)
                                    (cse-node:number      child-cse-node) )
                    descendants) ) ) )
        (loop (for child-cse-node in (cse-node:pseudo-children cse-node) ) (do
            (:= descendants
                (bit-set:union
                    (bit-set:union1 (cse-node:descendants child-cse-node)
                                    (cse-node:number      child-cse-node) )
                    descendants) ) ) )

        descendants) )

(defun cse-node:add-pseudo-child ( cse-node child-cse-node )
    (assert (cse-node:is cse-node) )
    (assert (cse-node:is child-cse-node) )

    (push (cse-node:pseudo-children cse-node)       child-cse-node)
    (push (cse-node:parents         child-cse-node) cse-node      )
    (cse-node:propagate-new-descendants cse-node)
    () )

(defun cse-node:propagate-new-descendants ( cse-node )
    (let ( (new-descendants (cse-node:calculate-descendants cse-node) ) )
        (if (! (bit-set:= new-descendants
                          (cse-node:descendants cse-node) ) )
        (then
            (:= (cse-node:descendants cse-node) new-descendants)
            (loop (for parent-cse-node in (cse-node:parents cse-node) ) (do
                (cse-node:propagate-new-descendants parent-cse-node) ) ) ) ) )
    () )


(defun cse-node:make-copy-cse-nodes ( cse-node )
    (assert (|| (! (cse-node:labels cse-node) )
                (cse-node:datatype cse-node) ) )

    (loop (for name in (cse-node:labels cse-node) )
          (initial result-list   ()
                   copy-cse-node () )
    (do
        (:= copy-cse-node
            (cse-node:create-interior
                (if (== 'integer (cse-node:datatype cse-node) )
                    `(iassign dummy dummy)
```



```
                    `(fassign dummy dummy) )
                  `(,cse-node) ) )
        (cse-node:assign-to-name copy-cse-node name)
        (push result-list copy-cse-node) )
    (result
        (:= (cse-node:labels cse-node) () )
        result-list) ) )


(defun cse-node:initial? ( cse-node )
    (assert (cse-node:is cse-node) )
    (cse-node:initial-name cse-node) )



(defun cse-node:ancestor? ( cse-node1 cse-node2 )
    (assert (cse-node:is cse-node1) )
    (assert (cse-node:is cse-node2) )

    (bit-set:member? (cse-node:descendants cse-node1)
                     (cse-node:number      cse-node2) ) )
```

```
;==========================================================================
; DEPENDENCIES
;
; This module implements a simple Unix MAKE-like facility for keeping
; track of which transformations and computations on the flow graph are
; still valid.  For example, FG.MOVE-LOOP-INVARIANTS destroys several
; computed values, such as the dominator info, because it changes the
; block structure of the flow graph.  Any succeeding function that needs
; the loop information will first have to recompute the loop info.  We
; want to automate the process of recompuating values so that we have
; the flexibility of reordering or turning off transformations.
;
; (DEF-DEPENDENCY FUNCTION
;     (NEEDS    N-F1 [N-F2 ...] )
;     (DESTROYS D-F1 [D-F2 ...] )
;     (REINITIALIZE R-F) )
;
;     States that FUNCTION needs the values computed by the functions
;     N-Fi before it can be executed; the values computed by the functions
;     D-Fi are destroyed by invoking FUNCTION.  The functions N-Fi and
;     D-Fi should have been declared in DEF-DEPENDENCYs.  If another
;     dependency function destroys the values computed by FUNCTION, the
;     reinitializing function R-F will be called after the destroying
;     function finishes.  R-F is also called at the very beginning of
;     flow analysis.
;
; (FG.INITIALIZE-DEPENDENCIES)
;     Forgets all previous dependencies declared by DEF-DEPENDENCY.
;
; (FG.INITIALIZE-DEPENDENT-FUNCTIONS)
;     Forgets about any previously computed dependency functions, invoking
;     all the reinitializers whether the corresponding function was
;     previously computed.
;
; (FG.INVOKE-DEPENDENT-FUNCTION FUNCTION)
;     If FUNCTION has already been invoked and not subsequently destroyed,
;     this will do nothing.  Otherwise, it calls the specified function,
;     which should have been declared in a DEF-DEPENDENCY.  Before invoking
;     FUNCTION, all the functions specified in the NEEDS clause of the
;     corresponding DEF-DEPENDENCY are recursively invoked if they have
;     not been previously computed.  After FUNCTION terminates, the
;     functions specified in the DESTROYS clause are forgotten; that is,
;     the next time they are invoked by FG.INVOKE-DEPENDENT-FUNCTION either
;     directly or implicitly by a NEEDS clause, the destroyed functions
;     will be called to recompute their values.  Any functions depending
;     on the destroyed functions are recursively destroyed.
;
; (FG.DESTROY-DEPENDENT-FUNCTION FUNCTION)
;     "Forgets" the fact that FUNCTION may have been previously computed.
;     Any functions that depend on FUNCTION via a NEEDS clause will also
;     be recursively destroyed.  The next time FG.INVOKE-DEPENDENT-FUNCTION
;     is called, the a "forgotten" function will actually be recomputed.
;
;==========================================================================

(include flow-analysis:flow-analysis-decls)

(defvar *dep.computed-functions* () )
                                ;*** List of all function names that
                                ;*** have been previously computed
                                ;*** and not subsequently destroyed.
```



```
(def-struct dep.dependency  ;*** One such record per DEF-DEPENDENCY.
    function                ;*** The name of the function.
    needs                   ;*** The list of functions needed.
    destroys                ;*** The list of functions destroyed.
    reinitialize            ;*** The reinitializing function.
    )


(defun dep.function:dependency ( function )
    (loop (for dep in *fg.all-dependencies*) (do
        (if (== function (dep.dependency:function dep) ) (then
            (return dep) ) ) )
    (result () ) ) )


(defmacro def-dependency args
    '(dep.define-dependency ',args) )

(defun dep.define-dependency ( (function . clauses ) )
    (let ( (dep (dep.dependency:new function function) ) )
        (assert (litatom function) )

        (loop (for clause in clauses) (do
            (assert (consp clause) )
            (caseq (car clause)
                (needs
                    (:= (dep.dependency:needs        dep) (cdr  clause) ) )
                (destroys
                    (:= (dep.dependency:destroys     dep) (cdr  clause) ) )
                (reinitialize
                    (:= (dep.dependency:reinitialize dep) (cadr clause) ) )
                (t
                    (error (list clause
                                 "Invalid DEF-DEPENDENCY syntax.") ) ) ) ) )

        (:= *fg.all-dependencies*
            (top-level-remove (dep.function:dependency function)
                              *fg.all-dependencies*) )
        (push *fg.all-dependencies* dep)
        function) )


(defun fg.initialize-dependencies ()
    (:= *fg.all-dependencies* () ) )


(defun fg.initialize-dependent-functions ()
    (loop (for dep in *fg.all-dependencies*) (do
        (let ( (reinitialize (dep.dependency:reinitialize dep) ) )
            (if reinitialize (then
                (msg 0 "Invoking " reinitialize t)
                (funcall reinitialize)
                () ) ) ) ) ) )

    (:= *dep.computed-functions* () )
    () )


(defvar *dep.nesting* 0)

(defun fg.invoke-dependent-function ( function )
    (let ( (dep           (dep.function:dependency function) )
           (*dep.nesting* (+ 1 *dep.nesting*) ) )
```

```
        (assert dep)

        (if (! (memq function *dep.computed-functions*) ) (then

                    ;*** Recursively invoke any needed functions
                    ;
            (loop (for needed-function in (dep.dependency:needs dep) ) (do
                (fg.invoke-dependent-function needed-function) ) )

                    ;*** Invoke the function.
                    ;
            (msg 0 (t (* 2 *dep.nesting*) ) "Invoking " function t)
            (funcall function)

                    ;*** Destroy the specified functions
                    ;
            (loop (for destroyed-function in (dep.dependency:destroys dep) )
            (do
                (fg.destroy-dependent-function destroyed-function) ) )

                    ;*** Remember the function as computed
                    ;
            (if (! (memq function *dep.computed-functions*) ) (then
                (push *dep.computed-functions* function) ) ) ) ) )

    *dep.computed-functions*)


(defun fg.destroy-dependent-function ( function )
    (let ( (dep            (dep.function:dependency function) )
           (*dep.nesting* (+ 1 *dep.nesting*) ) )
        (assert dep)

                ;*** Recursively destroy all functions that NEED this
                ;*** function
                ;
        (loop (for next-dep in *fg.all-dependencies*) (do
            (if (memq function (dep.dependency:needs next-dep) ) (then
                (fg.destroy-dependent-function
                    (dep.dependency:function next-dep) ) ) ) ) )

                ;*** Call this functions reinitializer if it has one.
                ;
        (let ( (reinitialize (dep.dependency:reinitialize dep) ) )
            (if (&& reinitialize
                    (memq function *dep.computed-functions*) )
            (then
                (msg 0 (t (* 2 *dep.nesting*) ) "Destroying "
                     function t)
                (funcall reinitialize)
                () ) ) )

                ;*** Remove the funtion from the list of remembered functions.
                ;
        (:= *dep.computed-functions*
            (top-level-removeq function *dep.computed-functions*) )
        () ) )
```

```
;***=========================================================================
;***
;*** This module orders the bblocks of the flow graph by depth first order.
;***
;*** (FG.SET-DEPTH-FIRST-ORDER)
;*** (FG.INITIALIZE-DEPTH-FIRST-ORDER)
;***     This should be called each time the basic block structure of the
;***     flow graph changes.  The two functions are synonyms for clarity
;***     in the function dependencies (ughh).
;***
;*** (FG.DEPTH-FIRST-ORDERED-BBLOCK-LIST ORDERING)
;***     Returns a list of BBLOCKs in depth first order (ORDERING is either
;***     REVERSE or FORWARD).  The results of the last call to this function
;***     are remembered, so that a depth first search is actually done
;***     only once.  The :DFO-NUMBER of each BBLOCK is set to its position
;***     in the forward ordering.
;***
;***=========================================================================

(include flow-analysis:flow-analysis-decls)

(declare (special
    *dfo.current-order*      ;*** one of (), REVERSE, or FORWARD
    *dfo.ordered-list*       ;*** order list of BBLOCKs.
    *dfo.visited-bblocks*    ;*** bit set of currently visited BBLOCKs.

    *bit-set.empty-set*
    ) )

(defun fg.set-depth-first-order ()
    (fg.initialize-depth-first-order) )

(defun fg.initialize-depth-first-order ()
    (:= *dfo.current-order* () )
    (:= *dfo.ordered-list*  () ) )



(defun fg.depth-first-ordered-bblock-list ( ordering )
    (assert (&& (litatom ordering)
                (memq ordering '(reverse forward) ) ) )

    (if (! *dfo.current-order*) (then
        (:= *dfo.visited-bblocks* *bit-set.empty-set*)
        (dfo.search *fg.entry-bblock*)

        (:= *dfo.visited-bblocks* *bit-set.empty-set*)
        (:= *dfo.current-order* 'forward)

        (loop (for bblock in *dfo.ordered-list*)
              (incr i from 1)
        (do
            (:= (bblock:dfo-number bblock) i) ) ) ) )

    (if (== ordering *dfo.current-order*) (then
        *dfo.ordered-list*)
    (else
        (:= *dfo.current-order* ordering)
        (:= *dfo.ordered-list* (dreverse *dfo.ordered-list*) ) ) ) )
```

1

```
(defun dfo.search ( bblock )
    (:= *dfo.visited-bblocks* (bit-set:union1 *dfo.visited-bblocks*
                                              (bblock:number bblock) ) )

    (for (succ-bblock in (bblock:succs bblock) ) (do
        (if (! (bit-set:member? *dfo.visited-bblocks*
                                (bblock:number succ-bblock) ) )
            (dfo.search succ-bblock) ) ) )

    (push *dfo.ordered-list* bblock) )
```

2

```
;==========================================================================
;
; DERVIATIONS
;
; This module provides "derivations" of integer variable definitions.
; A derivation of a variable is an expression for the variable in terms
; of the induction variables of the innermost loop containing the
; variable's definition; if the definition is not contained in a loop,
; then the expression is in terms of initial program inputs.
;
; The derivation expressions are diophantine expressions, which are defined
; elsewhere; the recognized operators are +, *, -, and &.  & expresses
; alternative -- its operands are the alternative derivations for a
; definition.
;
; Derivations are computed by first, for each loop, finding the set of
; induction variables of that loop and splicing in a dummy BBLOCK loop
; header at the top of the loop that looks like:
;
;     (LOOP-ASSIGN I1)
;     (LOOP-ASSIGN I2)
;     ...
;     (LOOP-ASSIGN In)
;
; where the Ii are the loop's induction variables.  In this module, a
; variable I is an induction variable of a loop if:
;
;     1. The variable has a definition in the loop.
;     2. That definition reaches the loop header via one of the back-edges.
;     3. The variable is live on entry to the header of the loop.
;
; After the dummy loop assignments are inserted, reaching definitions
; are re-calculated.  This lets us easily identify which reaching
; definitions that reach an operand are induction variables defined in
; the previous iteration of the loop body.
;
; A derivation for a definition is found by recursively tracing the
; reaching definitions for each operand backwards.  The backwards chaining
; stops at definitions that aren't one of the following known operators:
; IADD, IMUL, ISUB, INEG, and ASSIGN.  Note specifically that the backwards
; chaining stops at a LOOP-ASSIGN that doesn't specify an equivalent
; expression for the loop induction variable.  The unknowns of the
; derivation are the STATs for which a derivation cannot be derived.
; Multiple reaching definitions for a variable are included in the
; derivation using the & operator.
;
; Derivations for a definition are remembered as they are found in the
; field STAT:KNOWN-DERIVATION, so getting a derivation for all expressions
; is linear in the size of the flow graph.
;
; (FG.INITIALIZE-DERIVATIONS)
;     Initializes this module.
;
; (FG.INSERT-LOOP-ASSIGNMENTS)
;     Inserts a (LOOP-ASSIGN I) for each induction variable of the loop
;     at the beginning of the loop header.
;
; (STAT:DERIVATION STAT)
;     Returns the diophantine expression representing the derivation of
;     the variable defined by STAT.  The leaves of the expression are
;     STATs and constants.
;
; (STAT:OPERAND-DERIVATION STAT OPERAND)
```



```
;     Returns the derivation of an operand of STAT; OPERAND is just a
;     variable name that is read by STAT.
;
; (STAT:INDEX-DERIVATION STAT)
;     Returns the derivation of the index operand of STAT, which should
;     be a vector reference.
;
;==========================================================================


(include flow-analysis:flow-analysis-decls)


;***=======================================================================
;***
;*** (FG.INITIALIZE-DERIVATIONS)
;***
;***=======================================================================

(defun fg.initialize-derivations ()
    () )


;***=======================================================================
;***
;*** (FG.INSERT-LOOP-ASSIGNMENTS)
;***
;*** This might be speeded up (who cares?) and simplified (I care) by
;*** just looking at the definitions that reach the loop header;  it isn't
;*** necessary to consider each back edge individually.
;***
;***=======================================================================

(defun fg.insert-loop-assignments ()
    (loop (for-each-loop loop)
          (initial induction-vars () )
    (do
        (:= induction-vars () )

                 ;*** Gather the set of induction variables of the loop
                 ;
        (loop (for-each-bblock-set-element (loop:back-edges loop)
                                           back-edge-bblock)
        (do
            (:= induction-vars
                (unionq induction-vars
                        (der.loop:back-edge-bblock:induction-vars
                             loop back-edge-bblock) ) ) ) )

                 ;*** Insert a (LOOP-ASSIGN I) at the top of the loop header
                 ;*** for each induction variable I.
                 ;
        (loop (for var in induction-vars) (do
            (stat:insert-stat
                (bblock:first-stat (loop:header loop) )
                (stat:create `(loop-assign ,var) ) ) ) ) ) )

    () )


;***=======================================================================
;***
;*** (DER.LOOP:BACK-EDGE-BBLOCK:INDUCTION-VARS LOOP BACK-EDGE-BBLOCK)
```

```
;**
;*** Returns a list of the induction variables of LOOP that have
;*** definitions reaching the loop header via the back-edge whose tail
;*** is BACK-EDGE-BBLOCK.
;***
;***=====================================================================

(defun der.loop:back-edge-bblock:induction-vars ( loop back-edge-bblock )
    (let ( (induction-names () )
           (edge-reaching-defs
               (stat-set:intersection
                   (loop:stats loop)
                   (bblock:reaching-out back-edge-bblock)
                   (bblock:reaching-in  (loop:header loop) ) ) ) )

               ;*** EDGE-REACHING-DEFS is the set of definitions contained
               ;*** in the loop that reach the loop header via the back
               ;*** edge.
               ;***
               ;*** Enumerate over all the variables that are live on
               ;*** entry to the loop header, and if any one of them
               ;*** has definitions in EDGE-REACHING-DEFS, it is an
               ;*** induction variable.
               ;
        (loop (for-each-name-set-element (bblock:live-in (loop:header loop) )
                                         live-name)
        (do
            (if (! (stat-set:= *fg.empty-stat-set*
                               (stat-set:intersection
                                   (name:defining-stats live-name)
                                   edge-reaching-defs) ) )
            (then
                (push induction-names live-name) ) ) ) )

        induction-names) )


;***=====================================================================
;***
;*** (STAT:DERIVATION STAT)
;***
;***=====================================================================

(defun stat:derivation ( stat )
    (assert (stat:is stat) )

    (if (stat:known-derivation stat) (then
        (stat:known-derivation stat) )
    (else
        (:= (stat:known-derivation stat)
            (caseq (stat:operator stat)
                ( (iadd isub imul)
                    `(,(caseq (stat:operator stat)
                          (iadd '+)
                          (isub '-)
                          (imul '*) )
                      ,(stat:operand-derivation
                           stat (stat:part stat 'read1) )
                      ,(stat:operand-derivation
                           stat (stat:part stat 'read2) ) ) )
                (ineg
                    `(- ,(stat:operand-derivation
                           stat (stat:part stat 'read1) ) ) )
                (assign
                    (stat:operand-derivation
                        stat (stat:part stat 'read1) ) )
                (loop-assign
                    (if-let ( (read2 (stat:part stat 'read2) ) )
                        (stat:operand-derivation stat read2)
                        stat) )

                (t
                    stat) ) ) ) ) )


;***=====================================================================
;***
;*** (STAT:OPERAND-DERIVATION STAT)
;***
;***=====================================================================

(defun stat:operand-derivation ( stat operand )
    (if (numberp operand) (then
        operand)
    (else
        (if-let ( (var&derivation
                      (assoc operand (stat:known-operand-derivations stat) ) ) )
        (then
            (cadr var&derivation) )
        (else
            (let*( (reaching-defs (stat:operand-reaching-defs stat operand))
                   (derivation
                       (loop (for-each-stat-set-element reaching-defs
                                                        def-stat)
                             (initial result () )
                       (do
                           (push result (stat:derivation def-stat) ) )
                       (result `(& ,,(dreverse result) ) ) ) ) )
                (push (stat:known-operand-derivations stat)
                      `(,operand ,derivation) )
                derivation) ) ) ) ) )


;***=====================================================================
;***
;*** (STAT:INDEX-DERIVATION STAT)
;***
;***=====================================================================

(defun stat:index-derivation ( vector-stat )
    (stat:operand-derivation
        vector-stat
        (stat:part vector-stat 'index) ) )
```

```
;==============================================================================
;
; DISAMBIGUATOR
;
; This module implements the interface functions of the "disambiguator"
; as described in DOC:DISAMB.DOC.  The description of the interface won't
; be repeated here (to make sure that there is one, and only one,
; description that is kept up to date).
;
; (FG.INITIALIZE-NADDR:STAT-MAPPING)
; (FG.INITIALIZE-DISAMBIGUATOR)
;     Initializes this module, clearing any old data structures.
;
; (FG.DISAMBIGUATE)
;     Prepares for disambiguation by:
;
;        1. Adding all the assertions to the assertion database.
;        2. Obtaining and storing derivations for each vector index.
;
; (FG.CREATE-NADDR:STAT-MAPPING)
;     Creates the mapping from source NADDR to STATs.
;
; (START-TRACE)
;     See DOC:DISAMB.DOC.
;
; (PREDECESSORS SOURCE-OPERATION TRACE-DIRECTION DATUM)
;     See DOC:DISAMB.DOC.
;
; (OPER:LIVE-IN OPER)
;     See DOC:DISAMB.DOC.
;
; (OPER:LIVE-OUT OPER)
;     See DOC:DISAMB.DOC.
;
; (OPER:LIVE-OUT-ON-EDGE OPER DIRECTION)
;     See DOC:DISAMB.DOC.
;
;==============================================================================


(include flow-analysis:flow-analysis-decls)



;***===========================================================================
;***
;*** DIS.RW
;***
;*** Each read and write of a variable on the trace presented to the
;*** disambiguator via PREDECESSORS is represented using a DIS.RW (read/write)
;*** record.
;***
;***********************
(def-struct dis.rw   ;***
    stat             ;*** The STAT reading/writing.
    name             ;*** The scalar or vector variable being read/written
                     ;***    (or name of of the vector).
    name-type        ;*** SCALAR or VECTOR.
    operand-number   ;*** Operand number of the variable in the NADDR
                     ;***    operation (STAT:PART numbering).
    type             ;*** READ, CONDITIONAL-READ, WRITTEN
    (datum           ;*** Random codegenerator value associated with :STAT.
        () suppress);***
```

```
    )                ;***
;*****************;***
;***
;*** All the DIS.RW records representing read/writes on the trace that
;*** haven't yet been killed by new read/writes are stored in an
;*** association list mapping a name onto the DIS.RW records
;*** reading/writing the name.  As each new element in the trace is
;*** presented, the mapping is updated -- DIS.RW records representing
;*** killed variables are removed, and records representing the reads
;*** and writes of the new element are added.
;***
;*****************;***
(declare (special    ;***
    *dis.name:rws*   ;*** Hash table mapping variable names onto list of the
    ) )              ;*** form:
                     ;***
                     ;***     (RW1 RW2 RW3 ...)
                     ;***
                     ;*** where RW1 are DIS.RW records representing the
                     ;*** reads/writes of a variable VAR (scalar or vector).
;**********************
;***
;*** (DIS.NAME:RWS NAME)
;***     Maps a name to a list of DIS.RW records describing reads and writes
;***     of that name.  NAME may be a vector.
;***
;*** (DIS.NAME:DEFINE-RWS NAME RWS)
;***     Associates a list of DIS.RW records (RWS) with NAME, removing any
;***     previous association.
;***
;*** (DIS.NAME:ADD-RW NAME RW)
;***     Adds RW to the list of DIS.RW records associated with NAME.
;***
;***===========================================================================


(defun dis.name:rws ( name )
    (let ( (rws (hash-table:get *dis.name:rws* name) ) )
        (if (== rws *hash-table.not-found*)
            ()
            rws) ) )


(defun dis.name:define-rws ( name rws )
    (hash-table:put *dis.name:rws* name rws) )


(defun dis.name:add-rw ( name rw )
    (let ( (rws (hash-table:get *dis.name:rws* name) ) )
        (if (== rws *hash-table.not-found*) (then
            (hash-table:put *dis.name:rws* name `(,rw) ) )
        (else
            (hash-table:put *dis.name:rws* name `(,rw ,.rws) ) ) ) ) )


;***===========================================================================
;***
;*** The client of PREDECESSORS identifies operations by handing in the
;*** lists representing the NADDR operations, and the disambiguator needs
;*** to find out which STAT that corresponds too.  So we use a hash table:
;***
;**************************
(declare (special         ;***
    *dis.oper:stat*       ;*** A hash table created by HASH-TABLE:CREATE
```

```
                              ;*** that has NADDR operations (lists) as EQ keys,
    ) )                       ;*** and STATs as associated values.
;***************************
;***========================================================================


(defun fg.initialize-disambiguator ()
    (:= *dis.name:rws* () )
    () )


(defun fg.initialize-naddr:stat-mapping ()
    (:= *dis.oper:stat* (hash-table:create) )
    () )


(defun fg.disambiguate ()

        ;*** Collect all the assertions in the program.
        ;
    (loop (for-each-stat stat)
          (when (== 'assert (stat:operator stat) ) )
    (do
        (de:assert (stat:part stat 'compare-op)
                   (stat:operand-derivation stat (stat:part stat 'read1) )
                   (stat:operand-derivation stat (stat:part stat 'read2) )
                   stat) ) )

        ;*** Obtain derivations for the indices of vector references.
        ;
    (loop (for-each-stat stat)
          (when (stat:property? stat 'vector-reference) )
    (do
        (stat:index-derivation stat) ) )

    () )


(defun fg.create-naddr:stat-mapping ()
    (loop (for-each-stat stat) (do
        (hash-table:put *dis.oper:stat* (stat:source stat) stat) ) )
    () )


(defun start-trace ()
    (:= *dis.name:rws* (hash-table:create) )
    () )


(defun predecessors ( source-operation trace-direction datum )
    (let ( (stat (hash-table:get *dis.oper:stat* source-operation) ) )
        (assert (!== stat *hash-table.not-found*) )

        (dis.stat:conflicts
            stat
            trace-direction
            datum) ) )


;***========================================================================
;***
;*** (DIS.TEST STAT-NUMBER-LIST)
;***
```



```
;*** This is a test function for testing out this module.  STAT-NUMBER-LIST
;*** is a list of STAT numbers that are to be the trace; conditional jumps
;*** are always assumed to go to the right.  The trace predecessors of each
;*** trace STAT are printed out in complete, gory detail.
;***
;***========================================================================

(defun dis.test ( stat-number-list )
    (fg.initialize-disambiguator)

    (loop (for number in stat-number-list)
          (initial stat () )
          (next    stat (number:stat number) )
    (do
        (msg 0 t number ": " (stat:source stat) t)
        (loop (for (pred reason source-operand source-type
                                pred-operand   pred-type)
                   in (dis.stat:conflicts stat 'right stat) )
        (do
            (msg "  " (j (stat:number pred) 2)
                 ": " (j (stat:source pred) -20)
                 "  " (j (if (== 'operand-conflict reason) 'yes 'maybe) -5)
                 "  " (j source-operand 2)
                 ":"  (caseq source-type (read 'r) (written 'w) (t 'c) )
                 ":"  (j pred-operand 2)
                 ":"  (caseq pred-type   (read 'r) (written 'w) (t 'c) ) ) ) )
        ) ) )

;***========================================================================
;***
;*** (DIS.STAT:CONFLICTS STAT TRACE-DIRECTION DATUM)
;***
;*** Returns the conflicts between STAT (the next element of the trace)
;*** and previous trace elements.  The returned conflicts are in the form
;*** as specified for PREDECESSORS by DISAMB.DOC.  TRACE-DIRECTION is
;*** the direction a conditional-jump takes, and DATUM is the arbitrary
;*** client value associated with STAT.
;***
;*** As a side effect, the mapping between names and read/written variables
;*** is updated to reflect the reading/writing done by STAT.
;***
;***========================================================================

(defun dis.stat:conflicts ( stat trace-direction datum )

(let*( (stat-written-rw
            (dis.stat:create-written-rw stat datum trace-direction) )
                ;
                ;*** The DIS.RW record representing the variable written
                ;*** by STAT ( () if there is no written variable).

       (stat-written-name
            (if stat-written-rw (dis.rw:name stat-written-rw) () ) )
                ;
                ;*** The variable name written.

       (stat-read-rws
            (dis.stat:create-read-rws stat datum trace-direction) )
                ;
                ;*** The DIS.RW records representing the variables read
                ;*** by STAT.

       (conflicts    () )
```

```
            ;*** The conflicts that will be the result of this function.

    (new-prev-rws () ) )
            ;*** The new list of DIS.RW records representing the
            ;*** read/writes of the variable written by STAT.  This
            ;*** list is the old such list, plus any reads of the
            ;*** variable by STAT, minus all reads/writes killed by
            ;*** by STAT.

        ;*** Construct the conflicts between the operands read by
        ;*** this STAT and previous STATs.  Add the STAT's RW to the
        ;*** mapping NAME:RWS for each read name (only if the name
        ;*** isn't also defined by this stat).
        ;
  (loop (for stat-read-rw in stat-read-rws) (do
      (loop (for prev-rw in (dis.name:rws (dis.rw:name stat-read-rw) ) )
      (do
          (if-let ( (conflict (dis.rw:rw:conflict stat-read-rw prev-rw) ) )
              (push conflicts conflict) ) ) )

      (if (!== stat-written-name (dis.rw:name stat-read-rw) ) (then
          (dis.name:add-rw (dis.rw:name stat-read-rw) stat-read-rw) ) ) ) )

        ;*** Construct the conflicts between the operand written by
        ;*** this STAT and previous STATs.  Remember all the previous
        ;*** RWs listed under the name written by STAT that weren't
        ;*** definitely known conflicts,  forgetting all others (the
        ;*** ones killed by STAT); add the the write RW to the list.
        ;
  (if stat-written-rw (then
      (loop (for prev-rw in (dis.name:rws stat-written-name) ) (do
          (if-let ( (conflict (dis.rw:rw:conflict stat-written-rw prev-rw)))
          (then
              (push conflicts conflict)
              (if (== 'possible-operand-conflict (cadr conflict) ) (then
                  (push new-prev-rws prev-rw) ) ) )
          (else
              (push new-prev-rws prev-rw) ) ) ) )

      (push new-prev-rws stat-written-rw)
      (dis.name:define-rws stat-written-name new-prev-rws) ) )

  conflicts) )

;***=========================================================================
;***
;*** (DIS.RW:RW:CONFLICT RW PREV-RW)
;***
;*** Returns the PREDECESSORS-format conflict representing the conflict,
;*** if any, between a read/write of a variable by one trace element and
;*** the read/write of the same variable by another previous trace element.
;*** For example, a read of a variable (PREV-RW) followed by another read
;*** (RW) is not a conflict, but a write after a read would be.
;***
;*** The form of the conflict (from DISAMB.DOC) is:
;***
;*** (PRED REASON SOURCE-OPERAND SOURCE-TYPE PRED-OPERAND PRED-TYPE)
;***
;***=========================================================================

(defun dis.rw:rw:conflict ( rw prev-rw )
(let ( (conflict-type () ) )
```



```
  (if (|| (== 'written (dis.rw:type rw) )
          (&& (== 'read    (dis.rw:type rw) )
              (== 'written (dis.rw:type prev-rw) ) ) )
  (then
      (? ( (!== 'vector (dis.rw:name-type rw) )
           (:= conflict-type
               (if (== 'conditional-read (dis.rw:type prev-rw) )
                   'conditional-conflict
                   'operand-conflict) ) )

         ( (== 'conditional-read (dis.rw:type prev-rw) )
           (:= conflict-type 'conditional-conflict) )

         ( t
           (let ( (equal? (de:possibly-equal?
                              (stat:index-derivation (dis.rw:stat rw) )
                              (stat:index-derivation (dis.rw:stat prev-rw))
                              (dis.rw:stat rw) ) ) )
              (if (|| (== 'maybe equal?)
                      (== 'yes   equal?) )
              (then
                  (:= conflict-type
                      (if (== 'maybe equal?)
                          'possible-operand-conflict
                          'operand-conflict) ) ) ) ) ) ) ) )

  (if conflict-type (then
      '(,(dis.rw:datum            prev-rw)
        ,conflict-type
        ,(dis.rw:operand-number rw)
        ,(dis.rw:type           rw)
        ,(dis.rw:operand-number prev-rw)
        ,(dis.rw:type           prev-rw) ) )
  (else
      () ) ) ) ) )

;***=========================================================================
;***
;*** (DIS.STAT:CREATE-READ-RWS STAT DATUM TRACE-DIRECTION)
;***
;*** Returns a list of DIS.RW records representing the variables read
;*** by STAT, including the "conditional-reads" of variables that are
;*** live on the off-trace edge.  For now, we assume that every vector
;*** is live on every off-trace edge.
;***
;*** *** But we now do live analysis of vector names.  A vector write
;*** *** should move up a above a conditional jump if the vector is dead
;*** *** on the off-trace edge, no?
;***
;***=========================================================================

(defun dis.stat:create-read-rws ( stat datum trace-direction )
  (let ( (result () ) )

              ;*** If this is a conditional jump, make a record for every
              ;*** variable that is live on the off-trace edge.  Also make
              ;*** make a record for every array name, assuming for now
              ;*** that every vector that is read on the off-trace edge.
              ;
    (if (stat:property? stat 'conditional-jump) (then
        (let*( (bblock      (stat:bblock stat) )
```

```
                (succ-bblock (if (== 'left trace-direction)
                                 (cadr (bblock:succs bblock) )
                                 (car  (bblock:succs bblock) ) ) )
                (succ-stat   (bblock:first-stat succ-bblock) ) )

                   ;*** First the off-trace live scalars.
                   ;
              (loop (for-each-name-set-element (stat:live-in succ-stat)
                                               live-name)
                  (when (!== 'vector (name:type live-name) ) )
              (do
                  (push result
                      (dis.rw:new
                         stat           stat
                         name           live-name
                         name-type      (name:type live-name)
                         operand-number ()
                         type           'conditional-read
                         datum          datum) ) ) )


                   ;*** Then all array names.
                   ;
              (loop (for-each-vector-name vector-name) (do
                  (push result
                      (dis.rw:new
                         stat           stat
                         name           vector-name
                         name-type      'vector
                         operand-number ()
                         type           'conditional-read
                         datum          datum) ) ) ) ) ) )



              ;*** Make a record for each read variable.
              ;
        (loop (for-each-stat-operand-read stat operand operand-number) (do
            (push result
                (dis.rw:new
                   stat           stat
                   name           operand
                   name-type      (name:type operand)
                   operand-number operand-number
                   type           'read
                   datum          datum) ) ) )

        result) )

;***========================================================================
;***
;*** (DIS.STAT:CREATE-WRITTEN-RW STAT DATUM TRACE-DIRECTION)
;***
;*** Returns a DIS.RW record representing the variable written by STAT.
;*** () if STAT doesn't write a variable.
;***
;***========================================================================

(defun dis.stat:create-written-rw ( stat datum trace-direction )
    (let ( (oper (stat:source stat) ) )
        (if (oper:part oper 'written)
            (dis.rw:new
```



```
               stat           stat
               name           (oper:part oper 'written)
               name-type      (name:type (oper:part oper 'written) )
               operand-number (oper:part-description oper 'written)
               type           'written
               datum          datum)
            () ) ) )



;***========================================================================
;***
;*** (OPER:LIVE-IN OPER)
;*** (OPER:LIVE-OUT OPER)
;*** (OPER:LIVE-OUT-ON-EDGE OPER DIRECTION)
;***
;***========================================================================


(defun oper:live-in ( oper )
    (let ( (stat (hash-table:get *dis.oper:stat* oper) ) )
        (assert (!== stat *hash-table.not-found*) )
        (dis.name-set:list-scalars (stat:live-in stat) ) ) )


(defun oper:live-out ( oper )
    (let ( (stat (hash-table:get *dis.oper:stat* oper) ) )
        (assert (!== stat *hash-table.not-found*) )
        (dis.name-set:list-scalars (stat:live-out stat) ) ) )


(defun oper:live-out-on-edge ( oper direction )
    (assert (memq direction '(left right) ) )
    (let ( (stat (hash-table:get *dis.oper:stat* oper) ) )
        (assert (!== stat *hash-table.not-found*) )
        (dis.name-set:list-scalars
            (name-set:intersection
                (stat:live-out stat)
                (stat:live-in
                    (caseq direction
                        (left  (car  (stat:succs stat) ) )
                        (right (cadr (stat:succs stat) ) ) ) ) ) ) ) ) )


;***========================================================================
;***
;*** (DIS.NAME-SET:LIST-SCALARS SET)
;***
;*** Returns a list of all the scalar variable names in SET.
;***
;***========================================================================

(defun dis.name-set:list-scalars ( set )
    (loop (for-each-name-set-element set name)
          (when (!== 'vector (name:type name) ) )
    (save name) ) )
```

```lisp
(include flow-analysis:flow-analysis-decls)

(declare (special
    *dt.de-calls*
    *dt.de-successes*
    *dt.conflicts*
    ) )

(defun fg.disambiguator-tool ()
    (let*( (vector-stats          (dt.collect-vector-stats) )
           (total-vector-stats    (length vector-stats) )
           (*dt.de-calls*        0)
           (*dt.de-successes*    0)
           (*dt.conflicts*       () ) )
        (msg 0 t (j (length vector-stats) 4) " vector STATs" t)

        (loop (incr i from 1 to (length vector-stats) )
              (bind stat1 (nth-elt vector-stats i) )
        (do
            (loop (incr j from (+ 1 i) to (length vector-stats) )
                  (bind stat2 (nth-elt vector-stats j) )
            (do
                (if (dt.conflicting-vector-stats? stat1 stat2) (then
                    (dt.print-conflicting-vector-stats stat1 stat2) ))))))

        (msg 0 t (j total-vector-stats 4) " vector STATs." t)
        (msg (j (- (// (* total-vector-stats (+ -1 total-vector-stats) )
                       2)
                   *dt.de-calls*)
                4)
             " pairs were trivial (different vectors, different loops)." t)
        (msg (j *dt.de-calls* 4)
             " pairs required the diophantine equation solver." t)
        (msg (j *dt.de-successes* 4)
             " of those pairs were possibly conflicting." t)
        (msg (j (length *dt.conflicts*) 4)
             " unique conflicting pairs of indices." t)
        () ) )


(defun dt.conflicting-vector-stats? ( stat1 stat2 )
    (&& (!== stat1 stat2)

        (== (stat:part stat1 'vector)
            (stat:part stat2 'vector) )

        (== (dt.stat:containing-loop stat1)
            (dt.stat:containing-loop stat2) )

        (let ( (deriv1 (stat:index-derivation stat1) )
               (deriv2 (stat:index-derivation stat2) ) )
            (&& (:= *dt.de-calls* (+ 1 *dt.de-calls*) )
                (== 'maybe (de:possibly-equal? deriv1 deriv2 stat2) )
                    ;
                    ;*** Passing in STAT2 is a crock -- it just usually
                    ;*** happens that STAT2 comes "after" STAT1.  Sigh.
                    ;*** which stat should we pass in to pick up which
                    ;*** "valid" assertions?

                (:= *dt.de-successes* (+ 1 *dt.de-successes*) ) ) ) ) )
```

```lisp
(defun dt.print-conflicting-vector-stats ( stat1 stat2 )
    (let*( (deriv1        (stat:index-derivation stat1) )
           (deriv2        (stat:index-derivation stat2) )
           (dexpr         (de:normalize-equation '(- ,deriv1 ,deriv2) ) )
           (prev-conflict (assoc# dexpr *dt.conflicts*) ) )

        (msg 0 t    (stat:number stat1) ": " (stat:source stat1)
             (t 40) (stat:number stat2) ": " (stat:source stat2) t)

        (if prev-conflict (then
            (msg "Same as "
                 (stat:number (nth-elt prev-conflict 2) ) "//"
                 (stat:number (nth-elt prev-conflict 3) ) "." t) )
        (else
            (push *dt.conflicts* '(,dexpr ,stat1 ,stat2) )
            (hprin1 (dt.de:pretty (de:normalize deriv1) ) )
            (terpri)
            (hprin1 (dt.de:pretty (de:normalize deriv2) ) )
            (terpri)
            (hprin1 '(= 0 ,(dt.de:pretty dexpr) ) ) ) )
        () ) )


(defun dt.collect-vector-stats ()
    (loop (for-each-stat stat)
          (when (stat:property? stat 'vector-reference) )
          (initial result () )
    (do
        (push result stat) )
    (result (dreverse result) ) ) )


(defun dt.stat:containing-loop ( stat )
    (loop (for-each-loop loop)
          (initial containing-loop () )
    (do
        (if (loop:bblock:member? loop (stat:bblock stat) ) (then
            (:= containing-loop loop) ) ) )
    (result containing-loop) ) )

(defun dt.de:pretty ( expr )
    (? ( (consp expr)
         (loop (for sub-expr in (cdr expr) )
               (initial result      ()
                        sub-result  () )
         (do
             (:= sub-result (dt.de:pretty sub-expr) )
             (caseq (car expr)
                 (+ (if (!= 0 sub-result) (then
                        (push result sub-result) ) ) )
                 (* (? ( (= 0 sub-result)
                         (return 0) )
                       ( (!= 1 sub-result)
                         (push result sub-result) ) ) )
                 (& (push result sub-result) ) ) )
         (result
             (? ( (! result)
                  (if (== '+ (car expr) ) 0 1) )
                ( (= 1 (length result) )
                  (car result) )
                ( t
                  '(,(car expr) .,(dreverse result) ) ) ) ) ) )
```

```
( (stat:is expr)
  (if (== 'loop-assign (stat:operator expr) )
      `(,(stat:part expr 'written) ,(stat:number expr) )
      `(,(stat:source expr) ,(stat:number expr) ) ) )

( t
  expr) ) )
```

```
;***========================================================================
;***
;*** This module sets the :DOMINATORS field of each BBLOCK to be the set
;*** of BBLOCKs that dominate that BBLOCK.  See chapter 13 of the Dragon
;*** Book for the definition of dominators and an explanation of the
;*** algorithm used to calculate them.
;***
;***========================================================================

(include flow-analysis:flow-analysis-decls)


(defun fg.set-dominators ()

    (loop (for-each-bblock bblock)
          (initial universe (bblock-set:universe) )
    (do
        (:= (bblock:dominators bblock) universe) )
    (result
        (:= (bblock:dominators *fg.entry-bblock*)
            (bblock-set:singleton *fg.entry-bblock*) ) ) )


    (loop (initial change () )
    (next change () )
    (do
        (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward) )
              (initial new-dominators () )
        (do
            (:= new-dominators
                (bblock-set:union1
                    (apply 'bblock-set:intersection
                           (for (pred-bblock in (bblock:preds bblock) )
                                (save (bblock:dominators pred-bblock) ) ) )
                    bblock) )

            (if (! (bblock-set:= new-dominators
                                 (bblock:dominators bblock) ) )
                (:= change t) )

            (:= (bblock:dominators bblock) new-dominators) ) ) )

    (while change) )
    )
```

```
;==========================================================================
;
; Find Loops
;
; This module finds the loops in the flow graph.  A loop is defined
; here differently than in the Dragon Book.  Each loop has a loop header
; that dominates all the nodes in the body of the loop.  A loop header
; is identified by an edge T -> H in the flow graph such that H
; dominates T.  A loop consists of the header H, ALL the backedges
; T -> H, and all the predecessors of each T that are dominated by
; H.
;
; (FG.FIND-LOOPS)
;    Finds all the loops in the flow graph.
;
; (FG.PRINT-LOOPS)
;    Prints out all the loops.
;
;==========================================================================

(include flow-analysis:flow-analysis-decls)


(defun fg.find-loops ()
    (fg.initialize-loops)

            ;*** Consider each edge in turn;  if the head of the edge
            ;*** dominates the tail, then the edge is a backedge of a loop.
            ;
    (loop (for-each-bblock bblock) (do
        (loop (for succ-bblock in (bblock:succs bblock) ) (do
            (if (bblock:dominates? succ-bblock bblock) (then
                (loop:create bblock succ-bblock) ) ) ) ) ) )


            ;*** For each loop, find all the BBLOCKs in the loop.
            ;*** Create the :STATS of the loop (the set of all statements
            ;*** within the loop.
    (loop (for-each-loop loop) (do
        (fl.loop:find-body loop)

        (loop (for-each-bblock-set-element (loop:bblocks loop) bblock) (do
            (loop (for-each-bblock-stat bblock stat) (do
                (:= (loop:stats loop)
                    (stat-set:union1 (loop:stats loop) stat) ) ) ) ) ) ) )


            ;*** For each loop, find all the BBLOCKs of the loop that
            ;*** are exits (having at least on successor not in the loop).
            ;
    (loop (for-each-loop loop) (do
        (loop (for-each-bblock-set-element (loop:bblocks loop) bblock) (do
            (loop (for succ-bblock in (bblock:succs bblock) ) (do
                (if (! (loop:bblock:member? loop succ-bblock) ) (then
                    (:= (loop:exits loop)
                        (bblock-set:union1 (loop:exits loop) bblock) )
                    (return () ) ) ) ) ) ) ) ) )

            ;*** Sort the loops according to containment
    (fg.sort-loops)
    )
```



```
(defun fl.loop:find-body ( loop )
    (loop (for-each-bblock-set-element (loop:back-edges loop) back-edge-bblock)
    (do
        (if (!== back-edge-bblock (loop:header loop) ) (then

            (loop (initial bblock ()
                           stack  (list back-edge-bblock) )
            (while stack)
            (do
                (pop stack bblock)
                (loop (for pred-bblock in (bblock:preds bblock) ) (do
                    (if (! (loop:bblock:member? loop pred-bblock) ) (then
                        (:= (loop:bblocks loop)
                            (bblock-set:union1 (loop:bblocks loop)
                                               pred-bblock) )
                        (push stack pred-bblock) ) ) ) ) ) ) ) ) ) ) )


(defun fg.print-loops ()
    (loop (for-each-loop loop) (do
        (msg 0 t)
        (msg "Header:      " (bblock:number (loop:header loop) ) t)
        (msg "Bblocks:     " (e (bblock-set:print (loop:bblocks    loop) ) ) t)
        (msg "Stats:       " (e (stat-set:print   (loop:stats      loop) ) ) t)
        (msg "Back edges:  " (e (bblock-set:print (loop:back-edges loop) ) ) t)
        (msg "Exits:       " (e (bblock-set:print (loop:exits      loop) ) ) t)
        (msg "Invariants: "
            (for (stat in (loop:invariants loop) )
                (save (stat:number stat) ) )
            t)
        () ) ) )
```

```
(include flow-analysis:flow-analysis-decls)
(build '(flow-analysis:dependencies) )


;***=========================================================================
;***
;*** (FG.INITIALIZE)
;***
;*** Initializes the flow-graph by forgetting all previous STATs and BBLOCKs,
;*** and prepares for the creation of a new flow-graph.
;***
;***=========================================================================

(defun fg.initialize ()
    (fg.initialize-naddr-to-flow-graph)
    (fg.initialize-stats)
    (fg.initialize-bblocks)
    (fg.initialize-temporary-name)
    (fg.initialize-dependent-functions)
    () )


;***=========================================================================
;***
;*** (FG.ANALYZE&OPTIMIZE NADDR)
;***
;***=========================================================================

(defun fg.analyze&optimize ( naddr )
    (let ( (result-naddr () ) )

        (fg.initialize)

        (fg.naddr-to-flow-graph naddr)

        (if *fg.rename-variables?* (then
            (fg.invoke-dependent-function 'fg.rename-variables) ) )

        (if *fg.move-loop-invariants?* (then
            (fg.invoke-dependent-function 'fg.move-loop-invariants) ) )

        (if *fg.remove-induction-variables?* (then
            (if *fg.eliminate-common-subexpressions?* (then
                (fg.invoke-dependent-function
                    'fg.eliminate-common-subexpressions) ) )
            (if *fg.propagate-copies?* (then
                (fg.invoke-dependent-function 'fg.propagate-copies) ) )
            (if *fg.remove-dead-code?* (then
                (let ( (*fg.remove-assertions?* () ) )
                    (fg.invoke-dependent-function 'fg.remove-dead-code) ) ) )
            (fg.invoke-dependent-function 'fg.remove-induction-variables) ) )

        (if *fg.fold-constants?* (then
            (fg.invoke-dependent-function 'fg.fold-constants) ) )

        (if *fg.eliminate-common-subexpressions?* (then
            (fg.invoke-dependent-function
                'fg.eliminate-common-subexpressions) ) )

        (if *fg.propagate-copies?* (then
            (fg.invoke-dependent-function 'fg.propagate-copies) ) )
```



```
        (if *skex.compact?* (then
            (fg.invoke-dependent-function 'fg.disambiguate)
            (if *fg.disambiguator-tool?* (then
                (fg.invoke-dependent-function 'fg.disambiguator-tool) ) )
            (if *fg.disambiguate-banks?* (then
                (fg.invoke-dependent-function 'fg.disambiguate-banks) ) ) ) )

        (if *fg.remove-assertions?* (then
            (fg.invoke-dependent-function 'fg.remove-assertions) ) )
        (if *fg.remove-dead-code?* (then
            (fg.invoke-dependent-function 'fg.remove-dead-code) ) )

        (:= result-naddr (fg.flow-graph-to-naddr) )

        (if *skex.compact?* (then
            (fg.invoke-dependent-function 'fg.create-naddr:stat-mapping)
            (fg.invoke-dependent-function 'fg.set-live-names) ) )

        result-naddr) )



;***=========================================================================
;***
;*** (FG.PRINT-FLOW-GRAPH)
;***
;*** Dumps out the current flow graph in semi-readable format.
;***
;***=========================================================================

(defun fg.print-flow-graph ( &optional bblock-fields )
    (loop (for-each-bblock bblock) (do
        (bblock:print bblock bblock-fields) ) ) )


;***=========================================================================
;***
;*** DEPENDENCIES
;***
;*** The dependencies between the different modules that crunch on the
;*** flow graph are recorded here to keep them all together for
;*** maintainability.  See DEPENDENCIES.LSP for details of
;*** def-dependencies.
;***
;***=========================================================================

(fg.initialize-dependencies)

(def-dependency fg.collect-names
    (reinitialize fg.initialize-names) )

(def-dependency fg.set-depth-first-order
    (reinitialize fg.initialize-depth-first-order) )

(def-dependency fg.set-reaching-defs
    (needs       fg.collect-names
                 fg.set-depth-first-order)
    (reinitialize fg.initialize-reaching-defs) )

(def-dependency fg.set-reaching-copies
    (needs       fg.collect-names
                 fg.set-depth-first-order) )
```

```
:f-dependency fg.set-reaching-uses
    (needs        fg.set-reaching-defs)
    (reinitialize fg.initialize-reaching-uses) )

(def-dependency fg.set-live-names
    (needs fg.collect-names
           fg.set-depth-first-order) )

(def-dependency fg.set-dominators
    (needs fg.set-depth-first-order) )

(def-dependency fg.find-loops
    (needs        fg.set-dominators)
    (reinitialize fg.initialize-loops) )

(def-dependency fg.set-loop-invariants
    (needs fg.set-reaching-defs
           fg.set-reaching-uses
           fg.find-loops) )

(def-dependency fg.move-loop-invariants
    (needs    fg.set-reaching-defs
              fg.set-reaching-uses
              fg.set-live-names
              fg.set-loop-invariants
              fg.find-loops)
    (destroys fg.set-reaching-defs
              fg.set-reaching-copies
              fg.set-live-names
              fg.set-dominators
              fg.find-loops
              fg.set-depth-first-order) )

(def-dependency fg.remove-induction-variables
    (needs    fg.set-reaching-defs
              fg.set-reaching-uses
              fg.set-reaching-copies
              fg.set-dominators
              fg.find-loops)
    (destroys fg.collect-names
              fg.set-dominators
              fg.find-loops
              fg.set-depth-first-order) )

(def-dependency fg.eliminate-common-subexpressions
    (needs        fg.set-live-names)
    (destroys     fg.collect-names)
    (reinitialize fg.initialize-cse) )

(def-dependency fg.propagate-copies
    (needs    fg.set-reaching-uses
              fg.set-reaching-copies)
    (destroys fg.collect-names) )

(def-dependency fg.fold-constants
    (needs fg.set-reaching-defs
           fg.set-reaching-uses)
    (destroys fg.collect-names) )

(def-dependency fg.remove-assertions
    (destroys fg.collect-names
              fg.set-dominators
```



```
              fg.find-loops
              fg.set-depth-first-order) )

(def-dependency fg.rename-variables
    (needs    fg.set-reaching-defs
              fg.set-reaching-uses)
    (destroys fg.collect-names) )

(def-dependency fg.remove-dead-code
    (needs    fg.set-reaching-defs
              fg.set-reaching-uses)
    (destroys fg.collect-names
              fg.set-dominators
              fg.find-loops
              fg.set-depth-first-order) )

(def-dependency fg.insert-loop-assignments
    (needs        fg.find-loops
                  fg.set-reaching-defs
                  fg.set-live-names)
    (destroys     fg.set-depth-first-order
                  fg.find-loops
                  fg.collect-names)
    (reinitialize fg.initialize-derivations) )

(def-dependency fg.create-naddr:stat-mapping
    (reinitialize fg.initialize-naddr:stat-mapping) )

(def-dependency fg.disambiguate
    (needs        fg.insert-loop-assignments    ;*** This must be first
                  fg.set-reaching-defs
                  fg.set-dominators)
    (reinitialize fg.initialize-disambiguator) )

(def-dependency fg.disambiguator-tool
    (needs fg.disambiguate
           fg.find-loops) )

(def-dependency fg.disambiguate-banks
    (needs fg.disambiguate) )
```

```
;=========================================================================
;
; Inter-module declarations for the flow analysis modules.
;
; Modules that manipulate the flow graph should INCLUDE this file at
; compile time.
;
;=========================================================================


(declare (special
    *hash-table.not-found*  ;*** From UTILITIES:HASH-TABLE.
    *bit-set.empty-set*     ;*** From UTILITIES:BIT-SET.

    *number-of-banks*       ;*** The number of memory banks we are
                            ;*** compiling for.

    *fg.total-stats*        ;*** Current number of STATs
    *fg.number:stat*        ;*** Array for mapping STAT numbers onto
                            ;*** STATs.

    *fg.total-bblocks*      ;*** Current number of BBLOCKs
    *fg.number:bblock*      ;*** Array for mapping BBLOCK numbers onto
                            ;*** BBLOCKs.
    *fg.entry-bblock*       ;*** Entry block (one with no predecessors)

    *fg.empty-stat-set*     ;*** The empty STAT-SET.

    *fg.number-of-reaching-iterations*
                            ;*** # of iterations used for calculating
                            ;*** reaching defs

    *fg.name:name-descriptor*
                            ;*** A hash table mapping variable names onto
                            ;*** NAME-DESCRIPTORS.
    *fg.number:name*        ;*** An array mapping numbers to names.
    *fg.total-names*        ;*** Total number of names.
    *fg.all-vector-names*   ;*** List of all array names.
    *fg.empty-name-set*     ;*** The empty NAME-SET.

    *fg.all-dependencies*   ;*** List of all dependency descriptors.

    *fg.all-loops*          ;*** List of all loops.

    *fg.rename-variables?*
    *fg.eliminate-common-subexpressions?*
    *fg.move-loop-invariants?*
    *fg.remove-induction-variables?*
    *fg.fold-constants?*
    *fg.propagate-copies?*
    *fg.remove-dead-code?*
    *fg.remove-assertions?*
    *fg.disambiguator-tool?*
    *fg.disambiguate-banks?*
    *fg.show-unknown-bank-references?*
                            ;*** Options defined in FLOW-ANALYSIS-OPTIONS

    *skex.compact?*
    ) )


(declare
    (lexpr vector-map:initialize)
```

1

```
    (lexpr vector-map:add-element)
    (lexpr bblock:print)
    (lexpr fg.print-flow-graph) )


(eval-when (compile)
    (build '(
        interpreter:naddr
        utilities:bit-set
        utilities:sharp-sharp

        flow-analysis:stat
        flow-analysis:stat-set
        flow-analysis:bblock
        flow-analysis:bblock-set
        flow-analysis:name
        flow-analysis:name-set
        flow-analysis:loop
        ) ) )
```

2

```
;===========================================================================
;
; FLOW ANALYSIS OPTIONS
;
; This module contains the definitions of options dealing with flow
; analysis.
;
;===========================================================================

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


(def-option *fg.rename-variables?* t flow-analysis: "
If T then variables are renamed wherever possible.
")


(def-option *fg.eliminate-common-subexpressions?* t flow-analysis: "
If T then common subexpression elimination is done on NADDR basic blocks
during optimization.
")


(def-option *fg.move-loop-invariants?* t flow-analysis: "
If T then invariants are moved out of loops during NADDR optimization.
")


(def-option *fg.remove-induction-variables?* t flow-analysis: "
If T then induction variables are eliminated and simplified.
")


(def-option *fg.fold-constants?* t flow-analysis: "
If T then constant folding is performed.
")


(def-option *fg.propagate-copies?* t flow-analysis: "
If T then copy propagation is performed.
")


(def-option *fg.remove-dead-code?* t flow-analysis: "
If T then unreachable or useless code is removed from the flow graph.
")


(def-option *fg.remove-assertions?* t flow-analysis: "
If T then assertions are removed during dead code removal.
")


(def-option *fg.disambiguator-tool?* () flow-analysis: "
If T then the disambiguator tool is invoked, printing out all possible
vector conflicts in the program.
")


(def-option *fg.disambiguate-banks?* () flow-analysis: "
If T then the bank disambiguator is invoked, modifying vector references to
contain the bank they refer to.
```



```
")


(def-option *fg.show-unknown-bank-references?* () flow-analysis: "
If T then the bank disambiguator will display all vector references that
have unknown banks.
")
```

```
;***==========================================================================
;***
;***  This module converts the current flow graph back into NADDR.  It
;***  is primarily useful for testing.
;***
;***==========================================================================

(include flow-analysis:flow-analysis-decls)

;***==========================================================================
;***
;*** (FG.FLOW-GRAPH-TO-NADDR)
;***
;*** Returns the list of NADDR corresponding to the current flow-graph.
;***
;*** The conversion is done in a single pass.  Each basic block has a LABEL
;*** generated of the form L<number> where <number> is the number of the
;*** block.  This lets us convert edges in the flow graph into GOTOs easily.
;*** A GOTO is never generated for a block that "falls through" to another
;*** block.
;***
;*** The :SOURCE field of each STAT is replaced by the new source operation
;*** generated for it.  At present, only conditional jumps generate new
;*** source (because the labels have changed).
;***
;***==========================================================================


(defun fg.flow-graph-to-naddr ()

(let ( (naddr              () ) )

    (loop (initial prev-succ-bblock () )
    (for-each-bblock bblock)
    (do
        (if (&& prev-succ-bblock
                (!== prev-succ-bblock bblock) )
        (then
            (push naddr '(goto ,(fgtn.bblock:label prev-succ-bblock) ) ) ) )

        (if (bblock:preds bblock)
            (push naddr '(label ,(fgtn.bblock:label bblock) ) ) )

        (loop (for-each-bblock-stat bblock stat)
              (initial new-source () )
        (do
            (:= new-source
                (caseq (stat:group stat)
                    (if-then-else
                        '(,(stat:operator stat)
                          ,(stat:part stat 'read1)
                          ,(stat:part stat 'read2)
                          ,(stat:part stat 'probability)
                          ,(fgtn.bblock:label (car (bblock:succs bblock) ) )
                          ,(fgtn.bblock:label (cadr (bblock:succs bblock)))))
                    (cond-jump
                        '(,(stat:operator stat)
                          ,(stat:part stat 'read1)
                          ,(stat:part stat 'probability)
                          ,(fgtn.bblock:label (car (bblock:succs bblock) ) )
                          ,(fgtn.bblock:label (cadr (bblock:succs bblock) ))))
```

```
                    (t
                        (stat:source stat) ) ) )
            (push naddr new-source)
            (:= (stat:source stat) new-source) ) ) )

    (next prev-succ-bblock
          (if (>= 1 (length (bblock:succs bblock) ) )
              (car (bblock:succs bblock) )
              () ) )
    (result
        (if prev-succ-bblock (then
            (push naddr
                  '(goto ,(fgtn.bblock:label prev-succ-bblock) ) ) ) )

        (dreverse naddr) ) ) ) )


(defun fgtn.bblock:label ( bblock )
    (atomconcat 'l (bblock:number bblock) ) )
```

```
;==============================================================================
;
; INDUCTION VARIABLE REMOVAL
;
; This module implements the induction variable removal algorithm of the
; Dragon book with some minor modifications.
;
; (FA.REMOVE-INDUCTION-VARIABLES)
;     Simplifies induction variables, munging the whole program.
;
; Here is a summary of the algorithm.  Details of the implementation are
; provided in the procedure comments below.
;
; A primary induction variable of a loop is a variable whose only
; assignments within the loop are of the form:
;
;     I := I +/- C
;
; where C is a loop invariant.  Each such assignment to I in the loop
; is called a primary induction statement.  This module assumes a name
; is loop invariant iff there are no assignments to it within the loop.
;
; A secondary induction variable is a variable K that is assigned exactly
; once within the loop by a statement of one of the forms:
;
;     K := C */+/- J       K := J */+/- C       K := -J
;
; where J is either a primary induction variable or a secondary induction
; variable.  If J is secondary, there are more requirements.  Let I be
; primary variable of J.  There can be no assignment to I between the
; single assignment to J and the single assignment to K, and the definition
; of J in the loop must be the only reaching definition of J reaching
; the assignment to K.
;
; Each secondary variable is expressed as a linear function of its primary
; variable:
;
;     K := A * I + B
;
; All the secondary induction statements of a primary with the same linear
; function are grouped together.
;
; A secondary or primary induction variable is "useless" if its only uses
; are for calculating other induction variables in the same family or
; within comparisons and conditional jumps.
;
; The program is rewritten as follows:  Each group of secondary induction
; statements with the same linear function are assigned a new unique name.
; Each assignment to a secondary i.v.  K is replaced by:
;
;     K := K'
;
; where K' is the new name.  Then at the end of the loop header, each
; new secondary variable is initialized:
;
;     K' := A * I + B
;
; Then after each assignment to the primary induction variable within
; the loop, all the new secondary variables are stepped in parallel.  If
; the assignment is of the form:
;
;     I := I + C
;
```

```
; the secondary assignments added right after it have the form:
;
;     K' := K' + C * B
;
; Then conditional jumps and comparisons are rewritten to use the
; "simplest" non-useless secondary induction variable possible; this will
; possibly allow us to delete the original secondary induction variable.
; Suppose there is a comparsion:
;
;     J RELOP X
;
; where J is either a primary induction variable or else a secondary i.v.
; with linear function:
;
;     J = AJ * I + BJ
;
; We look for another useful secondary i.v. K' that has the form:
;
;     K' = AK * I + BK  where  AK = R * AJ  for some constant R.
;
; If we find such a K', we can rewrite the conditional test to be:
;
;     TEMP := R * X - R * BJ + BK
;     K' RELOP TEMP
;
; The code for TEMP can be placed in the loop header if X is a loop
; constant.  If there are many choices for K', we favor ones that have
; R = 1 or BK = 0.
;
; After all this rewriting, many of the secondary and primary induction
; variables may now be truly dead -- dead code removal will eliminate
; them entirely from the program.
;
; When writing the assignments to the new secondary i.v.s and when
; rewriting comparisions, we rely on the fact that constant folding/
; simplification will clean up the code considerably later on.
;
;==============================================================================


;***===========================================================================
;***
;*** An IV-FAMILY represents one primary induction variable and its family
;*** of secondary induction variables.
;***
;***=========================
                          ;***
(def-struct iv-family     ;***
    name                  ;*** Name of the primary induction variable.
    primary-stats         ;*** STAT-SET of primary induction statements.
    secondary-stats       ;*** STAT-SET of secondary induction statements.
    (secondary-ivs        ;*** List of SECONDARY-IV records representing
        () suppress)      ;*** the secondary induction variables of this family.
    )                     ;***
;***=========================
;***
;***
;*** A SECONDARY-IV represents all the secondary induction variables that
;*** are the same function of the primary induction variable.
;***
;***=========================
                          ;***
(def-struct secondary-iv;***
```

```
    stats               ;*** STAT-SET of secondary induction STATs all with
                        ;*** the same linear function of the primary variable.
    name                ;*** The new variable name generated for these
                        ;*** induction STATs.
                        ;***
    original-names      ;*** List of names of original secondary i.v.s.
                        ;***
    (iv-family          ;*** The induction family to which this belongs.
        () suppress)    ;***
    a                   ;*** Constants of the linear function for these
    b                   ;*** names, of the form AI + B for  I the primary
                        ;*** induction variable.  Constants are represented
                        ;*** as normalized diophantine expressions.
                        ;***
    a-address           ;*** The loop-invariant variable names holding these
    b-address           ;*** constants' values, or else the numeric value
                        ;*** of the constant if known.
                        ;***
    useful?             ;*** True if any of the STATs forming this secondary
                        ;*** induction variable are used for anything other
                        ;*** than computing other induction variables.
    )                   ;***
;***========================
;***==================================================================

(eval-when (compile load)
    (include flow-analysis:flow-analysis-decls) )

(defvar *ivr.debug?* () )   ;*** If T, then dump out debugging info.


(declare (special
    *ivr.loop*                  ;*** Current loop being optimized.
    *ivr.iv-families*           ;*** List of all IV-FAMILYs for current loop.

    *ivr.name:#defs*            ;*** Hash tables used in our local loop
    *ivr.name:defining-stats*   ;*** flow analysis.
    *ivr.name:using-stats*      ;***

    *ivr.name:iv-family*        ;*** Hash tables mapping i.v. names onto
    *ivr.name:secondary-iv*     ;*** IV-FAMILYs and SECONDARY-IVs.
    ) )


(defun fa.remove-induction-variables ()

    (loop (for-each-loop loop)
          (bind *ivr.loop*                  loop
                *ivr.iv-families*           ()
                *ivr.name:#defs*            ()
                *ivr.name:defining-stats*   ()
                *ivr.name:using-stats*      ()
                *ivr.name:iv-family*        ()
                *ivr.name:secondary-iv*     () )
    (do
        (ivr.analyze-uses&defs)
        (ivr.collect-primary-ivs)
        (ivr.collect-secondary-ivs)

        (if *ivr.debug?* (then
            (msg 0 t "----------" t)
            (ivr.print-iv-families) ) )
```

```
        (loop (for iv-family in *ivr.iv-families*) (do
            (ivr.iv-family:simplify-induction-variables iv-family) ) )

        (if *ivr.debug?* (then
            (msg 0 t "Loop: " t)
            (bblock:print (loop:pre-header loop) )
            (loop (for-each-bblock-set-element (loop:bblocks loop) bblock) (do
                (bblock:print bblock) ) ) ) ) ) )
    () )


(defun ivr.print-iv-families ()

    (msg 0 t "Loop: " t)
    (loop (for-each-bblock-set-element (loop:bblocks *ivr.loop*) bblock) (do
        (bblock:print bblock) ) )

    (loop (for iv-family in *ivr.iv-families*) (do
        (msg 0 t
            "Primary iv:    " (iv-family:name iv-family)
            (c (if (ivr.iv-family:useful? iv-family) " useful" "") )
            t
            "Primary stats: "
            (h (loop (for-each-stat-set-element
                        (iv-family:primary-stats iv-family)
                        stat)
                  (save stat) ) )
            t)

        (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) )
        (do
            (msg 0 t
                "Secondary name:  " (secondary-iv:name secondary-iv)
                (c (if (secondary-iv:useful? secondary-iv) " useful" "") )
                t
                "Secondary names: " (secondary-iv:original-names
                                      secondary-iv)
                t
                "Secondary stats: "
                (h (loop (for-each-stat-set-element
                            (secondary-iv:stats secondary-iv)
                            stat)
                      (save stat) ) )
                t
                "Secondary a = " (secondary-iv:a secondary-iv) t
                "            b = " (secondary-iv:b secondary-iv) t) ) ) ) )

    () )


;***=====================================================================
;***
;*** LOCAL LOOP FLOW ANALYSES
;***
;*** Optimization of nested loops requires incremental flow analysis,
;*** since optimizing an outer loop affects the analysis of an inner one
;*** and vice versa.  So we do our own simple, special-case analysis at
;*** the beginning of optimizing each loop.   Groan.
;***
;*** (IVR.ANALYZE-USES&DEFS)
;***     Performs the analysis accessed by the two functions below on
;***     the current loop.  Notice that whe recalculate LOOP:STATS, since
;***     the optimization of an outer loop may have added new stats to
```

```
;***     the loop.
;***
;*** (IVR.NAME:LOOP-DEF-COUNT NAME)
;***     Returns the number of defs of NAME in the current loop.
;***
;*** (IVR.NAME:LOOP-DEFINING-STATS NAME)
;***     Returns the set of STATs that define NAME in the current loop.
;***
;*** (IVR.NAME:LOOP-USING-STATS NAME)
;***     Returns the set of STATs that use NAME in the current loop.
;***
;*** (IVR.NAME:CONSTANT? NAME)
;***     True if NAME is a constant within the loop (either a number or
;***     else not defined in the loop -- we rely on the fact that loop
;***     invariants have already been moved out.)
;***
;*** (IVR.NAME:IV-FAMILY NAME)
;***     If NAME has been identified as a primary induction variable,
;***     returns the corresponding IV-FAMILY;  otherwise, returns ().
;***
;*** (IVR.NAME:SECONDARY-IV NAME)
;***     If NAME has been identified as a secondary induction variable,
;***     returns the corresponding SECONDARY-IV;  otherwise, returns ().
;***
;*** (IVR.NAME:IV-FAMILY&CONSTANTS NAME)
;***     NAME must be an induction variable.  Returns a list of the form:
;***
;***         (IV-FAMILY A B)
;***
;***     where IV-FAMILY is the primary family of NAME, and A and B are
;***     its linear function constants (1 and 0 if NAME is a primariy
;***     iv).
;***
;***=========================================================================

(defun ivr.analyze-uses&defs ()

    (:= *ivr.name:#defs*
        (hash-table:create () () 0) )
    (:= *ivr.name:defining-stats*
        (hash-table:create () () *stat-set.empty-set*) )
    (:= *ivr.name:using-stats*
        (hash-table:create () () *stat-set.empty-set*) )
    (:= *ivr.name:iv-family*
        (hash-table:create () () () ) )
    (:= *ivr.name:secondary-iv*
        (hash-table:create () () () ) )

    (:= (loop:stats *ivr.loop*) *stat-set.empty-set*)

    (loop (for-each-bblock-set-element (loop:bblocks *ivr.loop*) bblock) (do
        (loop (for-each-bblock-stat bblock stat)
              (when (!== 'loop-assign (stat:operator stat) ) )
        (do

            (:= (loop:stats *ivr.loop*) (stat-set:union1 &&& stat) )

            (if-let ( (written (stat:part stat 'written) ) ) (then
                (let ( (count     (hash-table:get *ivr.name:#defs* written) )
                       (def-stats (hash-table:get *ivr.name:defining-stats*
                                                  written) ) )
                    (hash-table:put *ivr.name:defining-stats* written
                                    (stat-set:union1 def-stats stat) )
```



```
                    (hash-table:put *ivr.name:#defs* written
                                    (+ 1 count) ) ) ) ) )

            (loop (for-each-stat-operand-read stat operand)
                  (bind using-stats (hash-table:get *ivr.name:using-stats*
                                                    operand) )
            (do
                (hash-table:put *ivr.name:using-stats* operand
                                (stat-set:union1 using-stats stat)))))))
    () )


(defun ivr.name:loop-def-count ( name )
    (hash-table:get *ivr.name:#defs* name) )


(defun ivr.name:loop-defining-stats ( name )
    (hash-table:get *ivr.name:defining-stats* name) )


(defun ivr.name:loop-using-stats ( name )
    (hash-table:get *ivr.name:using-stats* name) )


(defun ivr.name:constant? ( name )
    (|| (numberp name)
        (== 0 (hash-table:get *ivr.name:#defs* name) ) ) )


(defun ivr.name:iv-family ( name )
    (hash-table:get *ivr.name:iv-family* name) )


(defun ivr.name:secondary-iv ( name )
    (hash-table:get *ivr.name:secondary-iv* name) )


(defun ivr.name:iv-family&constants ( name )
    (if-let ( (iv-family (hash-table:get *ivr.name:iv-family* name) ) ) (then
        '(,iv-family (& (+ (* 1) ) )
                     (& (+ (* 0) ) ) ) ) )

    (else (if-let ( (secondary-iv (hash-table:get *ivr.name:secondary-iv*
                                                  name) ) )
    (then
        '(,(secondary-iv:iv-family secondary-iv)
          ,(secondary-iv:a         secondary-iv)
          ,(secondary-iv:b         secondary-iv) ) )

    (else
        (error (list name "Name is not an induction variable.") ) ) ) ) ) )


;***=========================================================================
;***
;*** (IVR.COLLECT-PRIMARY-IVS)
;***
;*** Discovers all the primary induction variables and creates an IV-FAMILY
;*** for each one.
;***
;***=========================================================================

(defun ivr.collect-primary-ivs ()
```

```
    (:= *ivr.iv-families*
        (loop (for var in (ivr.primary-induction-variables) )
              (bind iv-family
                    (iv-family:new
                         name          var
                         primary-stats (ivr.name:loop-defining-stats var)))
        (save
            (hash-table:put *ivr.name:iv-family* var iv-family)
            iv-family) ) )
    () )


;***=====================================================================
;***
;*** (IVR.PRIMARY-INDUCTION-VARIABLES)
;***
;*** Returns a list of the primary induction variables of the current
;*** loop.
;***
;***=====================================================================

(defun ivr.primary-induction-variables ()
    (loop (for-each-stat-set-element (loop:stats *ivr.loop*) stat)
          (initial candidate-vars  ()
                   candidate-stats () )
    (do
        (if (ivr.stat:possible-primary-induction? stat) (then
            (:= candidate-vars  (unionq1 &&& (stat:part stat 'written) ) )
            (:= candidate-stats (stat-set:union1 &&& stat) ) ) ) )
    (result
        (loop (for var in candidate-vars)
              (when (stat-set:contains? candidate-stats
                                        (ivr.name:loop-defining-stats var) ) )
        (save var) ) ) ) )


;***=====================================================================
;***
;*** (IVR.STAT:POSSIBLE-PRIMARY-INDUCTION? STAT)
;***
;*** Returns true iff STAT is a primary induction STAT; that is, STAT
;*** is of the form I := I +/- C, where C is a constant or loop invariant.
;***
;***=====================================================================

(defun ivr.stat:possible-primary-induction? ( stat )
    (assert (stat:is stat) )

    (let ( (written  (stat:part stat 'written) )
           (operator (stat:operator stat) ) )
        (&& (memq operator '(iadd isub) )
            (let ( (read1 (stat:part stat 'read1) )
                   (read2 (stat:part stat 'read2) ) )
                (|| (&& (== written read1)
                        (ivr.name:constant? read2) )
                    (&& (== 'iadd operator)
                        (== written read2)
                        (ivr.name:constant? read1) ) ) ) ) ) ) )


;***=====================================================================
;***
```



```
;*** (IVR.COLLECT-SECONDARY-IVS)
;***
;*** Finds all the secondary induction variables, creating SECONDARY-IVs
;*** for them.
;***
;***=====================================================================

(defun ivr.collect-secondary-ivs ()
 add in a secondary iv for each primary.
        ;*** Scan each of the statements of each of the blocks of the
        ;*** loop, looking for possible secondary ivs.
        ;***
    (loop (for-each-bblock-set-element (loop:bblocks *ivr.loop*) bblock) (do
        (loop (for-each-bblock-stat bblock stat) (do
            (ivr.stat:secondary-induction? stat) ) ) ) )

        ;*** Reverse the lists of SECONDARY-IVs, for prettiness.
        ;***
    (loop (for iv-family in *ivr.iv-families*) (do
        (:= (iv-family:secondary-ivs iv-family) (dreverse &&&) ) ) )

        ;*** Mark the "useful" SECONDARY-IVs.
        ;***
    (loop (for iv-family in *ivr.iv-families*) (do
        (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) ) (do
            (ivr.secondary-iv:mark-if-useful secondary-iv) ) ) ) )
    () )


;***=====================================================================
;***
;*** (IVR.STAT:SECONDARY-INDUCTION? STAT)
;***
;*** Checks to see if STAT qualifies as a secondary induction stat of
;*** some IV-FAMILY; if so, a the secondary i.v.  is added to the family
;*** and true is returned.
;***
;*** To be secondary i.v., STAT must have the form:
;***
;***    STAT: K := J +/-/* C   or   K := - J
;***
;*** where J is an already-discovered primary or secondary i.v.
;***
;*** If J is a secondary i.v., then only one definition of J may reach
;*** K and no assignment of the primary variable, I, of J occurs between
;*** the assignment to J and STAT.
;***
;*** Because an complicated analysis is needed to check this requirement
;*** in general (using reaching copies, which would need to be
;*** incrementally computed),  we use more restricted criteria suggested
;*** by the Dragon Book:
;***
;***    If the assignments to J and K are in the same block, then the
;***    assignment to J must occur first and there must be no assignment
;***    to I in between.
;***
;***    If the assignments to J and K are in different blocks, then J's
;***    block must dominate K's block, and all the assignments to I must
;***    be in back-edge blocks of the loop (blocks whose successor is
;***    the loop header); there must be no assignment to K in those
;***    back-edge blocks after the assignment to I.
;***
;*** Both of these conditions satisfy the more general requirement.  The
```

```
;*** first condition is suggested by the Dragon Book.  The second condition
;*** adds a little more generality needed for unrolling loops and "folding"
;*** the induction variable.
;***
;***=====================================================================

(defun ivr.stat:secondary-induction? ( stat ) (prog ()
(assert (stat:is stat) )
(let*( (k            () )
       (j            () )
       (c            () )
       (read1        () )
       (read2        () ) )

    (if (! (memq (stat:operator stat) '(iadd imul isub ineg) ) ) (then
        (return () ) ) )

    (desetq (read1 read2) (stat:part stat 'read) )
    (?( (ivr.name:constant? read1)
        (:= c read1)
        (:= j read2) )
      ( (ivr.name:constant? read2)
        (:= c read2)
        (:= j read1) )
      ( t
        (return () ) ) )

    (:= k (stat:part stat 'written) )
    (if (|| (== j k)
            (!== 1 (ivr.name:loop-def-count k) ) )
    (then
        (return () ) ) )

    (if (! (ivr.stat:j:k:secondary-induction? stat j k) ) (then
        (return () ) ) )

    (ivr.stat:add-secondary-iv stat k j c)
    (return t) ) ) )


;***=====================================================================
;***
;*** (IVR.STAT:J:K:SECONDARY-INDUCTION? K-STAT J K)
;***
;*** K-STAT is a stat of the form:
;***
;***    K := J +/- C   or   K := - J
;***
;*** This function checks the following conditions:
;***
;***    J is a primary iv;  OR
;***
;***    J is a secondary iv, with these requirements:
;***
;***        If the assignments to J and K are in the same block, then
;***        the assignment to J must occur first and there must be no
;***        assignment to I in between.
;***
;***        If the assignments to J and K are in different blocks, then
;***        J's block must dominate K's block, and all the assignments
;***        to I must be in back-edge blocks of the loop (blocks whose
;***        successor is the loop header); there must be no assignment
;***        to K in those back-edge blocks after the assignment to I.
```



```
;***
;*** These requirements identify a subset of the following general
;*** condition required for K-STAT to be a secondary iv:
;***
;***    J is primary, or if J is a secondary, then only one definition
;***    of J may reach K and no assignment of the primary variable, I,
;***    of J occurs between the assignment to J and STAT.
;***
;*** The more general condition can be checked using reaching copies,
;*** but it needs incremental flow analysis (ugh).  So we use these
;*** restricted conditions which are easier to check and will get most
;*** cases.
;***
;***=====================================================================

(defun ivr.stat:j:k:secondary-induction? ( k-stat j k )

(let ( (i              () )
       (secondary-iv   () )
       (j-stat         () ) )

   (?( (ivr.name:iv-family j)
       t)

     ( (:= secondary-iv (ivr.name:secondary-iv j) )
       (:= j-stat (stat-set:choose (ivr.name:loop-defining-stats j) ) )
       (:= i (iv-family:name (secondary-iv:iv-family secondary-iv) ) )

       (if (== (stat:bblock j-stat) (stat:bblock k-stat) ) (then
           (ivr.bblock:no-assignments? j-stat k-stat i) )

       (else
           (&& (stat:dominates? j-stat k-stat)
               (loop (for-each-stat-set-element
                        (ivr.name:loop-defining-stats i)
                        i-stat)
                     (bind i-bblock (stat:bblock i-stat) )
               (do
                   (if (! (bblock-set:member? (loop:back-edges *ivr.loop*)
                                              i-bblock) )
                   (then
                       (return () ) ) )
                   (if (! (ivr.bblock:no-assignments?
                               i-stat
                               (bblock:last-stat i-bblock)
                               k) )
                   (then
                       (return () ) ) ) )
               (result t) ) ) ) ) )

     ( t
       () ) ) ) )


;***=====================================================================
;***
;*** (IVR.BBLOCK:NO-ASSIGNMENTS? STAT1 STAT2 I)
;***
;*** Returns true if STAT2 is in the same block as STAT1, STAT1 comes after
;*** STAT2 in the block, and there are no assignments to variable I in between
;*** STAT1 and STAT2.
;***
;***=====================================================================
```

```
(defun ivr.bblock:no-assignments? ( stat1 stat2 i )
    (loop (initial stat (stat:succ stat1) ) (do
        (if (! stat) (then
            (return () ) ) )
        (if (== stat stat2) (then
            (return t) ) )
        (if (== i (stat:part stat 'written) ) (then
            (return () ) ) )
        (:= stat (stat:succ stat) ) ) ) )


;***=====================================================================
;***
;*** (IVR.STAT:ADD-SECONDARY-IV STAT K J C)
;***
;*** A secondary induction variable is created corresponding to STAT
;*** which is of one of the forms:
;***
;***      K := J + C    K := C + J
;***      K := J - C    K := C - J
;***      K := - J
;***      K := J * C    K := C * J
;***
;*** where J is an already-known induction variable whose form is:
;***
;***      J = JA * I + JB   (for primary induction variable I)
;***
;*** JA = 1 and JB = 0 if J is a primary induction variable.
;***
;*** The linear function constants of the new secondary induction variable
;*** K are calculated by substituting in the constants of J.
;***
;*** Note that if there is already a secondary induction variable with
;*** the same constants, we just add STAT to the set of STATs of that
;*** secondary variable.
;***
;***=====================================================================

(defun ivr.stat:add-secondary-iv ( stat k j c )
    (assert (stat:is stat) )

    (let*( ( (iv-family ja jb) (ivr.name:iv-family&constants j) )
           (a                  () )
           (b                  () )
           (secondary-iv       () ) )

        ;*** Calculate the linear function constants of STAT by
        ;*** substituting in for the constants of the induction variable.
        ;***
      (caseq (stat:operator stat)

        (iadd   ;*** K := J + C  ==>  JA * I + (JB + C)
                ;***
            (:= a ja)
            (:= b '(+ ,jb ,c) ) )

        (isub   ;*** K := J - C  ==>   JA * I + (JB  - C)
                ;*** K := C - J  ==>  -JA * I + (-JB + C)
                ;***
            (if (== c (stat:part stat 'read2) ) (then
                (:= a ja)
                (:= b '(- ,jb ,c) ) )
             (else
                (:= a '(* -1 ,ja) )
                (:= b '(- ,c ,jb) ) ) ) )

        (ineq   ;*** K := -J ==>  -JA * I - JB
                ;***
            (:= a '(* -1 ,ja) )
            (:= b '(* -1 ,jb ) ) )

        (imul   ;*** K := C * J ==>  C * JA * I + C * JB
                ;***
            (:= a '(* ,c ,ja) )
            (:= b '(* ,c ,jb) ) )

        (t
            (error (list stat "Unexpected operator.") ) ) )

        ;*** Add STAT to the SECONDARY-IV that already has the same
        ;*** constants, or else create a new SECONDARY-IV for STAT
        ;*** if no previous secondary iv has the same constants.
        ;***
    (:= a (de:normalize a) )
    (:= b (de:normalize b) )

    (:= secondary-iv (ivr.iv-family:a:b:secondary-iv iv-family a b) )

    (push (secondary-iv:original-names secondary-iv) k)
    (hash-table:put *ivr.name:secondary-iv* k secondary-iv)
    (if (! (secondary-iv:name secondary-iv) ) (then
        (:= (secondary-iv:name secondary-iv)
            (fa.temporary-name (stat:part stat 'written) ) ) ) )

    (:= (secondary-iv:stats secondary-iv)
        (stat-set:union1 &&& stat) )
    (:= (iv-family:secondary-stats iv-family)
        (stat-set:union1 &&& stat) )
    () ) )


;***=====================================================================
;***
;*** (IVR.IV-FAMILY:A:B:SECONDARY-IV IV-FAMILY A B)
;***
;*** Searches IV-FAMILY for a secondary induction variable record with
;*** constants A and B.   If one is found, it is returned.   Otherwise
;*** a new one is created, added to the IV-FAMILY and returned.
;***
;***=====================================================================

(defun ivr.iv-family:a:b:secondary-iv ( iv-family a b )
    (assert (iv-family:is iv-family) )

    (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) )
        (when (&& (= a (secondary-iv:a secondary-iv) )
                  (= b (secondary-iv:b secondary-iv) ) ) )
    (do
        (return secondary-iv) )
    (result
        (let ( (secondary-iv (secondary-iv:new iv-family iv-family
                                               stats     *stat-set.empty-set*
                                               a         a
                                               b         b) ) )
            (push (iv-family:secondary-ivs iv-family) secondary-iv)
```

```
          secondary-iv) ) ) )

;***===========================================================================
;***
;*** (IVR.SECONDARY-IV:MARK-IF-USEFUL SECONDARY-IV)
;***
;*** Sets the :USEFUL? field of SECONDARY-IV.  A secondary variable is
;*** useful if at least one of the secondary induction STATs is used for
;*** something other than calculating other secondary induction variables
;*** in the same family or conditional jumps.
;***
;***===========================================================================

(defun ivr.secondary-iv:mark-if-useful ( secondary-iv )
    (assert (secondary-iv:is secondary-iv) )

    (if (for-some (var in (secondary-iv:original-names secondary-iv) )
            (ivr.iv-family:induction-var:useful?
                (secondary-iv:iv-family secondary-iv)
                var) )
    (then
        (:= (secondary-iv:useful? secondary-iv) t) ) )

    () )


;***===========================================================================
;***
;*** (IVR.IV-FAMILY:USEFUL? IV-FAMILY)
;***
;*** Returns true if the primary induction variable is used for something
;*** other than conditional jumps, asserts, or computing induction
;*** variables within the family.
;***
;***===========================================================================
```

X

flush this?

```
(defun ivr.iv-family:useful? ( iv-family )
    (assert (iv-family:is iv-family) )

    (ivr.iv-family:induction-var:useful?
        iv-family
        (iv-family:name iv-family) ) )


;***===========================================================================
;***
;*** (IVR.IV-FAMILY:INDUCTION-VAR:USEFUL? IV-FAMILY VAR)
;***
;*** Returns true if VAR is used for something other than conditional
;*** jumps, asserts, or computing other induction variables within the
;*** family.
;***
;***===========================================================================

(defun ivr.iv-family:induction-var:useful? ( iv-family var )
    (assert (iv-family:is iv-family) )

    (let ( (iv-stats (stat-set:union (iv-family:primary-stats   iv-family)
                                     (iv-family:secondary-stats iv-family))))

        (loop (for-each-stat-set-element
                    (stat-set:difference (ivr.name:loop-using-stats var)
```



```
                                            iv-stats)
                stat)
            (when (! (memq (stat:group stat) '(if-compare assert))))
        (do
            (return t) )
        (result () ) ) ) ) )


;***===========================================================================
;***
;*** (IVR.IV-FAMILY:SIMPLIFY-INDUCTION-VARIABLES IV-FAMILY)
;***
;*** Replaces all assignments to secondary induction variables of IV-FAMILY
;*** in the current loop by assignments rewritten in terms of their linear
;*** functions of the primary variable, decoupling secondary variables
;*** from the primary ones.
;***
;*** Then all conditional-jump uses of the primary variable and secondary
;*** variables are rewritten to use the simplest secondary induction
;*** variable that is being used for something other than conditional
;*** jumps.  This will leave the primary and/or some of the secondary
;*** induction variables useless, to be deleted by dead code removal.
;***
;***===========================================================================

(defun ivr.iv-family:simplify-induction-variables ( iv-family )
    (assert (iv-family:is iv-family) )

        ;*** Generate code in the pre-header for the linear constants
        ;*** A and B of each secondary variable if needed, assigning
        ;*** :A-ADDRESS and :B-ADDRESS.
        ;***
    (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) ) (do
        (ivr.secondary-iv:generate-constant-code secondary-iv) ) )
```

yes, do this for the primary-iv.

```
        ;*** Generate pseudo-ops in the loop header that relate the
        ;*** secondary variables to the primary variable, for use by the
        ;*** assertion facility.
        ;***
    (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) ) (do
        (ivr.secondary-iv:generate-loop-assign secondary-iv) ) )
```

when not primary

```
        ;*** Rewrite each conditional jump use of the primary iv, provided
        ;*** the primary iv is used only for conditional jumps and
        ;*** secondary ivs.
        ;***
    (if (! (ivr.iv-family:useful? iv-family) ) (then
    (loop (for-each-stat-set-element (iv-family:primary-stats iv-family)
                                     induction-stat)
    (do
        (loop (for-each-stat-set-element
                    (ivr.name:loop-using-stats (iv-family:name iv-family) )
                    use-stat)
                (when (== 'if-compare (stat:group use-stat) ) )
        (do
            (ivr.iv-family:induction-var:jump-stat:simplify
                iv-family (iv-family:name iv-family) use-stat) ) ) ) ) ) )
```

flush

```
        ;*** Rewrite each conditional jump use of a useless secondary iv.
        ;***
    (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) )
        (when (! (secondary-iv:useful? secondary-iv) ) )
    (do
```

```
       (loop (for var in (secondary-iv:original-names secondary-iv) ) (do
           (loop (for-each-stat-set-element
                       (ivr.name:loop-using-stats var)
                       use-stat)
                 (when (== 'if-compare (stat:group use-stat) ) )
           (do
               (ivr.iv-family:induction-var:jump-stat:simplify
                   iv-family var use-stat) ) ) ) ) ) )

       ;*** Generate code for each of the secondary ivs that decouples
       ;*** them from the primary iv.
       ;***
   (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) ) (do
       (ivr.secondary-iv:simplify secondary-iv) ) )

   () ) )


;***=========================================================================
;***
;*** (IVR.SECONDARY-IV:GENERATE-CONSTANT-CODE SECONDARY-IV)
;***
;*** Generates code in the loop pre-header for evaluating each of the linear
;*** function constants of the secondary induction variable SECONDARY-IV,
;*** and sets :A-ADDRESS and :B-ADDRESS of SECONDARY-IV to contain the
;*** temporary NADDR names holding the constants.   (If a constant is
;*** a known number, then the -ADDRESS field will contain that number
;*** directly and no code will be generated.
;***
;***=========================================================================

(defun ivr.secondary-iv:generate-constant-code ( secondary-iv )
    (assert (secondary-iv:is secondary-iv) )

    (let*( (a-address (ivr.de:generate-and-insert-naddr
                            (secondary-iv:a secondary-iv)
                            'pre-header () ) )
           (b-address (ivr.de:generate-and-insert-naddr
                            (secondary-iv:b secondary-iv)
                            'pre-header () ) ) )
        (:= (secondary-iv:a-address secondary-iv) a-address)
        (:= (secondary-iv:b-address secondary-iv) b-address)
        () ) )


;***=========================================================================
;***
;*** (IVR.SECONDARY-IV:GENERATE-LOOP-ASSIGN SECONDARY-IV)
;***
;*** Generates information in the loop header that relates a secondary
;*** variable to its primary.  The pseuod-op generated looks like:
;***
;***     TEMP := A * I + B
;***     (LOOP-ASSIGN J TEMP)
;***
;*** where I is the primary iv, and J the secondary iv with linear
;*** constants A and B.
;***
;*** LOOP-ASSIGN is a pseudo-op that has meaning to STAT:DERIVATION, which
;*** treats it as a regular assignment (to all other code, it is a no-op).
;*** The effect of this is to produce derivations for the secondary ivs
;*** in terms of the primary ivs.
;***
```



```
;*** If we didn't do this, the assertion facility would have no way to
;*** relate the secondary ivs to the programmer's assertions that are
;*** in terms of the primary ivs, because we have decoupled the secondary
;*** from the primary ivs.
;***
;***=========================================================================

(defun ivr.secondary-iv:generate-loop-assign ( secondary-iv )
    (assert (secondary-iv:is secondary-iv) )

    (let*( (iv-family (secondary-iv:iv-family secondary-iv) )
           (i         (iv-family:name         iv-family) )
           ( (temp naddr)
                (ivr.dexpr:generate-naddr
                    `(+ (* ,i ,(secondary-iv:a-address secondary-iv) )
                        ,(secondary-iv:b-address secondary-iv) ) ) ) )

        (ivr.insert-naddr
            (append1 naddr
                    '(loop-assign ,(secondary-iv:name secondary-iv) ,temp) )
            'before
            (bblock:first-stat (loop:header *ivr.loop*) ) )

        () ) )


;***=========================================================================
;***
;*** (IVR.IV-FAMILY:INDUCTION-VAR:JUMP-STAT:SIMPLIFY IV-FAMILY J JUMP-STAT)
;***
;*** Simplifies a conditional jump that uses a useless secondary or primary
;*** induction variable by rewriting the conditional jump to use a simpler
;*** secondary induction variable.
;***
;*** The conditional jump test has the form:
;***
;***    J RELOP X
;***
;*** where J is either a secondary or primary iv and expressed in terms
;*** of the primary as:
;***
;***    J = AJ * I + BJ
;***
;*** We look for another useful secondary iv K that has the form:
;***
;***    K = AK * I + BK  where  AK = R * AJ  for some known number R.
;***
;*** If we find such a K, we can rewrite the conditional test to be:
;***
;***    TEMP := R * X - R * BJ + BK
;***    K RELOP TEMP                {if R is positive}
;***    TEMP RELOP K                {if R is negative}
;***
;*** The code for TEMP can be placed in the loop header if X is a loop
;*** constant.
;***
;*** Note that when we "replace" a STAT, we really only change its source,
;*** so that reaching uses, etc.  will continue to be available (sigh).
;***
;***=========================================================================

(defun ivr.iv-family:induction-var:jump-stat:simplify
       ( iv-family j jump-stat )
```

```
(assert (iv-family:is iv-family) )
(assert (stat:is      jump-stat) )
(let*( ( jump-oper  (stat:source jump-stat) )
       ( (() aj bj) (ivr.name:iv-family&constants j) )
       ( (secondary-iv r)
                   (ivr.iv-family:a:simplest-secondary-iv&r iv-family aj) ) )

    (if secondary-iv (then
        (let*( (k       (secondary-iv:name       secondary-iv) )
               (bk      (secondary-iv:b          secondary-iv) )
               (x       (if (== j (stat:part jump-stat 'read2) )
                            (stat:part jump-stat 'read1)
                            (stat:part jump-stat 'read2) ) )
               (temp    () ) )

            (:= temp (ivr.de:generate-and-insert-naddr
                         '(+ (* ,r ,x)
                             (* -1 ,r ,bj)
                             ,bk)
                         (if (ivr.name:constant? x)
                             'pre-header
                             'before)
                         jump-stat) )

            (:= (stat:source jump-stat)
                (if (> r 0) (then
                    (oper:substitute-operand
                        (oper:substitute-operand jump-oper k j 'read)
                        temp x 'read) )
                (else
                    (oper:substitute-operand
                        (oper:substitute-operand jump-oper temp j 'read)
                        k x 'read) ) ) ) ) ) )
    () ) )

;***=======================================================================
;***
;*** (IVR.IV-FAMILY:A:SIMPLEST-SECONDARY-IV&R IV-FAMILY)
;***
;*** Finds the SECONDARY-IV K in IV-FAMILY that has the simplest linear
;*** function constants AK and BK such that AK = R * A for some constant
;*** number R.  If such a secondary iv is found the result returned is:
;***
;***     (SECONDARY-IV R)
;***
;*** Otherwise, () is returned.  R must be a compile-time number, since
;*** we'll be using it in rewriting relops above, and we'll need its sign.
;***
;*** Only secondary induction variables marked as "useful" (that have
;*** uses other than computing other induction variables) are considered
;*** here.
;***
;*** The possibilities are ranked as follows, the simplest first:
;***
;***    R  = 1, BK  = 0
;***    R  = 1, BK != 1
;***    R != 1, BK  = 0
;***    R != 1, BK != 0
;***
;***=======================================================================
```



```
(defun ivr.iv-family:a:simplest-secondary-iv&r ( iv-family a )
    (assert (iv-family:is iv-family) )

    (loop (for secondary-iv in (iv-family:secondary-ivs iv-family) )
          (when (secondary-iv:useful? secondary-iv) )

          (bind r-exp (de:divide (secondary-iv:a secondary-iv) a) )
          (when r-exp)
          (bind r-sum (cadr r-exp) )
          (when (== 2 (length r-sum) ) )
          (bind r-prod (cadr r-sum) )
          (when (== 2 (length r-prod) ) )
          (bind (() r) r-prod)

          (bind b (secondary-iv:b secondary-iv) )
          (initial simplest-secondary-iv ()
                   simplest-r            ()
                   simplest-cost         10000
                   cost                  0)
    (do
        (:= cost 0)
        (if (!== r 1) (then
            (:= cost 2) ) )
        (if (!= b '(& (+ (* 0) ) ) ) (then
            (:= cost (+ cost 1) ) ) )

        (if (< cost simplest-cost) (then
            (:= simplest-cost          cost)
            (:= simplest-r             r)
            (:= simplest-secondary-iv secondary-iv) ) ) )

    (result
        (if simplest-secondary-iv
            '(,simplest-secondary-iv ,simplest-r)
            () ) ) ) )

;***=======================================================================
;***
;*** (IVR.SECONDARY-IV:SIMPLIFY SECONDARY-IV)
;***
;*** Replaces all assignments to secondary induction variables of IV-FAMILY
;*** in the current by assignments rewritten in terms of their linear
;*** functions of the primary variable, decoupling secondary variables
;*** from the primary ones.
;***
;*** Each assignment of a secondary variable in SECONDARY-IV is replaced by:
;***
;***    J := NEW-NAME
;***
;*** where NEW-NAME is the name of a new induction variable created to
;*** replace the set of all the secondary ivs that have the same linear
;*** function of the primary (SECONDARY-IV represents that set).
;***
;*** After each primary induction statement I := I + C, we insert
;***
;***    NEW-NAME := NEW-NAME + A * C
;***
;*** In the loop header, NEW-NAME is initialized to be:
;***
;***    NEW-NAME := A * I + B.
;***
;*** Note that when we "replace" a STAT, we really only change its source,
```

```
;*** so that reaching uses, etc. will continue to be available (sigh).
;***
;***=========================================================================

(defun ivr.secondary-iv:simplify ( secondary-iv )
    (assert (secondary-iv:is secondary-iv) )

    (let*( (iv-family  (secondary-iv:iv-family secondary-iv) )
           (i          (iv-family:name iv-family) )
           (new-name   (secondary-iv:name      secondary-iv) )
           (a-address  (secondary-iv:a-address secondary-iv) )
           (a          (secondary-iv:a         secondary-iv) )
           (b          (secondary-iv:b         secondary-iv) ) )

           ;*** Replace each assignment of secondary variables J by
           ;*** J := NEW-NAME.
           ;***
       (loop (for-each-stat-set-element (secondary-iv:stats secondary-iv)
                                        stat)
          (bind j (stat:part stat 'written) )
       (do
           (:= (stat:source stat) `(iassign ,j ,new-name) ) ) )

           ;*** Generate statements in the pre-header of the form that
           ;*** evaluate:
           ;***
           ;***    NEW-NAME := A * I + B
           ;***
           ;*** where I is the primary induction variable and A and B are
           ;*** the linear function constants of the secondary variable.
           ;***
       (let ( (address (ivr.de:generate-and-insert-naddr     <- if not primary
                          `(+ (* ,i ,a) ,b)
                          'pre-header
                          () ) ) )
           (bblock:append-stat (loop:pre-header *ivr.loop*)
                               (stat:create `(iassign ,new-name ,address))))

           ;*** After each primary statement I := I +/- C, insert
           ;*** a statement of the form:
           ;***
           ;*** NEW-NAME := NEW-NAME + A * C        <- if not primary
           ;***
       (loop (for-each-stat-set-element (iv-family:primary-stats iv-family)
                                        primary-stat)
          (bind c       (if (== (stat:part primary-stat 'written)
                                (stat:part primary-stat 'read1) )
                            (stat:part primary-stat 'read2)
                            (stat:part primary-stat 'read1) )
                address (ivr.de:generate-and-insert-naddr
                            `(* ,a-address ,c)
                            (if (ivr.name:constant? c)
                                'pre-header
                                'after)
                            primary-stat)
                oper    (if (== 'isub (stat:operator primary-stat) )
                            `(isub ,new-name ,new-name ,address)
                            `(iadd ,new-name ,new-name ,address) ) )
       (do
          (stat:append-stat primary-stat (stat:create oper) ) ) )
       () ) )
```

19

```
;***=========================================================================
;***
;*** (IVR.DE:GENERATE-AND-INSERT-NADDR DEXPR WHERE? STAT)
;***
;*** Generates NADDR for an unnormalized diophantine expression and inserts
;*** it somewhere in the current, returning the name of the temporary
;*** variable holding the expression's value (or a number if the value
;*** of the expression is constant).  WHERE?  and STAT specify where the
;*** NADDR is to be inserted:
;***
;*** WHERE? = 'BEFORE      Right before STAT.
;*** WHERE? = 'AFTER       Right after  STAT.
;*** WHERE? = 'PRE-HEADER  At the end of the pre-header of the current loop.
;***
;***=========================================================================

(defun ivr.de:generate-and-insert-naddr ( dexpr where? stat)
    (assert (|| (stat:is stat)
                (! stat) ) )
    (let ( ( (address naddr) (ivr.dexpr:generate-naddr dexpr) ) )
        (ivr.insert-naddr naddr where? stat)
        address) )


;***=========================================================================
;***
;*** (IVR.INSERT-NADDR NADDR WHERE? STAT)
;***
;*** Inserts a list of NADDR operations into a spot in the current loop.
;*** WHERE?  and STAT have the same meaning as above.
;***
;***=========================================================================
X
(defun ivr.insert-naddr ( naddr where? stat )
    (assert (|| (stat:is stat)
                (! stat) ) )

    (caseq where?
        (before
            (loop (for oper in naddr) (do
                (stat:insert-stat stat (stat:create oper) ) ) ) )
        (after
            (loop (for oper in (dreverse naddr) ) (do
                (stat:append-stat stat (stat:create oper) ) ) ) )
        (pre-header
            (loop (for oper in naddr)
                  (initial pre-header (loop:pre-header *ivr.loop*) )
            (do
                (bblock:append-stat pre-header (stat:create oper)))))
        (t
            (error (list where? "Invalid WHERE? position.") ) ) )

    () )


;***=========================================================================
;***
;*** (IVR.DEXPR:GENERATE-NADDR DEXPR)
;***
;*** Generates NADDR for unnormalized diophantine expression DEXPR, which
;*** represents a linear function constant of an induction variable.  The
;*** result returned has the form:
```

20

```
;***
;***     (ADDRESS NADDR)
;***
;*** where ADDRESS is the temporary name generated to hold the constant
;*** and NADDR is a list of NADDR operations generating that constant.
;*** If DEXPR represents a single known number, then ADDRESS will be that
;*** number and NADDR will be ().
;***
;***=========================================================================

(defun ivr.dexpr:generate-naddr ( dexpr )
    (ivr.dexpr-+:generate-naddr (cadr (de:normalize dexpr) ) ) )


(defun ivr.dexpr-+:generate-naddr ( (() (() constant) . prods) )
    (if (! prods) (then
        '(,constant () ) )
    (else
        (:= prods (sort prods
                        (f:l ( prod1 prod2 )
                            (if (> (cadr prod1) (cadr prod2) )
                                t
                                (lexorder (cddr prod1) (cddr prod2) ) ) ) ) )

        (loop (for prod in prods)
              (initial address (if (== 0 constant) () constant)
                       naddr   ()
                       temp    () )
              (bind (prod-address prod-naddr) (ivr.dexpr-*:generate-naddr
                                                  prod
                                                  address) )
        (do
            (if address (then
                (:= temp  (fa.temporary-name () ) )
                (:= naddr (append naddr
                                  prod-naddr
                                  '( (,(if (> 0 (cadr prod) ) 'isub 'iadd)
                                         ,temp
                                         ,address
                                         ,prod-address) ) ) )
                (:= address temp) )
            (else
                (:= naddr   prod-naddr)
                (:= address prod-address) ) ) )
        (result '(,address ,naddr) ) ) ) ) )


(defun ivr.dexpr-*:generate-naddr ( (() constant . vars) ignore-sign? )
    (let ( (address () )
           (naddr   () )
           (temp    () ) )

        (if ignore-sign? (then
            (:= constant (abs constant) ) ) )

        (?( (== -1 constant)
            (:= temp    (fa.temporary-name () ) )
            (:= naddr   '( (ineg ,temp ,(car vars) ) ) )
            (:= address temp)
            (:= vars    (cdr vars) ) )
          ( (!== 1 constant)
            (:= address constant) ) )
```



```
        (loop (for var in vars) (do
            (if address (then
                (:= temp    (fa.temporary-name () ) )
                (:= naddr   (append naddr '( (imul ,temp ,address ,var) ) ) )
                (:= address temp) )
            (else
                (:= address var) ) ) )
        (result '(,address ,naddr) ) ) ) )
```

```lisp
;***=========================================================================
;***
;*** LIVE
;***
;*** This module computes live variable information, using the algorithm
;*** in the Dragon Book, Chap.  14.  As a special hack, the live-out of
;*** every exit BBLOCK is set to be the set of names defined in any LIVE
;*** pseudo-ops in the exit block.   This guarrantees that common
;*** sub-expression elimination will optimize the exit block correctly.
;***
;*** (FG.SET-LIVE-NAMES)
;***      Sets :LIVE-IN and :LIVE-OUT of each BBLOCK to be the NAME-SETs
;***      of names live on entry and exit to the block.
;***
;*** (STAT:LIVE-IN STAT)
;***      Returns the NAME-SET of names live on entry to STAT.
;***
;*** (STAT:LIVE-OUT STAT)
;***      Returns the NAME-SET of names live on exit to STAT.
;***
;***=========================================================================


(include flow-analysis:flow-analysis-decls)

(defmacro bblock:def ( bblock ) `(bblock:gen  ,bblock) )
(defmacro bblock:use ( bblock ) `(bblock:kill ,bblock) )


(defun fg.set-live-names ()
    (ln.set-def&use)


    (loop (for-each-bblock bblock) (do
        (:= (bblock:live-in  bblock) *fg.empty-name-set*)
        (:= (bblock:live-out bblock) *fg.empty-name-set*) ) )

    (loop (initial change () )
    (next change () )
    (do
        (loop (for bblock in (fg.depth-first-ordered-bblock-list 'reverse) )
              (initial new-out () )
        (do
            (:= new-out
                (apply 'name-set:union
                       (for (succ-bblock in (bblock:succs bblock) ) (save
                            (bblock:live-in succ-bblock) ) ) ) )

            (if (! (name-set:= new-out (bblock:live-out bblock) ) )
                (:= change t) )

            (:= (bblock:live-out bblock) new-out)
            (:= (bblock:live-in bblock)
                (name-set:union (name-set:difference (bblock:live-out bblock)
                                                     (bblock:def      bblock))
                                    (bblock:use bblock) ) ) ) ) )
    (while change) )

    (ln.hack-exit-bblocks)
    () )
```

1

```lisp
(defun ln.set-def&use ()
    (loop (for-each-bblock bblock)
    (do
        (loop (for-each-bblock-stat^ bblock stat)
              (initial def *fg.empty-name-set*
                       use *fg.empty-name-set*)
        (do
            (if (stat:definition? stat)
                (let ( (defined-name (stat:part stat 'written) ) )
                    (:= def (name-set:unioni def defined-name) )
                    (:= use (name-set:difference use
                                  (name-set:singleton defined-name) ) ) ) )

            (loop (for-each-stat-operand-read stat name) (do
                (:= use (name-set:unioni use name) ) ) ) )
        (result
            (:= (bblock:def bblock) def)
            (:= (bblock:use bblock) use) ) ) ) )
    () )


(defun ln.hack-exit-bblocks ()
    (loop (for-each-bblock bblock)
          (when (! (bblock:succs bblock) ) )
    (do
        (loop (for-each-bblock-stat bblock stat)
              (when (== 'live (stat:operator stat) ) )
        (do
            (loop (for name in (stat:part stat 'read) ) (do
                (:= (bblock:live-out bblock)
                    (name-set:unioni &&& name) ) ) ) ) ) ) ) )


;***
;*** We could do some cacheing here of results, like was done for
;*** STAT:REACHING-DEFS, but it might not be worth it.
;***


(defun stat:live-in ( stat )
    (ln.stat:live-in-exit stat t) )

(defun stat:live-out ( stat )
    (ln.stat:live-in-exit stat () ) )


(defun ln.stat:live-in-exit ( stat entry? )
    (assert (stat:is stat) )

    (loop (initial bblock   (stat:bblock stat)
                   live-out (bblock:live-out bblock) )
    (for-each-bblock-stat^ bblock succ-stat)
    (until (&& (! entry?)
               (== stat succ-stat) ) )
    (do

        (if (stat:definition? succ-stat) (then
            (:= live-out
                (name-set:difference live-out
                    (name-set:singleton (stat:part succ-stat 'written) ) ) ) ))
```

2

```
    (loop (for-each-stat-operand-read succ-stat name) (do
        (:= live-out (name-set:union1 live-out name) ) ) ) )

(until (&& entry?
           (== stat succ-stat) ) )
(result live-out) ) )
```

```
;==============================================================================
;
; LOOP
;
; A LOOP describes one loop in the flow graph:
;
;======================
(def-struct loop        ;
    header              ; A BBLOCK that is the single entry into the loop.
    found-pre-header    ; A BBLOCK that is the pre-header of the entry
                        ; (this is filled in on the fly).
    back-edges          ; BBLOCK-SET of BBLOCKs in the loop whose
                        ; successors include the header.
    bblocks             ; BBLOCK-SET of all the BBLOCKs in the loop,
                        ; include the header and the back edges.
    stats               ; STAT-SET of stats in the loop.
    invariants          ; Ordered list of STATs that are invariant.
    exits               ; All the BBLOCKs in the loop that have a
                        ; successor not in the loop.
    )                   ;
;======================
;
; (FG.INITIALZE-LOOPS)
;     Forgets all previous loops.
;
; (LOOP:CREATE TAIL-BBLOCK HEAD-BBLOCK)
;     TAIL-BBLOCK and HEAD-BBLOCK describe a backedge of a loop.  If
;     there is already a loop with HEAD-BBLOCK as the head, then this
;     backedge is just added to the loop.  If not, a new loop is created
;     (and remembered).
;
; (FG.SORT-LOOPS)
;     Sorts the loops by order of containment (outer loops come first)
;     and secondarily by source order of the headers.
;
; (LOOP:BBLOCK:MEMBER? LOOP BBLOCK)
;     Returns true if BBLOCK is part of LOOP.
;
; (LOOP:STAT:MEMBER? LOOP STAT)
;     Returns true if STAT is part of LOOP.
;
; (LOOP:NAME:DEFINING-STATS LOOP NAME)
;     Returns the STAT-SET of definitions of NAME within the loop.
;
; (LOOP:PRE-HEADER LOOP)
;     Returns the pre-header BBLOCK of LOOP (creating one if necessary).
;     A pre-header is a block that is a predecessor of the header and that
;     has only one successor, the header.  The pre-header is where any
;     invariants will be moved to.
;
; (LOOP (FOR-EACH-LOOP LOOP) ...
;     Enumerates LOOP through each loop of the flow graph, in order of
;     containment (outer loops first).
;
;==============================================================================

(include flow-analysis:flow-analysis-decls)

(defvar *fg.all-loops* () )      ;*** List of all the loops.

(defun fg.initialize-loops ()
    (:= *fg.all-loops* () ) )
```



```
(defun loop:create ( tail-bblock head-bblock )
    (loop (for loop in *fg.all-loops*) (do
        (if (== head-bblock (loop:header loop) ) (then
            (:= (loop:back-edges loop)
                (bblock-set:union1 (loop:back-edges loop) tail-bblock) )
            (:= (loop:bblocks loop)
                (bblock-set:union1 (loop:bblocks loop) tail-bblock) )
            (return loop) ) ) )
    (result
        (push *fg.all-loops*
            (loop:new header     head-bblock
                      back-edges (bblock-set:singleton tail-bblock)
                      bblocks    (bblock-set:union1
                                     (bblock-set:singleton head-bblock)
                                     tail-bblock) ) ) ) ) )


(defun loop:bblock:member? ( loop bblock )
    (assert (loop:is   loop) )
    (assert (bblock:is bblock) )
    (bblock-set:member? (loop:bblocks loop) bblock) )

(defun loop:stat:member? ( loop stat )
    (assert (loop:is   loop) )
    (assert (stat:is stat) )
    (loop:bblock:member? loop (stat:bblock stat) ) )


(defun loop:name:defining-stats ( loop name )
    (assert (loop:is loop) )
    (assert (litatom name) )
    (stat-set:intersection (name:defining-stats name)
                           (loop:stats          loop) ) )


(defun loop:pre-header ( loop )
    (if (loop:found-pre-header loop) (then
        (loop:found-pre-header loop) )
    (else
        (loop (initial non-loop-preds () )
        (for pred in (bblock:preds (loop:header loop) ) )
        (when (! (loop:bblock:member? loop pred) ) )
        (do
            (push non-loop-preds pred) )
        (result
            (:= (loop:found-pre-header loop)
                (if (= 1 (length non-loop-preds) ) (then
                    (car non-loop-preds) )
                (else
                    (bblock:splice (loop:header loop) (bblock:create)
                                   non-loop-preds) ) ) ) ) ) ) ) )


(defun fg.sort-loops ()
    (:= *fg.all-loops*
        (sort *fg.all-loops*
            #'(lambda ( loop1 loop2 )
                (|| (loop:bblock:member? loop1 (loop:header loop2) )
                    (< (bblock:number (loop:header loop1) )
                       (bblock:number (loop:header loop2) ) ) ) ) ) )
    () )
```

```
(def-simple-loop-clause for-each-loop ( clause )
    (let ( ( (for-each-loop var) clause) )

        (if (! (&& (= 2 (length clause) )
                   (litatom var) ) )
            (error (list clause "Invalid FOR-EACH-LOOP syntax.") ) )

        `( (for ,var in *fg.all-loops*) ) ) )
```

```
;========================================================================
;
; LOOP INVARIANT MOTION
;
; This module moves invariant code out of loops, closely following
; the algorithms in the Dragon Book, Chapter 13.
;
; (FG.SET-LOOP-INVARIANTS)
;     Sets the :INVARIANTS field of each LOOP to be an ordered list
;     of STATs that compute values that are invariant for each execution
;     of the loop.  The list is ordered by order that the invariant
;     statements are found.
;
; (FG.MOVE-LOOP-INVARIANTS)
;     Moves as many of the invariants as possible out of each loop.
;     *** WARNING *** Most of the flow analysis information is
;     invalidated after this; the only thing guarranteed to be correct
;     is the BBLOCK, STAT, and NAME information.
;
;========================================================================

(include flow-analysis:flow-analysis-decls)


(defun fg.set-loop-invariants ()
    (loop (for-each-loop loop) (do
        (lim.loop:set-invariants loop) ) ) )

(declare (special
    *lim.moved-invariants*  ;*** This is a global set of all invariants
                            ;*** that have been moved out of their loops.
    ) )

(defun fg.move-loop-invariants ()
    (let ( (*lim.moved-invariants* () ) )

        (loop (for-each-loop loop) (do
            (lim.loop:move-invariants loop) ) ) ) )


;***=====================================================================
;***
;*** (LIM.LOOP:SET-INVARIANTS LOOP)
;***
;*** This sets the :INVARIANTS field of LOOP to be the ordered list of
;*** invariants found within the loop body (that includes contained loops).
;*** If def S1 defines a name A that is used by def S2, and both S1 and
;*** S2 are invariant, S1 is guarranteed to come before S2 in :INVARIANTS.
;***
;*** Conceptually, repeated passes are made over the stats of the loop
;*** marking invariants, until no new invariants are marked.  For
;*** efficiency, statements are only rechecked for invariancy if at least
;*** one of their reaching definitions were marked as invariant.
;***
;***=====================================================================


(defun lim.loop:set-invariants ( loop )

        ;*** First check every statement in LOOP for possible invariancy.
        ;
    (loop (for-each-stat-set-element (loop:stats loop) stat) (do
        (lim.loop:stat:process-possible-invariancy loop stat) ) )
```



```
        ;*** Now make repeated passes over the using stats of the
        ;*** invariants so far discovered, adding any newly discovered
        ;*** invariants to LOOP, stopping when we make an entire pass
        ;*** without discovering a new invariant.
        ;
    (loop (initial change () )
    (next change () )
    (do
        (loop (for invariant-stat in (loop:invariants loop) ) (do
            (loop (for stat in (stat:reaching-uses invariant-stat) )
                  (when (stat-set:member? (loop:stats loop) stat) )
            (do
                (if (lim.loop:stat:process-possible-invariancy loop stat) (then
                    (:= change t) ) ) ) ) ) ) )

    (while change) )

    (:= (loop:invariants loop)
        (dreverse (loop:invariants loop) ) ) )



;***=====================================================================
;***
;*** (LIM.LOOP:STAT:PROCESS-POSSIBLE-INVARIANCY LOOP STAT)
;***
;*** This checks to see if STAT (assumed to be in LOOP) is an invariant
;*** in LOOP that isn't now marked as invariant.  If it is, it is added
;*** to the :INVARIANTS of LOOP, and true is returned.  False is returned
;*** otherwise.
;***
;***=====================================================================

(defun lim.loop:stat:process-possible-invariancy ( loop stat )
    (if (&& (stat:definition? stat)
            (! (memq stat (loop:invariants loop) ) )
            (lim.loop:stat-invariant? loop stat) )
    (then
        (push (loop:invariants loop) stat)
        t)
    (else
        () ) ) )


;***=====================================================================
;***
;*** (LIM.LOOP:STAT-INVARIANT? LOOP STAT)
;***
;*** Returns true if STAT is invariant in LOOP, that is, if each operand
;*** of STAT is invariant.
;***
;***=====================================================================

(defun lim.loop:stat-invariant? ( loop stat )
    (loop (for-each-stat-operand-read stat name) (do
        (if (! (lim.loop:stat-operand-invariant? loop stat name) )
            (return () ) ) )
    (result t) ) )

;***=====================================================================
;***
;*** (LIM.LOOP:STAT-OPERAND-INVARIANT? LOOP STAT NAME)
```

```
;***
;*** Returns true if the operand NAME of STAT is invariant in LOOP.
;*** An operand is invariant if either:
;***
;*** 1. It is a constant, or
;*** 2. All of its reaching defs are outside of the loop, or
;*** 3. There is exactly one reaching def and it is already marked invariant.
;***
;***
;***============================================================================

(defun lim.loop:stat-operand-invariant? ( loop stat name )

            ;*** Is the operand a constant?
            ;
    (|| (numberp name)

                ;*** Or are all the reaching defs outside of the loop,
                ;*** or is there exactly one reaching def and it is marked
                ;*** invariant?
                ;
        (loop (initial all-outside?   t
                       all-invariant? t
                       count          0)
        (for-each-stat-set-element (stat:operand-reaching-defs stat name)
                                   def-stat)
        (do
            (++ count)
            (:= all-outside?
                (&& all-outside?
                    (! (loop:stat:member? loop def-stat) ) ) )
            (:= all-invariant?
                (&& all-invariant?
                    (memq def-stat (loop:invariants loop) ) ) ) )
        (result
            (|| all-outside?
                (&& (= 1 count)
                    all-invariant?) ) ) ) ) )


;***============================================================================
;***
;*** (LIM.LOOP:MOVE-INVARIANTS LOOP)
;***
;*** Moves as many invariant STATs out of LOOP as possible.  An invariant
;*** statement S that defines name A may be moved out of the loop if:
;***
;*** 1. A is not defined elsewere in the loop.
;*** 2. a) A is dead on every exit from the loop, or b) S dominates every
;***    exit of the loop.
;*** 3. Each use of this definition of A is reached only by this definition
;***    and has no other defs of A reaching it.
;*** 4. If the reaching definitions of the operands of S come from inside
;***    the loop, then those definitions have been previously moved out
;***    of the loop as an invariant.
;***
;*** Condition 2b) is "dangerous", in that the order of computation is
;*** now changed relative to conditionals.  E.g. a division A/B could
;*** be moved out of a loop, up past a conditional that tests to see if
;*** B is non-zero.  The Dragon Book recommends programmer-accessible
;*** switches for disabling it.
;***
;*** Note that invariants are tested for moving eligibility in the order
;*** that they were marked invariant (in the order of LOOP:INVARIANTS).
;*** This insures that we consider an invariant only after all its
;*** predecessor invariants have been considered;  if a def has operands
;*** that depend on previous invariants, those invariants must have been
;*** moved out in order for def to be moved out.
;***
;***============================================================================

(defun lim.loop:move-invariants ( loop )
    (loop (initial name      () )
    (for stat in (loop:invariants loop) )
    (when (! (stat-set:member? *lim.moved-invariants* stat) ) )
    (do
        (if (stat:definition? stat) (then
            (:= name (stat:part stat 'written) )
            (if (&& (lim.loop:name-not-defined-elsewhere?   loop name)
                    (|| (lim.loop:name-dead-on-exit?         loop name)
                        (lim.loop:stat-dominates-all-exits? loop stat) )
                    (lim.loop:def-reaches-all-uses?         loop stat name)
                    (lim.loop:operand-defs-moved?           loop stat) )
                (then
                    (stat:extract stat)
                    (bblock:append-stat (loop:pre-header loop)
                                        stat)
                    (:= *lim.moved-invariants*
                        (stat-set:union1 *lim.moved-invariants* stat) ) ) ))))))

;***============================================================================
;***
;*** (LIM.LOOP:NAME-NOT-DEFINED-ELSEWHERE? LOOP NAME)
;***
;*** Returns true if there is only one definition of NAME within LOOP.
;***
;***============================================================================

(defun lim.loop:name-not-defined-elsewhere? ( loop name )
    (= 1
       (stat-set:size (stat-set:intersection (loop:stats loop)
                                             (name:defining-stats name) ) ) ) )


;***============================================================================
;***
;*** (LIM.LOOP:NAME-DEAD-ON-EXIT? LOOP NAME)
;***
;*** Returns true if NAME is dead on every exit from LOOP.
;***
;***============================================================================

(defun lim.loop:name-dead-on-exit? ( loop name )
    (loop (label 1)
    (for-each-bblock-set-element (loop:exits loop) exit)
    (do
        (loop (for exit-succ in (bblock:succs exit) )
        (when (! (loop:bblock:member? loop exit-succ) ) )
        (do
            (if (name-set:member? (bblock:live-in exit-succ) name)
                (leave 1 () ) ) ) ) ) )
    (result t) ) )


;***============================================================================
;***
```

```
;*** (LIM.LOOP:STAT-DOMINATES-ALL-EXITS? LOOP STAT)
;***
;*** Returns true if STAT dominates every exit of LOOP.
;***
;***=====================================================================

(defun lim.loop:stat-dominates-all-exits? ( loop stat )
    (loop (initial stat-bblock (stat:bblock stat) )
    (for-each-bblock-set-element (loop:exits loop) exit-bblock)
    (do
        (if (! (bblock:dominates? stat-bblock exit-bblock) )
            (return () ) ) )
    (result t) ) )


;***=====================================================================
;***
;*** (LIM.LOOP:DEF-REACHES-ALL-USES? LOOP DEF-STAT NAME)
;***
;*** DEF-STAT is assumed to define NAME.  Returns true if each use of
;*** DEF-STAT within LOOP has DEF-STAT as the only reaching def that
;*** defines NAME.
;***
;***=====================================================================

(defun lim.loop:def-reaches-all-uses? ( loop def-stat name )
    (loop (label 1)
    (for use-stat in (stat:reaching-uses def-stat) )
    (when (stat-set:member? (loop:stats loop) use-stat) )
    (do
        (loop (for-each-stat-set-element
                    (stat:operand-reaching-defs use-stat name)
                    reaching-stat)
        (do
            (if (!== reaching-stat def-stat)
                (leave 1 () ) ) ) ) )
    (result t) ) )


;***=====================================================================
;***
;*** (LIM.LOOP:OPERAND-DEFS-MOVED? LOOP STAT)
;***
;*** Returns true if each reaching def of every read operand of STAT is
;*** outside of the loop (either because it was outside originally or
;*** because it was an invariant that was moved).
;***
;***=====================================================================

(defun lim.loop:operand-defs-moved? ( loop stat )
    (loop (for-each-stat-operand-read stat name)
          (initial loop-reaching-defs () )
    (do
        (:= loop-reaching-defs
            (stat-set:intersection (stat:operand-reaching-defs stat name)
                                   (loop:stats loop) ) )
        (if (! (stat-set:= *fg.empty-stat-set*
                           (stat-set:difference loop-reaching-defs
                                                *lim.moved-invariants*) ) )
            (return () ) ) )
    (result t) ) )
```

```
;***========================================================================
;***
;*** This module contains functions for converting NADDR to flow graphs.
;***
;***========================================================================


(include flow-analysis:flow-analysis-decls)


;***========================================================================
;***
;*** (FG.INITIALIZE-NADDR-TO-FLOW-GRAPH NADDR)
;***
;*** Initializes this module.
;***
;***========================================================================

(defun fg.initialize-naddr-to-flow-graph ()
    (:= *fg.entry-bblock* () )
    () )


;***========================================================================
;***
;*** (FG.NADDR-TO-FLOW-GRAPH NADDR)
;***
;*** Constructs a flow-graph from NADDR (a list of NADDR operations).
;***
;***========================================================================

(defun fg.naddr-to-flow-graph ( naddr )
    (ntfg.naddr-to-stats naddr)
    (ntfg.stats-to-bblocks)
    (fg.remove-unreachable-bblocks)
    )


;***========================================================================
;***
;*** (NTFG.NADDR-TO-STATS NADDR)
;***
;*** Constructs the graph of STATs corresponding to NADDR (a list of NADDR
;*** operations).
;***
;*** Two passes are used.  The first pass creates a STAT for each NADDR
;*** operation (except LABEL and GOTO) and installs the "value" of each
;*** NADDR label in a symbol table.  The second pass goes over each STAT
;*** and replaces the symbol labels in the :SUCC field with the
;*** corresponding STAT successor.
;***
;*** When we are finished, the :SUCC and :PRED fields are lists of STATs
;*** that are the predecessors and successors of the STAT in the flow
;*** graph.  When we make basic blocks, those fields will be set to single
;*** STATs (the pred and succ within the BBLOCK).
;***
;***========================================================================

(defun ntfg.naddr-to-stats ( naddr )

(let ( (labels (hash-table:create) ) )
```

```
        ;
        ;*** LABELS is a hash table mapping NADDR labels onto either
        ;*** the STATs representing the NADDR operation attached to
        ;*** that label, or else other labels.  For example, if L5
        ;*** is mapped onto L6, then L5 and L6 reference the same
        ;*** NADDR operation.

        ;*** The first pass creates a STAT for each operation, and
        ;*** sets its SUCCS field to be a list of the symbolic NADDR
        ;*** labels of the successor.  Fall-through successors that
        ;*** have no NADDR label are stored directly in the SUCCS
        ;*** field of the predecessors.  Labels and their corresponding
        ;*** STATs/other labels are installed in the table as they
        ;*** are encountered.
        ;
(loop (initial stat           ()
               current-labels ()
               prev-stat      () )
(for oper in naddr)
(do
    (caseq (oper:operator oper)

        (label       ;*** Remember this label on the list of current
                     ;*** equivalent labels.  If the previous operation
                     ;*** falls through to here, set its successor
                     ;*** to be this label.
                     ;
            (push current-labels (oper:part oper 'label1) )
            (if prev-stat (then
                (push (stat:succ prev-stat)
                      (oper:part oper 'label1) )
                (:= prev-stat () ) ) ) )


        (goto        ;*** Equate all current equivalent labels with
                     ;*** the destination of this GOTO.  If the previous
                     ;*** operation falls through to here, set its
                     ;*** successor to be the destination of the GOTO.
                     ;
            (for (label in current-labels) (do
                (hash-table:put labels label
                                          (oper:part oper 'label1) ) ) )
            (:= current-labels () )

            (if prev-stat (then
                (push (stat:succ prev-stat)
                      (oper:part oper 'label1) )
                (:= prev-stat () ) ) ) )

        (t           ;*** Some random operation:  Create a STAT for
                     ;*** it and equate all current equivalent labels
                     ;*** to it.  If the previous operation falls
                     ;*** through to here set its successor to be the
                     ;*** new STAT.  If STAT is a conditional jump,
                     ;*** set its successors to be the true/false
                     ;*** labels.
                     ;
            (:= stat (stat:create oper) )
            (for (label in current-labels) (do
                (hash-table:put labels label stat) ) )
            (:= current-labels () )

            (if prev-stat
```

```
                    (push (stat:succ prev-stat) stat) )

                (if (stat:property? stat 'conditional-jump) (then
                    (push (stat:succ stat)
                          (oper:part oper 'label1) )
                    (if (oper:part oper 'label2) (then
                        (push (stat:succ stat)
                              (oper:part oper 'label2) )
                        (:= prev-stat () ) )
                    (else
                        (:= prev-stat stat) ) ) )
                (else
                    (:= prev-stat stat) ) ) ) ) ) )


            ;*** Now make a second pass over all STATs, replacing symbolic
            ;*** label successors with actual STATs, and setting the
            ;*** predecessors field of each STAT.
            ;
    (loop (for-each-stat stat) (do

                ;*** Replace labels with actual STATs
                ;
        (loop (initial rest-succs (stat:succ stat) )
        (while rest-succs)
        (do
            (if (litatom (car rest-succs) )
                (:= (car rest-succs)
                    (ntfg.label:stat labels (car rest-succs) ) ) ) )
        (next rest-succs (cdr rest-succs) ) )

                ;*** The :SUCC field was built in reverse (tricky!).
                ;
        (:= (stat:succ stat) (dreverse (stat:succ stat) ) )

                ;*** Set the predecessor field of each successor to
                ;*** include this stat.
                ;
        (for (succ-stat in (stat:succ stat) ) (do
            (push (stat:pred succ-stat) stat) ) ) ) )
    ) )

;***========================================================================
;***
;*** (NTFG.LABEL:STAT LABELS LABEL)
;***
;*** Returns the STAT associated with LABEL in the hash table LABELS.
;*** Labels can be equated with other labels in the table, so we have
;*** to follow such chains until we find an actual STAT.
;***
;***========================================================================

(defun ntfg.label:stat ( labels label )
    (loop (while (litatom label) ) (do
        (:= label (hash-table:get labels label) ) )
    (result label) ) )


;***========================================================================
;***
;*** (NTFG.STATS-TO-BBLOCKS)
;***
```

```
;*** Constructs the basic blocks corresponding to the graph of STATS built
;*** by NTFG.NADDR-TO-STATS.  Once the BBLOCKs are built, the :SUCC and
;*** :PRED field of each STAT are changed to form a doubly linked list of
;*** STATs within the BBLOCK.
;***
;*** Sets *FG.ENTRY-BBLOCK* to the first STAT's BBLOCK (we assume it is
;*** the entry node of the graph).
;***
;***========================================================================

(defun ntfg.stats-to-bblocks ()

            ;*** For each STAT in the graph (in source order), if it is
            ;*** the leader of a basic block, start a new basic block.
            ;*** If it isn't a leader, add it to the current basic block.
            ;
    (loop (initial bblock () )
    (for-each-stat stat)
    (do
        (if (ntfg.stat:bblock-leader? stat) (then
            (:= bblock (bblock:create) )
            (:= (bblock:first-stat bblock) stat)

            (loop (initial next-stat stat)
            (do
                (:= (stat:bblock next-stat) bblock)
                (:= (bblock:last-stat bblock) next-stat) )
            (next next-stat (car (stat:succ next-stat) ) )
            (while (&& next-stat
                       (! (ntfg.stat:bblock-leader? next-stat) ) ) ) ) ) ) ) )


            ;*** For each BBLOCK, set its predecessor and successor blocks.
            ;*** Also sets the entry BBLOCK to be the first one.
            ;
    (:= *fg.entry-bblock* () )
    (loop (for-each-bblock bblock) (do
        (if (! *fg.entry-bblock*)
            (:= *fg.entry-bblock* bblock) )

        (:= (bblock:preds bblock)
            (for (stat in (stat:pred (bblock:first-stat bblock) ) ) (save
                (stat:bblock stat) ) ) )
        (:= (bblock:succs bblock)
            (for (stat in (stat:succ (bblock:last-stat bblock) ) ) (save
                (stat:bblock stat) ) ) ) ) )

            ;*** Currently :PRED and :SUCC of each STAT are lists of
            ;*** of the successor and predecessor STATs of that STAT.
            ;*** Change :PRED and :SUCC so that they now form a doubly
            ;*** linked list of STATs within a BBLOCK.  :PRED of the
            ;*** first stat in a BBLOCK is ();  likewise of :SUCC of the
            ;*** last STAT.
            ;
    (loop (for-each-bblock bblock) (do
        (loop (initial stat      (bblock:first-stat bblock)
                       prev-stat () )
        (while stat)
        (while (== bblock (stat:bblock stat) ) )
        (while (listp (stat:succ stat) ) )  ;*** have we done this STAT yet?
        (do
            (:= (stat:pred stat) prev-stat)
            (if prev-stat
```

```
            (:= (stat:succ prev-stat) stat) ) )
      (next prev-stat stat
            stat      (car (stat:succ stat) ) )
      (result
          (:= (stat:succ prev-stat) () ) ) ) ) )

   () )


;***=========================================================================
;***
;*** (NTFG.STAT:BBLOCK-LEADER? STAT)
;***
;*** Returns true if STAT is the leader (beginning) of a basic block.
;*** A STAT is a leader if it has 0 or more than 1 predecessors or if
;*** its predecessor has more than one successor.
;***
;***=========================================================================


(defun ntfg.stat:bblock-leader? ( stat )
    (|| (! (stat:pred stat) )
        (< 1 (length (stat:pred stat) ) )
        (< 1 (length (stat:succ (car (stat:pred stat) ) ) ) ) ) )
```

```
;============================================================================
;
; NAMES
;
; This module defines the type "name".  A name is a variable
; name used within the flow graph.  For now, names are
; represented externally as symbols to make handling NADDR
; straightforward; a hash table is used to map names onto
; descriptive information.  We'll see how much we pay for this
; hashing.
;
; (FG.INITIALIZE-NAMES)
;     Forgets all previous names and prepares for the creation
;     of new ones.
;
; (NAME:CREATE NAME TYPE SIZE)
;     Creates a new name with symbolic name NAME.  TYPE is either
;     SCALAR or VECTOR.  SIZE is the declared size of a vector.
;
; (NAME:DEFINING-STATS NAME)
;     Returns the STAT-SET of definitions that define NAME (initially empty).
;
; (NAME:ADD-DEFINING-STAT NAME STAT )
;     Adds STAT to the set of definitions defining NAME.
;
; (NAME:USING-STATS NAME)
;     Returns the STAT-SET of statements reading NAME (initially empty).
;
; (NAME:ADD-USING-STAT NAME STAT )
;     Adds STAT to the set of statements using NAME.
;
; (NAME:TYPE NAME)
;     The type of a name, either VECTOR or SCALAR.
;
; (NAME:LENGTH NAME)
;     The declared size of a vector name.
;
; (NAME:NUMBER NAME)
;     Maps NAME to a unique number.
;
; (NUMBER:NAME INDEX)
;     Returns the name with :NUMBER INDEX.
;
; ##N I
;     Syntax for (NUMBER:NAME I)
;
; (LOOP (FOR-EACH-NAME NAME) ...
;     Enumerates through each known NAME.
;
; (LOOP (FOR-EACH-VECTOR-NAME NAME) ...
;     Enumerates through each known NAME declared to be a vector.
;
; (FG.PRINT-NAMES)
;    Prints out all known NAMEs and their associated info.
;
;============================================================================

(include flow-analysis:flow-analysis-decls)

(def-struct name-descriptor ;*** A NAME-DESCRIPTOR is an internal record
                            ;*** describing the information about a name.
    name                    ;*** The name this describes.
    number                  ;*** The unique number assigned to this name.
```



```
    type                ;*** SCALAR or VECTOR.
    length              ;*** Length of a VECTOR.
    defining-stats      ;*** The STAT-SET of definitions defining this stat.
    using-stats         ;*** The STAT-SET of statements using this stat.
    )

(declare (special
    *fg.name:name-descriptor*
                            ;*** A hash table mapping variable names onto
                            ;*** NAME-DESCRIPTORS.
    *fg.number:name*        ;*** An array mapping numbers to names.
    *fg.total-names*        ;*** Total number of names.
    *fg.all-vector-names*   ;*** List of all vector names.
    ) )

(declare (special
    *hash-table.not-found*
    ) )


(defun fg.initialize-names ()
    (:= *fg.all-vector-names*     () )
    (:= *fg.name:name-descriptor* (hash-table:create) )
    (vector-map:initialize '*fg.number:name* '*fg.total-names* 100)
    () )


(defun name:create ( name type length )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )

        (if (== *hash-table.not-found* desc) (then
            (hash-table:put *fg.name:name-descriptor* name
                (name-descriptor:new name           name
                                     number         *fg.total-names*
                                     type           type
                                     length         length
                                     defining-stats *fg.empty-stat-set*
                                     using-stats    *fg.empty-stat-set*) )
            (vector-map:add-element '*fg.number:name* '*fg.total-names*
                                    name 100 t) )
        (else
                    ;*** if we have just discovered NAME to be a vector,
                    ;*** override any previous guess;  sigh, PARAM for
                    ;*** for vectors seems bogus?
                    ;
            (if (== 'vector type) (then
                (:= (name-descriptor:type   desc) type)
                (:= (name-descriptor:length desc) length) ) ) ) )

        name) )


(defun name:defining-stats ( name )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )
        (assert (!== desc *hash-table.not-found*)
            "NAME:DEFINING-STATS: name = " name)
        (name-descriptor:defining-stats desc) ) )


(defun name:add-defining-stat ( name stat )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )
        (assert (!== desc *hash-table.not-found*)
            "NAME:ADD-DEFINING-STAT: name = " name " stat = " (h stat) )
```

```
        (:= (name-descriptor:defining-stats desc)
            (stat-set:union1 (name-descriptor:defining-stats desc)
                             stat) ) ) )

(defun name:using-stats ( name )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )
        (assert (!== desc *hash-table.not-found*)
            "NAME:USING-STATS: name = " name)
        (name-descriptor:using-stats desc) ) )


(defun name:add-using-stat ( name stat )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )
        (assert (!== desc *hash-table.not-found*)
            "NAME:ADD-USING-STAT: name = " name " stat = " (h stat) )
        (:= (name-descriptor:using-stats desc)
            (stat-set:union1 (name-descriptor:using-stats desc)
                             stat) ) ) )

(defun name:type ( name )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )
        (assert (!== desc *hash-table.not-found*)
            "NAME:TYPE: name = " name)
        (name-descriptor:type desc) ) )


(defun name:length ( name )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )
        (assert (!== desc *hash-table.not-found*)
            "NAME:LENGTH: name = " name)
        (name-descriptor:length desc) ) )


(defun name:number ( name )
    (let ( (desc (hash-table:get *fg.name:name-descriptor* name) ) )
        (assert (!== desc *hash-table.not-found*)
            "NAME:NUMBER: name = " name)
        (name-descriptor:number desc) ) )

(defun number:name ( index )
    ([] *fg.number:name* index) )

(def-sharp-sharp n
    '([] *fg.number:name* ,(read) ) )

(def-simple-loop-clause for-each-name ( clause )
    (let ( ( (for-each-name var) clause)
           (index (intern (gensym) ) ) )

        (if (! (&& (= 2 (length clause) )
                   (litatom var) ) )
            (error (list clause "Invalid FOR-EACH-NAME syntax.") ) )

        '( (initial ,var () )
           (incr ,index from 0 to (+ -1 *fg.total-names*) )
           (next ,var ([] *fg.number:name* ,index) )
           (when ,var) ) ) )

(def-simple-loop-clause for-each-vector-name ( clause )
    (let ( ( (for-each-vector-name var) clause) )

        (if (! (&& (= 2 (length clause) )
                   (litatom var) ) )
```



```
            (error (list clause "Invalid FOR-EACH-VECTOR-NAME syntax.") ) )

        '( (for ,var in *fg.all-vector-names*) ) ) )

(defun fg.print-names ()
    (loop (for-each-name name) (do
        (msg 0 name ":" (t 10) (name:type name) (t 17) (name:length name)
             (t 21) (e (stat-set:print (name:defining-stats name) ) ) t) ) )
    () )
```

```
;===========================================================================
; NAME-SETS
;
; Sets of NAMEs are represented using NAME-SETs, currently implemented
; as BIT-SETs.
;
; *FG.EMPTY-NAME-SET*
;     The empty NAME-SET.
;
; (NAME-SET:UNIVERSE)
;     The set of all NAMEs.
;
; (NAME-SET:SINGLETON NAME)
;     Creates a new set containing NAME.
;
; (NAME-SET:MEMBER? SET NAME)
;     Returns true if NAME is a member of SET.
;
; (NAME-SET:INTERSECTION SET1 SET2 ...)
;     Returns a new set that is the intersection of all the given sets.
;
; (NAME-SET:UNION SET1 SET2 ...)
;     Returns a new set that is the union of all the given sets.
;
; (NAME-SET:UNION1 SET NAME)
;     Unions a single NAME into SET.
;
; (NAME-SET:DIFFERENCE SET1 SET2)
;     Returns a new set that contains all elements in SET1 not in SET2.
;
; (NAME-SET:= SET1 SET2)
;     Returns true if the two sets are equal.
;
; (NAME-SET:SIZE SET)
;     Returns the number of elements in the set.
;
; (LOOP (FOR-EACH-NAME-SET-ELEMENT SET NAME)
;     Enumerates NAME through each element in SET.  Uses
;     DEF-SIMPLE-LOOP-CLAUSE.
;
; (NAME-SET:PRINT SET)
;    Prints SET.
;
; (NAME-SET:LIST SET)
;    Returns a list of the names in SET.
;
;===========================================================================

(include flow-analysis:flow-analysis-decls)

(declare (special *fg.total-names*) )

(defvar *fg.empty-name-set* () )    ;*** the empty NAME-SET.

(defun name-set:universe ()
    (bit-set:universe *fg.total-names*) )

(defun name-set:singleton ( name )
    (bit-set:singleton (name:number name) ) )

(defun name-set:member? ( set name )
    (bit-set:member? set (name:number name) ) )
```



```
(defun name-set:intersection args
    (apply 'bit-set:intersection (listify-lexpr-args args) ) )

(defun name-set:union args
    (apply 'bit-set:union (listify-lexpr-args args) ) )

(defun name-set:union1 ( set name )
    (bit-set:union1 set (name:number name) ) )

(defun name-set:difference ( set1 set2 )
    (bit-set:difference set1 set2) )

(defun name-set:= ( set1 set2 )
    (bit-set:= set1 set2) )

(defun name-set:size ( set )
    (bit-set:size set) )

(def-simple-loop-clause for-each-name-set-element ( clause )
    (let ( ( (for-each-name-set-element set name) clause)
           (index (intern (gensym) ) ) )
        (if (! (&& (= 3 (length clause) )
                   (litatom name) ) )
           (error (list clause "Invalid FOR-EACH-NAME-SET-ELEMENT syntax.")))

        `( (initial ,name () )
           (for-each-bit-set-element ,set ,index)
           (next ,name (number:name ,index) )
           (when ,name) ) ) )

(defun name-set:print ( set )
    (msg "{")

    (loop (for-each-name-set-element set name)
          (initial first t)
    (do
        (if (! first) (then
            (msg " ") ) )
        (:= first () )
        (msg name) ) )

    (msg "}")
    () )


(defun name-set:list ( set )
    (loop (for-each-name-set-element set name) (save
        name) ) )
```

```
;==========================================================================
;
; REACHING COPIES
;
;
; This module implements the flow analysis algorithm that finds the "reaching
; copies" of each basic block.  The definition of a reaching copy used here
; is a little more general than that used in the Dragon Book on page 487.
;
; The set of reaching copies at point S2 is the set of statements S1 of
; the form:
;
;     S1: A := f( B1, B2, ... )
;
; such that for every path from the initial node to S2:
;
;     1.  The path includes S1.
;     2.  After the last occurrence of S1 on the path, there are no
;         definitions of the variables A or any of the Bi.
;
; Intuitively, if S1 is a reaching copy at S2, S1 could be copied right
; before S2 without affecting the program.
;
;
; (FG.SET-REACHING-COPIES)
;     Does the flow analysis of reaching copies, setting the
;     :REACHING-COPIES-IN and :REACHING-COPIES-OUT of each BBLOCK.
;
; (STAT:REACHING-COPIES STAT)
;     Returns the STAT-SET of definitions that are reaching copies at STAT.
;
;==========================================================================

(include flow-analysis:flow-analysis-decls)


;***=======================================================================
;***
;*** (FG.SET-REACHING-COPIES)
;***
;***=======================================================================

(defun fg.set-reaching-copies ()

    (rc.set-gen&kill)

    (loop (for-each-bblock bblock) (do
        (:= (bblock:reaching-copies-in  bblock) (stat-set:universe) )
        (:= (bblock:reaching-copies-out bblock)
            (stat-set:difference (stat-set:universe)
                                 (bblock:kill bblock) ) ) ) )

    (:= (bblock:reaching-copies-in  *fg.entry-bblock*) *fg.empty-stat-set*)
    (:= (bblock:reaching-copies-out *fg.entry-bblock*)
        (bblock:gen                 *fg.entry-bblock*) )

    (loop (initial change ()
                   new-in () )
    (next change () )
    (do
        (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward) )
        (do
            (:= new-in
```



```
                (apply 'stat-set:intersection
                       (for (pred-bblock in (bblock:preds bblock) ) (save
                            (bblock:reaching-copies-out pred-bblock) ) ) ) )

            (if (! (stat-set:= new-in (bblock:reaching-copies-in bblock) ) )
                (:= change t) )

            (:= (bblock:reaching-copies-in  bblock) new-in)
            (:= (bblock:reaching-copies-out bblock)
                (stat-set:union
                    (stat-set:difference (bblock:reaching-copies-in   bblock)
                                         (bblock:kill                 bblock) )
                    (bblock:gen bblock) ) ) ) ) )
    (while change) )

    () )


;***=======================================================================
;***
;*** (RC.SET-GEN&KILL)
;***
;*** Sets the :GEN and :KILL fields of each BBLOCK to be the definitions
;*** copy-generated and copy-killed by that BBLOCK.   A definition of
;*** the form:
;***
;***      A := f( B1, B2, ... )
;***
;*** is copy-generated if it is contained in the BBLOCK and follows any
;*** assignments to the variables A or Bi in the same BBLOCK.  The
;*** definition is killed by the BBLOCK if it isn't in it and if there
;*** is an assignment to at least one of the variables A or Bi.
;***
;***=======================================================================

(defun rc.set-gen&kill ()
    (loop (for-each-bblock bblock) (do

        (loop (for-each-bblock-stat bblock stat)
              (initial gen  *fg.empty-stat-set*
                       kill *fg.empty-stat-set*)
        (do
            (if (stat:definition? stat) (then
                (let ( (killed-defs (rc.stat:killed-copies stat) ) )
                    (:= kill (stat-set:union kill killed-defs) )
                    (:= gen  (stat-set:union1
                                (stat-set:difference gen killed-defs)
                                stat) ) ) ) ) )
        (result
            (:= (bblock:gen  bblock) gen)
            (:= (bblock:kill bblock) kill) ) ) ) )
    () )


;***=======================================================================
;***
;*** (STAT:REACHING-COPIES STAT)
;***
;*** Returns the STAT-SET of definitions copy-reaching STAT.  If need
;*** be, we might have to implement some sort of cacheing scheme here as
;*** was done for STAT:REACHING-DEFS.
;***
;***=======================================================================
```

```
(defun stat:reaching-copies ( stat )
    (assert (stat:is stat) )

    (loop (initial reaching-copies (bblock:reaching-copies-in
                                      (stat:bblock stat) ) )
          (for-each-bblock-stat (stat:bblock stat) pred-stat)
          (while (!== pred-stat stat) )
          (when (stat:definition? pred-stat) )
    (do
        (:= reaching-copies
            (stat-set:union1 (stat-set:difference &&&
                                  (rc.stat:killed-copies pred-stat) )
                             pred-stat) ) )
    (result reaching-copies) ) )


;***=========================================================================
;***
;*** (RC.STAT:KILLED-COPIES STAT)
;***
;*** Returns that STAT-SET of definitions that are copy-killed by STAT.
;*** STAT is assumed to be a definition.
;***
;***=========================================================================

(defun rc.stat:killed-copies ( stat )
    (let ( (written (stat:part stat 'written) ) )
        (stat-set:union
            (name:defining-stats written)
            (name:using-stats    written) ) ) )
```

```
;===========================================================================
;
; REACHING DEFINITIONS
;
; This module implements the flow analysis algorithm that finds the
; defintions that reach each basic block.  See chapters 12 and 14 of
; the Dragon Book for explanation of the algorithm.
;
; (FG.SET-REACHING-DEFS)
;     Does the flow analysis of reaching definitions, setting the
;     :REACHING-IN and :REACHING-OUT of each BBLOCK in the flow graph
;     to be a STAT-SET of definitions reaching that BBLOCK.
;
; (STAT:REACHING-DEFS STAT)
;     Returns the STAT-SET of definitions reaching STAT.
;
; (STAT:OPERAND-REACHING-DEFS STAT NAME)
;     Returns the STAT-SET of definitions of NAME reaching STAT.
;
; (FG.INITIALIZE-REACHING-DEFS)
;     Initializes this module.
;
; (STAT:KILLED-DEFS STAT)
;    Returns the STAT-SET of definitions possibly killed by STAT (which
;    is assumed to be a definition).
;
;===========================================================================


(include flow-analysis:flow-analysis-decls)

(declare (special
    *fg.number-of-reaching-iterations*
                                ;*** # of iterations used for calculating
                                ;*** reaching defs

    *rd.last-reaching-stat*     ;*** The last STAT given to STAT:REACHING-DEFS.
    *rd.last-reaching-defs*     ;*** The last result of the last call to
                                ;*** STAT:REACHING-DEFS.  These two variables
                                ;*** form a one-result cache (hack).
    ) )

;***========================================================================
;***
;*** (FG.INITIALIZE-REACHING-DEFS)
;***
;***========================================================================

(defun fg.initialize-reaching-defs ()
    (:= *rd.last-reaching-stat* () )
    (:= *rd.last-reaching-defs* () )
    )


;***========================================================================
;***
;*** (FG.SET-REACHING-DEFS)
;***
;***========================================================================


(defun fg.set-reaching-defs ()
    (fg.initialize-reaching-defs)
```



```
    (rd.set-gen&kill)

    (loop (for-each-bblock bblock) (do
        (:= (bblock:reaching-in  bblock) *fg.empty-stat-set*)
        (:= (bblock:reaching-out bblock) (bblock:gen bblock) ) ) )

    (:= *fg.number-of-reaching-iterations* 0)
    (loop (initial change ()
                   new-in () )
    (next change () )
    (do
        (++ *fg.number-of-reaching-iterations*)

        (loop (for bblock in (fg.depth-first-ordered-bblock-list 'forward) )
        (do
            (:= new-in
                (apply 'stat-set:union
                    (for (pred-bblock in (bblock:preds bblock) ) (save
                         (bblock:reaching-out pred-bblock) ) ) ) ) )

            (if (! (stat-set:= new-in (bblock:reaching-in bblock) ) )
                (:= change t) )

            (:= (bblock:reaching-in  bblock) new-in)
            (:= (bblock:reaching-out bblock)
                (stat-set:union
                    (stat-set:difference (bblock:reaching-in   bblock)
                                         (bblock:kill bblock) )
                    (bblock:gen bblock) ) ) ) ) )
    (while change) )


    () )


;***========================================================================
;***
;*** (RD.SET-GEN&KILL)
;***     Sets the :GEN and :KILL fields of each BBLOCK to be the definitions
;***     generated and killed by that BBLOCK.
;***
;***========================================================================

(defun rd.set-gen&kill ()
    (loop (for-each-bblock bblock)
    (do
        (loop (for-each-bblock-stat bblock stat)
              (initial gen        *fg.empty-stat-set*
                       kill       *fg.empty-stat-set*)
        (do
            (if (stat:definition? stat) (then
                (let ( (killed-defs (stat:killed-defs stat) ) )
                    (:= kill (stat-set:union kill killed-defs) )
                    (:= gen  (stat-set:union1
                                 (stat-set:difference gen killed-defs)
                                 stat) ) ) ) ) )
        (result
            (:= (bblock:gen  bblock) gen)
            (:= (bblock:kill bblock) kill) ) ) ) )
    () )
```

```
;***==========================================================================
;***
;*** (STAT:REACHING-DEFS STAT)
;***    Returns the STAT-SET of definitions reaching STAT.  Computes the
;***    set by starting with the defintions reaching STATs BBLOCK, and
;***    working down to STAT.  As a hack, the argument and the result
;***    of the previous call to STAT:REACHING-DEFS is remembered; if STAT
;***    happens to be the successor of the previous argument, we can
;***    quickly compute the reaching defs of STAT.  This handles the common
;***    case where we enumerate through the statements of a BBLOCK.
;***
;***==========================================================================

(defun stat:reaching-defs ( stat )
    (assert (stat:is stat) )

    (if (== stat *rd.last-reaching-stat*) (then
         *rd.last-reaching-defs*)

    (else
        (loop (initial reaching-defs ()
                       defined-name  ()
                       pred-stat     () )
        (begin
            (if (&& *rd.last-reaching-stat*
                     (== (stat:bblock stat)
                         (stat:bblock *rd.last-reaching-stat*) )
                     (== *rd.last-reaching-stat* (stat:pred stat) ) )
            (then
                (:= pred-stat     *rd.last-reaching-stat*)
                (:= reaching-defs *rd.last-reaching-defs*) )
            (else
                (:= pred-stat     (bblock:first-stat (stat:bblock stat) ) )
                (:= reaching-defs (bblock:reaching-in
                                      (stat:bblock stat) ) ) ) ) )
        (while (!== stat pred-stat) )
        (do
            (if (stat:definition? pred-stat)
                (:= reaching-defs
                    (stat-set:union1
                        (stat-set:difference reaching-defs
                            (stat:killed-defs pred-stat) )
                        pred-stat) ) ) )

        (next pred-stat (stat:succ pred-stat) )
        (result
            (:= *rd.last-reaching-stat* stat)
            (:= *rd.last-reaching-defs* reaching-defs)
            reaching-defs) ) ) ) )


;***==========================================================================
;***
;*** (STAT:OPERAND-REACHING-DEFS STAT NAME)
;***
;***==========================================================================

(defun stat:operand-reaching-defs ( stat name )
    (stat-set:intersection (stat:reaching-defs stat)
                           (name:defining-stats name) ) )


;***==========================================================================
```



```
;***
;*** (STAT:KILLED-DEFS STAT)
;***
;***==========================================================================

(defun stat:killed-defs ( stat )
    (name:defining-stats (stat:part stat 'written) ) )
```

```
;=========================================================================
;
; REACHING USES
;
; This module sets :REACHING-USES of each STAT to be the set of STATs that
; use the value of the STAT.
;
; The reaching uses are calculated by inverting STAT:REACHING-DEFs, not
; by a separate flow-analysis pass.  A separate flow analysis pass would
; require a representation for uses and use-sets, and measurements show
; that this representation (lists of STATs stored in each STAT) will
; consume less space as long as the average number of uses reaching a
; STAT is less than about 4 (for most programs it appears to be 2).
; Besides, this was a lot easier to implement.
;
; (FG.INITIALIZE-REACHING-USES)
;     Clears the :REACHING-USES of each STAT in the flow graph.
;
; (FG.SET-REACHING-USES)
;     Calculates the :REACHING-USES of each STAT.
;
;=========================================================================

(include flow-analysis:flow-analysis-decls)


(defun fg.initialize-reaching-uses ()
    (loop (for-each-stat stat) (do
        (:= (stat:reaching-uses stat) () ) ) )
    () )


(defun fg.set-reaching-uses ()
    (fg.initialize-reaching-uses)

        ;*** For each STAT in the flow graph, STAT is pushed on the the
        ;*** list :REACHING-USES of the STATs whose definitions reach
        ;*** the given STAT and are actually used by it.  The STATs are
        ;*** enumerated in basic block order so that STAT:REACHING-DEFS
        ;*** will go fastest (due to its cache of previous results).
        ;
    (loop (for-each-bblock bblock) (do
        (loop (for-each-bblock-stat bblock stat)
              (bind used-reaching-defs
                    (stat-set:intersection
                        (stat:reaching-defs stat)
                        (loop (for-each-stat-operand-read stat name)
                              (reduce stat-set:union *fg.empty-stat-set*
                                      (name:defining-stats name) ) ) ) )
        (do
            (loop (for-each-stat-set-element used-reaching-defs def-stat) (do
                (push (stat:reaching-uses def-stat) stat) ) ) ) ) ) )

    () )
```

```
;===========================================================================
;
; Dead Code Removal
;
; (FG.REMOVE-ASSERTIONS)
;     Removes assertions and LOOP-ASSIGNs from the flow graph; a following
;     (FG.REMOVE-DEAD-CODE) will remove any support code needed for the
;     assertions.
;
; (FG.REMOVE-DEAD-CODE)
;     This function removes "dead code" from the flow graph.  First all
;     useless STATs are removed, then all unreachable BBLOCKs are removed.
;     A STAT is useless if:
;
;       1) It is a definition and its value isn't used in ultimately
;          producing the final live values of the flow graph.
;
;       2) It is a conditional jump both of whose branches jump to the
;          same spot.
;
;
; (FG.REMOVE-UNREACHABLE-BBLOCKS)
;     Removes all BBLOCKs from the flow graph that cannot be reached from
;     the entry point.
;
;===========================================================================

(include flow-analysis:flow-analysis-decls)

;***=========================================================================
;***
;*** (FG.REMOVE-ASSERTIONS)
;***
;***=========================================================================

(defun fg.remove-assertions ()
    (loop (for-each-stat stat)
          (when (memq (stat:operator stat) '(assert loop-assign) ) )
    (do
        (rdc.stat:delete stat) ) ) )



;***=========================================================================
;***
;*** (FG.REMOVE-DEAD-CODE)
;***
;***=========================================================================

(defun fg.remove-dead-code ()
    (loop (while (rdc.remove-useless-stats) ) )

    (fg.remove-unreachable-bblocks)
    () )



;***=========================================================================
;***
;*** (FG.REMOVE-UNREACHABLE-BBLOCKS)
;***
;***=========================================================================
```

```
(defun fg.remove-unreachable-bblocks ()
    (let ( (unprocessed () ) )

                ;*** Find all BBLOCKs that have no predecessors and are
                ;*** not the entry node.
                ;
        (loop (for-each-bblock bblock) (do
            (if (&& (!== bblock *fg.entry-bblock*)
                    (! (bblock:preds bblock) ) )
                (push unprocessed bblock) ) ) )

                ;*** While there are BBLOCKs with no predecessors, pick
                ;*** one and delete it, noticing if any of its successors now
                ;*** have no predecessors.
                ;
        (loop (while unprocessed)
              (initial bblock () )
        (do
            (pop unprocessed bblock)
            (bblock:delete bblock)
            (for (succ-bblock in (bblock:succs bblock) ) (do
                (if (&& (!== succ-bblock *fg.entry-bblock*)
                        (! (bblock:preds succ-bblock) ) )
                    (push unprocessed succ-bblock) ) ) ) ) )
        )

        ;*** Merge adjacent BBLOCKs that should really be one.
        ;
    (loop (for-each-bblock bblock) (do
        (loop (while (bblock:merge-with-successor bblock) ) ) ) )

    () )


;***=========================================================================
;***
;*** (RDC.REMOVE-USELESS-STATS)
;***
;*** Removes all useless STATs from the flow graph.  See above for the
;*** definition of "useless".
;***
;*** A mark-and-sweep algorithm is used to collect useless definitions.
;*** Marking begins with all non-definition STATs and works out following
;*** the reaching-definition chains.  Any unmarked STAT is then deleted.
;***
;*** Next any conditionals whose branches jump to the same spot are
;*** deleted.
;***
;*** Deleting a conditional jump could cause more definitions to become
;*** useless, causing more conditional jumps to become useless.  So this
;*** procedure is called repeatedly until no more conditional jumps are
;*** deleted (ugh).  It returns true iff at least one conditional jump
;*** was deleted.
;***
;***=========================================================================

(defun rdc.remove-useless-stats ()
    (let ( (stack () )
                ;
                ;*** A stack of statements that are useful but haven't
                ;*** been marked from yet.

           (visited *fg.empty-stat-set*)
```

```
            ;
            ;*** The set of all STATs that have been marked as useful
            ;*** so far.

        (deleted-cond-jump? () ) )
            ;
            ;*** True if we deleted a conditional jump.

        ;*** Initialize STACK to be all statements that aren't one-
        ;*** or two-in-one-out or vector references.
        ;
    (loop (for-each-stat stat)
          (when (! (memq (stat:group stat)
                         '(one-in-one-out two-in-one-out vstore vload
                           loop-assign induction-assign) ) ) )
    (do
        (push stack stat) ) )

        ;*** Now iteratively mark the reaching definitions of STATs
        ;*** that are on the stack, pushing the reaching definitions
        ;*** onto the stack for eventual recursive marking.  We never
        ;*** trace from INDUCTION-ASSIGNs (sigh).
        ;
    (loop (while stack)
          (bind stat (pop stack) )
          (when (! (stat-set:member? visited stat) ) )
    (do
        (:= visited (stat-set:union1 visited stat) )
        (loop (for-each-stat-operand-read stat name) (do
            (loop (for-each-stat-set-element
                      (stat:operand-reaching-defs stat name)
                      reaching-stat)
            (do
                (push stack reaching-stat) ) ) ) ) ) )

        ;*** Delete all unmarked STATs.
        ;
    (loop (for-each-stat-set-element
              (stat-set:difference (stat-set:universe) visited)
              stat)
    (do
        (rdc.stat:delete stat) ) )

        ;*** Now delete any cond-jumps whose left and right branches
        ;*** go to the same spot.
        ;
    (loop (for-each-stat stat)
          (when (stat:property? stat 'conditional-jump) )
          (bind succ-bblocks (bblock:succs (stat:bblock stat) ) )
          (when (|| (&& (== 2 (length succ-bblocks) )
                        (== (car succ-bblocks) (cadr succ-bblocks) ) )
                    (== 1 (length succ-bblocks) ) ) )
    (do
        (rdc.stat:delete stat)
        (:= deleted-cond-jump? t) ) )

    deleted-cond-jump?) )

;***=========================================================================
;***
;*** (RDC.STAT:DELETE STAT)
;***
```



```
;*** Deletes STAT from the flow graph;  if STAT's BBLOCK is now empty, the
;*** BBLOCK is also deleted.
;***
;***=========================================================================

(defun rdc.stat:delete ( stat )
    (assert (stat:is stat) )

    (let ( (bblock (stat:bblock stat) ) )
        (stat:delete stat)
        (if (bblock:empty? bblock) (then
            (bblock:delete bblock) ) )
        () ) )
```

```
;==========================================================================
;
;
; STAT
;
; Every NADDR operation in the flow-graph except GOTO and LABEL is
; represented by a STAT record.  All STATs are numbered and are stored
; in an array that maps the numbers onto the STATs.
;
;==================================
(def-struct stat                        ;
    source                              ; NADDR statement
    number                              ; Number of this statement.
    (bblock () suppress)                ; The basic block containing this statement.
    (succ   () suppress)                ; Successor STAT in the containing
                                        ;   BBLOCK (NIL if this is the last
                                        ;   in the BBLOCK).
    (pred   () suppress)                ; Predeccessor STAT in the containing
                                        ;   BBLOCK (NIL if this is the first
                                        ;   in the BBLOCK).
    (known-derivation () suppress)      ; Disambiguator derviation of the
                                        ;   variable defined by this STAT,
                                        ;   if already known.
    (known-operand-derivations          ; Association list of (VAR DERIVATION)
        () suppress)                    ;   containing already-known derivations
                                        ;   of operands of STAT.
    (reaching-uses () suppress)         ; List of STATs that use the value
    )                                   ;   defined by this STAT.
;==================================
;
; (STAT:CREATE SOURCE)
;    Creates a new STAT record for NADDR operation SOURCE.  This is
;    the only way STATs should be created.
;
; (STAT:SUBSTITUTE-OPERAND STAT NEW-OPERAND OLD-OPERAND PART)
;    Creates a new STAT from the source of STAT, substituting NEW-OPERAND
;    for OLD-OPERAND.   See OPER:SUBSTITUTE-OPERAND for how the substitution
;    occurs.
;
; (STAT:EXTRACT STAT)
;    Splices out STAT from the flow graph.  STAT is eligible to be
;    placed somewhere else in the flow graph.
;
; (STAT:DELETE STAT)
;    Extracts STAT from the flow graph and then forgets about it.
;
; (STAT:INSERT-STAT STAT1 STAT2)
;    Inserts STAT2 before STAT1 in the flow graph.
;
; (STAT:APPEND-STAT STAT1 STAT2)
;    Appends STAT2 after STAT1 in the flow graph.
;
; (STAT:REPLACE-STAT STAT1 STAT2)
;    Replaces STAT1 by STAT2 in the flow graph.
;
; (LOOP (FOR-EACH-STAT STAT) ...)
;    This is the only public way for enumerating through all the STATs
;    of the flow graph.  The enumeration is in order of the original
;    NADDR source.  Defined via LOOP's DEF-SIMPLE-LOOP-CLAUSE.
;
; (LOOP (FOR-EACH-STAT-OPERAND-READ STAT NAME) ...)
;    Enumerates NAME through each scalar and vector name read by by STAT.
;
```



```
; (NUMBER:STAT NUMBER )
;    Returns the STAT with the given NUMBER.  Try not to use this.
;    Deleted STATs return ().
;
; ##S <number.
;    This is syntax for easy interactive access to particular STATs.
;    ##s 30 references STAT number 30.
;
;
; (STAT:PREDS STAT)
; (STAT:SUCCS STAT)
;    List of STATs that are flow predecessors/successors of STAT.
;
; (STAT:OPERATOR  STAT)
; (STAT:GROUP     STAT)
; (STAT:PART      STAT PART)
; (STAT:PROPERTY? STAT PROPERTY)
;    Convenient functions that just invoke the equivalent functions
;    on the NADDR source of the STAT.
;
; (STAT:DEFINITION? STAT)
;    Returns true if STAT is a definition of a variable (or vector element).
;
; (STAT:DOMINATES? STAT1 STAT2)
;    True if STAT1 dominates STAT2.
;
;==========================================================================

(include flow-analysis:flow-analysis-decls)

(declare (special
    *fg.total-stats*              ;*** Current number of STATs
    *fg.number:stat*              ;*** Array for mapping STAT numbers onto
                                  ;*** STATs.
    *stat.free-list*              ;*** List of free slots in *FG.NUMBER:STATS*.
    *stat.free-list-reversed?*    ;*** True if above list has been reversed.
    ) )


(defun fg.initialize-stats ()
    (vector-map:initialize '*fg.number:stat* '*fg.total-stats* 200 t)
    (:= *stat.free-list*           () )
    (:= *stat.free-list-reversed?* () )
    () )


(defun stat:create ( source )
    (if *stat.free-list*  (then
        (if (! *stat.free-list-reversed?*) (then
            (:= *stat.free-list* (dreverse *stat.free-list*) )
            (:= *stat.free-list-reversed?* t) ) )
        (let ( (index (pop *stat.free-list*) ) )
            (:= ([] *fg.number:stat* index)
                (stat:new source source
                          number index) ) ) )
    (else
        (vector-map:add-element '*fg.number:stat* '*fg.total-stats*
            (stat:new source source
                      number *fg.total-stats*)
            200
            t) ) ) )
```

```
(defun stat:substitute-operand ( stat new-operand old-operand part )
    (assert (stat:is stat) )

    (stat:create (oper:substitute-operand (stat:source stat)
                                          new-operand
                                          old-operand
                                          part) ) )


(defun stat:extract ( stat )
    (assert (stat:is stat) )

    (let ( (bblock    (stat:bblock stat) )
           (succ-stat (stat:succ stat) )
           (pred-stat (stat:pred stat) ) )

                ;*** Remove STAT from the doubly linked list of STATs in
                ;*** the BBLOCK.
                ;
        (if succ-stat
            (:= (stat:pred succ-stat) pred-stat) )
        (if pred-stat
            (:= (stat:succ pred-stat) succ-stat) )
        (:= (stat:succ stat) () )
        (:= (stat:pred stat) () )

                ;*** Update the first and last pointers of the basic
                ;*** block.
                ;
        (? ( (== (bblock:first-stat bblock)
                 (bblock:last-stat  bblock) )
             (:= (bblock:first-stat bblock) () )
             (:= (bblock:last-stat  bblock) () ) )
           ( (== stat (bblock:first-stat bblock) )
             (:= (bblock:first-stat bblock) succ-stat) )
           ( (== stat (bblock:last-stat bblock) )
             (:= (bblock:last-stat bblock) pred-stat) ) )

        stat) )


(defun stat:delete ( stat )
    (assert (stat:is stat) )

    (stat:extract stat)

            ;*** Delete this STAT from the array of STATs.
    (:= ([] *fg.number:stat* (stat:number stat) ) () )
    (if *stat.free-list-reversed?* (then
        (:= *stat.free-list* (dreverse *stat.free-list*) )
        (:= *stat.free-list-reversed?* () ) ) )
    (push *stat.free-list* (stat:number stat) )

    () )


(defun stat:insert-stat ( stat1 stat2 )
    (assert (stat:is stat1) )
    (assert (stat:is stat2) )

    (let ( (bblock    (stat:bblock stat1) )
           (pred-stat (stat:pred   stat1) ) )
        (:= (stat:succ stat2) stat1)
```



```
        (:= (stat:pred stat1) stat2)
        (:= (stat:pred stat2) pred-stat)
        (if pred-stat (then
            (:= (stat:succ pred-stat) stat2) )
        (else
            (:= (bblock:first-stat bblock) stat2) ) )
        (:= (stat:bblock stat2) bblock)
        () ) )


(defun stat:append-stat ( stat1 stat2 )
    (assert (stat:is stat1) )
    (assert (stat:is stat2) )

    (let ( (bblock    (stat:bblock stat1) )
           (succ-stat (stat:succ   stat1) ) )
        (:= (stat:succ stat1) stat2)
        (:= (stat:pred stat2) stat1)
        (:= (stat:succ stat2) succ-stat)
        (if succ-stat (then
            (:= (stat:pred succ-stat) stat2) )
        (else
            (:= (bblock:last-stat bblock) stat2) ) )
        (:= (stat:bblock stat2) bblock)
        () ) )


(defun stat:replace-stat ( stat1 stat2 )
    (assert (stat:is stat1) )
    (assert (stat:is stat2) )

    (stat:insert-stat stat1 stat2)
    (stat:delete stat1)
    () )


(def-simple-loop-clause for-each-stat ( clause )
    (let ( ( (for-each-stat var) clause)
           (index (intern (gensym) ) ) )

        (if (! (&& (= 2 (length clause) )
                   (litatom var) ) )
            (error (list clause "Invalid FOR-EACH-STAT syntax.") ) )

        `( (initial ,var () )
           (incr ,index from 0 to (+ -1 *fg.total-stats*) )
           (next ,var ([] *fg.number:stat* ,index) )
           (when ,var) ) ) )


(def-simple-loop-clause for-each-stat-operand-read ( clause )
    (let ( ( (for-each-stat-operand-read stat var index) clause) )
        `( (for-each-oper-operand-read (stat:source ,stat) ,var ,index) ) ) )


(defmacro number:stat ( number )
    `([] *fg.number:stat* ,number) )

(def-sharp-sharp s
    `([] *fg.number:stat* ,(read) ) )


(defun stat:preds ( stat )
```

```
    (assert (stat:is stat) )
    (let ( (bblock (stat:bblock stat) ) )
        (if (== stat (bblock:first-stat bblock) ) (then
            (loop (for pred-bblock in (bblock:preds bblock) ) (save
                (bblock:last-stat pred-bblock) ) ) )
        (else
            (list (stat:pred stat) ) ) ) ) )


(defun stat:succs ( stat )
    (assert (stat:is stat) )
    (let ( (bblock (stat:bblock stat) ) )
        (if (== stat (bblock:last-stat bblock) ) (then
            (loop (for succ-bblock in (bblock:succs bblock) ) (save
                (bblock:first-stat succ-bblock) ) ) )
        (else
            (list (stat:succ stat) ) ) ) ) )


(defun stat:operator ( stat )
    (assert (stat:is stat) )
    (oper:operator (stat:source stat) ) )

(defun stat:group ( stat )
    (assert (stat:is stat) )
    (oper:group (stat:source stat) ) )

(defun stat:part ( stat part )
    (assert (stat:is stat) )
    (oper:part (stat:source stat) part) )

(defun stat:property? ( stat property )
    (assert (stat:is stat) )
    (oper:property? (stat:source stat) property) )

(defun stat:definition? ( stat )
    (stat:part stat 'written) )


(defun stat:dominates? ( stat1 stat2 )
    (assert (stat:is stat1) )
    (assert (stat:is stat2) )
    (bblock:dominates? (stat:bblock stat1)
                       (stat:bblock stat2) ) )
```

```
;=========================================================================
;
; STAT-SETS
;
; Sets of STATs are represented using STAT-SETs, currently implemented
; as BIT-SETs.
;
; *FG.EMPTY-STAT-SET*
;     The empty STAT-SET.
;
; (STAT-SET:UNIVERSE)
;     The set of all STATs.
;
; (STAT-SET:SINGLETON STAT)
;     Creates a new set containing STAT.
;
; (STAT-SET:MEMBER? SET STAT)
;     Returns true if STAT is a member of SET.
;
; (STAT-SET:INTERSECTION SET1 SET2 ...)
;     Returns a new set that is the intersection of all the given sets.
;
; (STAT-SET:UNION SET1 SET2 ...)
;     Returns a new set that is the union of all the given sets.
;
; (STAT-SET:UNION1 SET STAT)
;     Unions a single STAT into SET.
;
; (STAT-SET:DIFFERENCE SET1 SET2)
;     Returns a new set that contains all elements in SET1 not in SET2.
;
; (STAT-SET:= SET1 SET2)
;     Returns true if the two sets are equal.
;
; (STAT-SET:SIZE SET)
;     Returns the number of elements in the set.
;
; (STAT-SET:CHOOSE SET)
;     Returns the first element of the set, NIL if the set is empty.
;
; (STAT-SET:CONTAINS? SET1 SET2)
;     True if SET1 contains SET2.
;
; (LOOP (FOR-EACH-STAT-SET-ELEMENT SET STAT)
;     Enumerates STAT through each element in SET.  Uses
;     DEF-SIMPLE-LOOP-CLAUSE.
;
; (STAT-SET:PRINT SET)
;    Prints SET by printing out the statement numbers.
;
;=========================================================================

(include flow-analysis:flow-analysis-decls)

(declare (special *fg.total-stats*) )

(defvar *fg.empty-stat-set* () )    ;*** the empty STAT-SET.

(defun stat-set:universe ()
    (bit-set:universe *fg.total-stats*) )

(defun stat-set:singleton ( stat )
    (bit-set:singleton (stat:number stat) ) )
```



```
(defun stat-set:member? ( set stat )
    (bit-set:member? set (stat:number stat) ) )

(defun stat-set:intersection args
    (apply 'bit-set:intersection (listify-lexpr-args args) ) )

(defun stat-set:union args
    (apply 'bit-set:union (listify-lexpr-args args) ) )

(defun stat-set:union1 ( set stat )
    (bit-set:union1 set (stat:number stat) ) )

(defun stat-set:difference ( set1 set2 )
    (bit-set:difference set1 set2) )

(defun stat-set:= ( set1 set2 )
    (bit-set:= set1 set2) )

(defun stat-set:size ( set )
    (bit-set:size set) )

(defun stat-set:choose ( set )
    (if-let ( (index (bit-set:choose set) ) )
        (number:stat index)
        () ) )

(defun stat-set:contains? ( set1 set2 )
    (bit-set:contains? set1 set2) )


(def-simple-loop-clause for-each-stat-set-element ( clause )
    (let ( ( (for-each-stat-set-element set stat) clause)
           (index (intern (gensym) ) ) )
        (if (! (&& (= 3 (length clause) )
                   (litatom stat) ) )
            (error (list clause "Invalid FOR-EACH-STAT-SET-ELEMENT syntax.")))

        `( (initial ,stat () )
           (for-each-bit-set-element ,set ,index)
           (next ,stat (number:stat ,index) )
           (when ,stat) ) ) )

(defun stat-set:print ( set )
    (bit-set:print set) )
```

```
;===========================================================================
;
; This module prints miscellaneous statistics about the flow graph.
;
; (FG.PRINT-STATISTICS)
;    Prints out whatever statistics we want today.
;
;===========================================================================

(include flow-analysis:flow-analysis-decls)



(defun fg.print-statistics ()
    (msg 0)

    (msg "# of iterations in REACHING-DEFS: "
         *fg.number-of-reaching-iterations*
         t t)

    (msg "# of BBLOCKs: " *fg.total-bblocks* t)
    (msg "# of STATs:   " *fg.total-stats* t)
    (msg "STATs//BBLOCK:    "
         (// (flonum *fg.total-stats*) (flonum *fg.total-bblocks*) )
         t)
    (loop (for-each-stat stat)
          (initial ud-length 0)
    (do
        (loop (initial used-defs *fg.empty-stat-set*)
        (for-each-stat-operand-read stat name)
        (next used-defs (stat-set:union used-defs
                                        (name:defining-stats name) ) )
        (result
            (:= ud-length
                (+ ud-length
                   (bit-set:size (bit-set:intersection
                                    used-defs
                                    (stat:reaching-defs stat) ) ) ) ) ) ) ) )
    (result
        (msg "Average length of ud-chain//STAT: "
             (// (flonum ud-length) (flonum *fg.total-stats*) )
             t t) ) )

    () )
```

```
;===========================================================================
;
; (FG.TEMPORARY-NAME ROOT)
;     Returns a new, uninterned unique temporary name of the form:
;
;       $ ROOT - n
;
;     where n is a unique number.  E.g. if ROOT = 'FOO then the result
;     might be $FOO-2.  ROOT may be (), which is the same as the empty
;     string.  If ROOT is of the form $NAME-n, then NAME is used as the
;     the root.  Thus if ROOT = '$FOO-49, the result might be $FOO-100.
;
; (FG.INITIALIZE-TEMPORARY-NAME)
;     Initializes the creation of temporary names by resetting the unique
;     number counter to 1.
;
;===========================================================================

(defvar *fg.temporary-name-counter* 0)


(defun fg.initialize-temporary-name ()
    (:= *fg.temporary-name-counter* 0) )


(defun fg.temporary-name ( root )
    (?( (! root)
        (atomconcat "$" (++ *fg.temporary-name-counter*) ) )

      ( (!== #/$ (anthchar root 1) )
        (atomconcat "$" root "-" (++ *fg.temporary-name-counter*) ) )

      ( t
        (loop (incr i from 1 to (stlength root) )
              (until (== #/- (anthchar root i) ) )
        (result (atomconcat "$"
                            (substring root 2 (- i 1) )
                            "-"
                            (++ *fg.temporary-name-counter*) ) ) ) ) ) )
```

```
;==========================================================================
;
; (FG.RENAME-VARIABLES)
;
; Renames variables as much as possible to avoid false data dependencies.
;
; The uses and defs of a variable can be veiwed as bipartite graph.  Every
; use and def is separate node.  There is an edge between a def and a use
; if the def reaches the use.  Within each connected component of this graph,
; the variable can be given a new unique name.  More precisely, the reads
; of the variable in uses and the writes of the variables within defs may
; be changed to use a new variable.
;
; We don't actually construct the graph, but instead pick a def of a variable
; and then trace out the connected component use-def chain.
;
;==========================================================================

(eval-when (compile load)
    (include flow-analysis:flow-analysis-decls) )

(defun fg.rename-variables ()

(let ( (renamings () ) )
            ;
            ;*** A list of triples of the form:
            ;***
            ;***    (STAT-SET NEW-NAME OLD-NAME READ-OR-WRITTEN)
            ;***
            ;*** that specifies that NEW-NAME should be substituted for
            ;*** OLD-NAME in the READ or WRITTEN part of every STAT in
            ;*** STAT-SET.  We construct the whole set of renamings for
            ;*** the program before doing any renaming; otherwise
            ;*** STAT:REACHING-DEFS will get horribly confused by a
            ;*** partially renamed program.

        ;*** For each NAME do:
        ;
    (loop (for-each-name name)
          (when (== 'scalar (name:type name) ) )
          (bind all-visited-defs *fg.empty-stat-set*)
    (do

            ;*** Pick an unvisited defining DEF-STAT of NAME.
            ;
        (loop (for-each-stat-set-element (name:defining-stats name) def-stat)
              (when (! (stat-set:member? visited-defs def-stat) ) )
              (initial name-counter        0)
              (bind    trace-stack         ()
                       visited-uses        *fg.empty-stat-set*
                       visited-defs        *fg.empty-stat-set*
                       contains-pseudo-op? () )
        (do
                ;*** Trace out the connected chain of uses and defs of
                ;*** NAME containing DEF-STAT.  To do the tracing, we
                ;*** keep a stack TRACE-STACK of pairs of the form (STAT
                ;*** READ-OR-WRITTEN), where STAT is a possibly unvisited
                ;*** use or def of NAME, and READ-OR-WRITTEN indicates
                ;*** whether STAT is being considered as a use or def
                ;*** of NAME.  At the end of this tracing, VISITED-USES
                ;*** and VISITED-DEFS will contain STAT-SETs of all the
                ;*** uses and defs that are part of the connected train
                ;*** that was traced out.
```

1

```
                ;
        (push trace-stack '(,def-stat written) )
        (loop (while trace-stack)
              (bind (stat read-or-written) (pop trace-stack) )
              (when (! (stat-set:member? (if (== 'written read-or-written)
                                             visited-defs
                                             visited-uses)
                                         stat) ) )
        (do
                ;*** If this STAT is a special pseudo-op, remember
                ;*** it so as to disable renaming of this chain.
                ;
            (if (memq (stat:operator stat) '(def-block live param) ) (then
                (:= contains-pseudo-op? t) ) )

                ;*** Mark the use or def as visited, and trace out
                ;*** reaching uses (defs) of this def (use).
                ;
            (if (== 'written read-or-written) (then
                (:= visited-defs (stat-set:union1 &&& stat) )
                (loop (for use-stat in (stat:reaching-uses stat) ) (do
                    (push trace-stack '(,use-stat read) ) ) ) )

            (else
                (:= visited-uses (stat-set:union1 &&& stat) )
                (loop (for-each-stat-set-element
                          (stat:operand-reaching-defs stat name)
                          reaching-def-stat)
                (do
                    (push trace-stack '(,reaching-def-stat written)))))))

            ;*** The tracing is finished.  Remember all the visited
            ;*** defs of NAME.
            ;
        (:= all-visited-defs (stat-set:union &&& visited-defs) )

            ;*** Make an entry on the RENAMINGS list if this is not the
            ;*** first chain of NAME and the chain contains no pseudo-op.
            ;
        (if (&& (> (++ name-counter) 1)
                (! contains-pseudo-op?) )
        (then
            (let ( (new-name (fg.temporary-name name) ) )
                (push renamings '(,visited-uses ,new-name ,name read) )
                (push renamings '(,visited-defs ,new-name ,name written)))))
        ) ) ) )


    ;*** All the partitions (chains) of every NAME have been found.
    ;*** Now do the actual renaming as specified by the RENAMINGS list.
    ;
(loop (for (stat-set new-name old-name read-or-written) in renamings) (do
    (loop (for-each-stat-set-element stat-set stat) (do
        (:= (stat:source stat)
            (oper:substitute-operand (stat:source stat)
                                     new-name
                                     old-name
                                     read-or-written) ) ) ) ) )

() ) )
```

2