```
#|

 Copyright (C) 1994 by Computational Logic, Inc.  All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Computational Logic, Inc. PROVIDES ABSOLUTELY NO WARRANTY.  THE EVENT SCRIPT
 IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,
 INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE DEFECTIVE, YOU
 ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 IN NO EVENT WILL Computational Logic, Inc. BE LIABLE TO YOU FOR ANY DAMAGES,
 ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT
 NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
 SUSTAINED BY THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

|#

; ------------------------------------------------------------
; was bags.events
; ------------------------------------------------------------
```

EVENT: Start with the initial **nqthm** theory.

DEFINITION:
delete $(x,\ l)$
= **if** listp (l)
then if $x =$ car (l) **then** cdr (l)
else cons (car (l), delete $(x,$ cdr $(l)))$ **endif**
else l **endif**

DEFINITION:
bagdiff $(x,\ y)$
= **if** listp (y)
then if car $(y) \in x$ **then** bagdiff (delete (car (y), $x)$, cdr (y))
else bagdiff $(x,$ cdr $(y))$ **endif**
else x **endif**

DEFINITION:
bagint $(x,\ y)$
$=$ **if** listp (x)
then if car $(x) \in y$
then cons (car (x), bagint (cdr (x), delete (car (x), y))))
else bagint (cdr (x), y) **endif**
else nil endif

DEFINITION:
occurrences $(x,\ l)$
$=$ **if** listp (l)
then if $x =$ car (l) **then** $1 +$ occurrences $(x,$ cdr $(l))$
else occurrences $(x,$ cdr $(l))$ **endif**
else 0 **endif**

DEFINITION:
subbagp $(x,\ y)$
$=$ **if** listp (x)
then if car $(x) \in y$ **then** subbagp (cdr (x), delete (car (x), y))
else f endif
else t endif

THEOREM: listp-delete
listp (delete $(x,\ l)$)
$=$ **if** listp (l) **then** $(x \neq$ car $(l)) \vee$ listp (cdr (l))
else f endif

EVENT: Disable listp-delete.

THEOREM: delete-non-member
$(x \notin y) \rightarrow$ (delete $(x,\ y) = y$)

THEOREM: delete-delete
delete $(y,$ delete $(x,\ z)) =$ delete $(x,$ delete $(y,\ z))$

THEOREM: equal-occurrences-zero
(occurrences $(x,\ l) =$ 0) $= (x \notin l)$

THEOREM: member-non-list
$(\neg$ listp $(l)) \rightarrow (x \notin l)$

THEOREM: member-delete
$(x \in$ delete $(y,\ l))$
$=$ **if** $x \in l$
then if $x = y$ **then** $1 <$ occurrences $(x,\ l)$
else t endif
else f endif

THEOREM: member-delete-implies-membership
$(x \in \text{delete}(y, l)) \rightarrow (x \in l)$

THEOREM: occurrences-delete
$\text{occurrences}(x, \text{delete}(y, l))$
$=$ **if** $x = y$
then if $x \in l$ **then** $\text{occurrences}(x, l) - 1$
else 0 **endif**
else $\text{occurrences}(x, l)$ **endif**

THEOREM: member-bagdiff
$(x \in \text{bagdiff}(a, b)) = (\text{occurrences}(x, b) < \text{occurrences}(x, a))$

THEOREM: bagdiff-delete
$\text{bagdiff}(\text{delete}(e, x), y) = \text{delete}(e, \text{bagdiff}(x, y))$

THEOREM: subbagp-delete
$\text{subbagp}(x, \text{delete}(u, y)) \rightarrow \text{subbagp}(x, y)$

THEOREM: subbagp-cdr1
$\text{subbagp}(x, y) \rightarrow \text{subbagp}(\text{cdr}(x), y)$

THEOREM: subbagp-cdr2
$\text{subbagp}(x, \text{cdr}(y)) \rightarrow \text{subbagp}(x, y)$

THEOREM: subbagp-bagint1
$\text{subbagp}(\text{bagint}(x, y), x)$

THEOREM: subbagp-bagint2
$\text{subbagp}(\text{bagint}(x, y), y)$

THEOREM: occurrences-bagint
$\text{occurrences}(x, \text{bagint}(a, b))$
$=$ **if** $\text{occurrences}(x, a) < \text{occurrences}(x, b)$ **then** $\text{occurrences}(x, a)$
else $\text{occurrences}(x, b)$ **endif**

THEOREM: occurrences-bagdiff
$\text{occurrences}(x, \text{bagdiff}(a, b)) = (\text{occurrences}(x, a) - \text{occurrences}(x, b))$

THEOREM: member-bagint
$(x \in \text{bagint}(a, b)) = ((x \in a) \wedge (x \in b))$

EVENT: Let us define the theory *bags* to consist of the following events: occurrences-bagint, bagdiff-delete, occurrences-bagdiff, member-bagint, member-bagdiff, subbagp-bagint2, subbagp-bagint1, subbagp-cdr2, subbagp-cdr1, subbagp-delete.

```
; ------------------------------------------------------------
; was naturals.events
; ------------------------------------------------------------


;; Tue Sep 26 10:20:45 1989, from ~wilding/numerical/newnat.events

;; NATURALS Theory

;; Created by Bill Bevier 1988 (see CLI internal note 057)

;; Modifications by Bill Bevier and Matt Wilding (9/89) including
;; adding some new metalemmas for times, reorganizing the theories,
;; removing some extraneous lemmas, and removing dependence upon
;; other theories (by adding the pertinent lemmas).

;; This script requires the bags theory

;; This script sets up a theory for the NATURALS with the following subtheories
;; ADDITION
;; MULTIPLICATION
;; REMAINDER
;; QUOTIENT
;; EXPONENTIATION
;; LOGS
;; GCDS

;; The theories of EXPONENTIATION, LOGS, and GCDS still need a lot of work

; ------------------------------------------------------------------------------
; ARITHMETIC
; ------------------------------------------------------------------------------

; -------------------- PLUS & DIFFERENCE --------------------

; ---------- EQUAL ----------
```

THEOREM: equal-plus-0
$((a + b) = 0) = ((a \simeq 0) \wedge (b \simeq 0))$

THEOREM: plus-cancellation
$((a + b) = (a + c)) = (\mathrm{fix}\,(b) = \mathrm{fix}\,(c))$

EVENT: Disable plus-cancellation.

THEOREM: equal-difference-0
$(((x - y) = \mathtt{0}) = (y \not< x)) \wedge ((\mathtt{0} = (x - y)) = (y \not< x))$

THEOREM: difference-cancellation
$((x - y) = (z - y))$
$=$ **if** $x < y$ **then** $y \not< z$
elseif $z < y$ **then** $y \not< x$
else $\mathrm{fix}\,(x) = \mathrm{fix}\,(z)$ **endif**

EVENT: Disable difference-cancellation.

```
; ---------- PLUS ----------
```

THEOREM: commutativity-of-plus
$(x + y) = (y + x)$

THEOREM: commutativity2-of-plus
$(x + (y + z)) = (y + (x + z))$

THEOREM: plus-zero-arg2
$(y \simeq \mathtt{0}) \rightarrow ((x + y) = \mathrm{fix}\,(x))$

THEOREM: plus-add1-arg1
$((1 + a) + b) = (1 + (a + b))$

THEOREM: plus-add1-arg2
$(x + (1 + y))$
$=$ **if** $y \in \mathbf{N}$ **then** $1 + (x + y)$
else $1 + x$ **endif**

THEOREM: associativity-of-plus
$((x + y) + z) = (x + (y + z))$

THEOREM: plus-difference-arg1
$((a - b) + c)$
$=$ **if** $b < a$ **then** $(a + c) - b$
else $\mathtt{0} + c$ **endif**

THEOREM: plus-difference-arg2
$(a + (b - c))$
$=$ **if** $c < b$ **then** $(a + b) - c$
else $a + \mathtt{0}$ **endif**

```
; ---------- DIFFERENCE-PLUS cancellation rules ----------
;
; Here are the basic canonicalization rules for differences of sums. These
; are subsumed by the meta lemmas and are therefore globally disabled.
; They are here merely to prove the meta lemmas.
```

THEOREM: difference-plus-cancellation-proof
$((x + y) - x) = \text{fix}\,(y)$

THEOREM: difference-plus-cancellation
$(((x + y) - x) = \text{fix}\,(y)) \wedge (((y + x) - x) = \text{fix}\,(y))$

EVENT: Disable difference-plus-cancellation.

THEOREM: difference-plus-plus-cancellation-proof
$((x + y) - (x + z)) = (y - z)$

THEOREM: difference-plus-plus-cancellation
$(((x + y) - (x + z)) = (y - z))$
$\wedge \quad (((y + x) - (x + z)) = (y - z))$
$\wedge \quad (((x + y) - (z + x)) = (y - z))$
$\wedge \quad (((y + x) - (z + x)) = (y - z))$

EVENT: Disable difference-plus-plus-cancellation.

THEOREM: difference-plus-plus-cancellation-hack
$((w + x + a) - (y + z + a)) = ((w + x) - (y + z))$

EVENT: Disable difference-plus-plus-cancellation-hack.

```
; Here are a few more facts about difference needed to prove the meta lemmas.
; These are disabled here. We re-prove them after the proof of the meta
; lemmas so that they will fire before the meta lemmas in subsequent proofs.
```

THEOREM: diff-sub1-arg2
$(a - (b - 1))$
$=$ **if** $b \simeq 0$ **then** $\text{fix}\,(a)$
elseif $a < b$ **then** 0
else $1 + (a - b)$ **endif**

EVENT: Disable diff-sub1-arg2.

THEOREM: diff-diff-arg1
$((x - y) - z) = (x - (y + z))$

THEOREM: diff-diff-arg2
$(a - (b - c))$
$=$ **if** $b < c$ **then** fix (a)
else $(a + c) - b$ **endif**

```
; diff-diff-diff should be removed, but since the hack lemmas for
; correctness-of-cancel-difference-plus are designed for it, we'll
; keep it around.
```

THEOREM: diff-diff-diff
$((b \leq a) \wedge (d \leq c))$
$\rightarrow$ $(((a - b) - (c - d)) = ((a + d) - (b + c)))$

EVENT: Disable diff-diff-diff.

THEOREM: difference-lessp-arg1
$(a < b) \rightarrow ((a - b) = 0)$

EVENT: Disable difference-lessp-arg1.

```
; ------------------------------------------------------------------------------
; Meta Lemmas to Cancel PLUS and DIFFERENCE expressions
; ------------------------------------------------------------------------------

; ---------- PLUS-TREE and PLUS-FRINGE ----------
```

DEFINITION:
plus-fringe (x)
$=$ **if** listp $(x) \wedge$ (car $(x) =$ `'plus`)
then append (plus-fringe (cadr (x)), plus-fringe (caddr (x)))
else cons $(x$, **nil**$)$ **endif**

DEFINITION:
plus-tree (l)
$=$ **if** $l \simeq$ **nil then** `''0`
elseif cdr $(l) \simeq$ **nil then** list (`'fix`, car (l))
elseif cddr $(l) \simeq$ **nil then** list (`'plus`, car (l), cadr (l))
else list (`'plus`, car (l), plus-tree (cdr (l))) **endif**

THEOREM: numberp-eval$-plus
$(\text{listp}\,(x) \wedge (\text{car}\,(x) = \texttt{'plus})) \rightarrow (\text{eval\$}\,(\mathbf{t},\ x,\ a) \in \mathbf{N})$

EVENT: Disable numberp-eval$-plus.

THEOREM: numberp-eval$-plus-tree
$\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(l),\ a) \in \mathbf{N}$

EVENT: Disable numberp-eval$-plus-tree.

THEOREM: member-implies-plus-tree-greatereqp
$(x \in y) \rightarrow (\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(y),\ a) \not< \text{eval\$}\,(\mathbf{t},\ x,\ a))$

EVENT: Disable member-implies-plus-tree-greatereqp.

THEOREM: plus-tree-delete
$\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(\text{delete}\,(x,\ y)),\ a)$
$=$ **if** $x \in y$ **then** $\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(y),\ a) - \text{eval\$}\,(\mathbf{t},\ x,\ a)$
else $\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(y),\ a)$ **endif**

EVENT: Disable plus-tree-delete.

THEOREM: subbagp-implies-plus-tree-greatereqp
$\text{subbagp}\,(x,\ y) \rightarrow (\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(y),\ a) \not< \text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(x),\ a))$

EVENT: Disable subbagp-implies-plus-tree-greatereqp.

THEOREM: plus-tree-bagdiff
$\text{subbagp}\,(x,\ y)$
$\rightarrow$ $(\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(\text{bagdiff}\,(y,\ x)),\ a)$
$=$ $(\text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(y),\ a) - \text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(x),\ a)))$

EVENT: Disable plus-tree-bagdiff.

THEOREM: numberp-eval$-bridge
$(\text{eval\$}\,(\mathbf{t},\ z,\ a) = \text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(x),\ a)) \rightarrow (\text{eval\$}\,(\mathbf{t},\ z,\ a) \in \mathbf{N})$

EVENT: Disable numberp-eval$-bridge.

THEOREM: bridge-to-subbagp-implies-plus-tree-greatereqp
$(\text{subbagp}\,(y,\ \text{plus-fringe}\,(z))$
$\wedge$ $(\text{eval\$}\,(\mathbf{t},\ z,\ a) = \text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(\text{plus-fringe}\,(z)),\ a)))$
$\rightarrow$ $((\text{eval\$}\,(\mathbf{t},\ z,\ a) < \text{eval\$}\,(\mathbf{t},\ \text{plus-tree}\,(y),\ a)) = \mathbf{f})$

EVENT: Disable bridge-to-subbagp-implies-plus-tree-greatereqp.

THEOREM: eval$-plus-tree-append
eval$ (**t**, plus-tree (append (x, y)), a)
= (eval$ (**t**, plus-tree (x), a) + eval$ (**t**, plus-tree (y), a))

EVENT: Disable eval$-plus-tree-append.

THEOREM: plus-tree-plus-fringe
eval$ (**t**, plus-tree (plus-fringe (x)), a) = fix (eval$ (**t**, x, a))

EVENT: Disable plus-tree-plus-fringe.

THEOREM: member-implies-numberp
((c ∈ plus-fringe (x)) ∧ (eval$ (**t**, c, a) ∈ **N**)) → (eval$ (**t**, x, a) ∈ **N**)

EVENT: Disable member-implies-numberp.

THEOREM: cadr-eval$-list
(car (eval$ (`'list`, x, a)) = eval$ (**t**, car (x), a))
∧ (cdr (eval$ (`'list`, x, a))
= **if** listp (x) **then** eval$ (`'list`, cdr (x), a)
else `0` **endif**)

EVENT: Disable cadr-eval$-list.

THEOREM: eval$-quote
eval$ (**t**, cons (`'quote`, *args*), a) = car (*args*)

EVENT: Disable eval$-quote.

THEOREM: listp-eval$
listp (eval$ (`'list`, x, a)) = listp (x)

EVENT: Disable listp-eval$.

```
; ---------- CANCEL PLUS ----------

; CANCEL-EQUAL-PLUS cancels identical terms in a term which is the equality
; of two sums. For example,
;
;   (EQUAL (PLUS A B C) (PLUS B D E)) => (EQUAL (PLUS A C) (PLUS D E))
```

;

DEFINITION:
cancel-equal-plus (x)
= **if** listp $(x) \wedge$ (car (x) = 'equal)
then if listp (cadr (x))
$\wedge$ (caadr (x) = 'plus)
$\wedge$ listp (caddr (x))
$\wedge$ (caaddr (x) = 'plus)
then list ('equal,
plus-tree (bagdiff (plus-fringe (cadr (x)),
bagint (plus-fringe (cadr (x)),
plus-fringe (caddr (x))))),
plus-tree (bagdiff (plus-fringe (caddr (x)),
bagint (plus-fringe (cadr (x)),
plus-fringe (caddr (x)))))
elseif listp (cadr (x))
$\wedge$ (caadr (x) = 'plus)
$\wedge$ (caddr $(x) \in$ plus-fringe (cadr (x)))
then list ('if,
list ('numberp, caddr (x)),
list ('equal,
plus-tree (delete (caddr (x), plus-fringe (cadr (x)))),
''0),
list ('quote, **f**))
elseif listp (caddr (x))
$\wedge$ (caaddr (x) = 'plus)
$\wedge$ (cadr $(x) \in$ plus-fringe (caddr (x)))
then list ('if,
list ('numberp, cadr (x)),
list ('equal,
''0,
plus-tree (delete (cadr (x), plus-fringe (caddr (x))))),
list ('quote, **f**))
else x **endif**
else x **endif**

THEOREM: correctness-of-cancel-equal-plus
eval$ (**t**, x, a) = eval$ (**t**, cancel-equal-plus (x), a)

```
; ---------- CANCEL-DIFFERENCE-PLUS ----------

; CANCEL-DIFFERENCE-PLUS cancels identical terms in a term which is the
; difference of two sums. For example,
```

```
;
;  (DIFFERENCE (PLUS A B C) (PLUS B D E)) => (DIFFERENCE (PLUS A C) (PLUS D E))
;
; Using rewrite rules, we canonicalize terms involving  PLUS and DIFFERENCE
; to be the DIFFERENCE of two sums. Then CANCEL-DIFFERENCE-PLUS cancels out
; like terms.
```

DEFINITION:
cancel-difference-plus (x)
= **if** listp $(x) \wedge$ (car (x) = `'difference`)
then if listp (cadr (x))
$\wedge$ (caadr (x) = `'plus`)
$\wedge$ listp (caddr (x))
$\wedge$ (caaddr (x) = `'plus`)
then list (`'difference`,
plus-tree (bagdiff (plus-fringe (cadr (x)),
bagint (plus-fringe (cadr (x)),
plus-fringe (caddr (x)))))),
plus-tree (bagdiff (plus-fringe (caddr (x)),
bagint (plus-fringe (cadr (x)),
plus-fringe (caddr (x)))))))
elseif listp (cadr (x))
$\wedge$ (caadr (x) = `'plus`)
$\wedge$ (caddr $(x) \in$ plus-fringe (cadr (x)))
then plus-tree (delete (caddr (x), plus-fringe (cadr (x))))
elseif listp (caddr (x))
$\wedge$ (caaddr (x) = `'plus`)
$\wedge$ (cadr $(x) \in$ plus-fringe (caddr (x))) **then** `''0`
else x **endif**
else x **endif**

THEOREM: correctness-of-cancel-difference-plus
eval$ (**t**, x, a) = eval$ (**t**, cancel-difference-plus (x), a)

```
; ---------- DIFFERENCE ----------

; Here are the rules for difference terms which we want to try before
; the meta lemmas. They help canonicalize terms to differences of sums.
```

THEOREM: difference-elim
$((y \in \mathbf{N}) \wedge (y \not< x)) \rightarrow ((x + (y - x)) = y)$

THEOREM: difference-leq-arg1
$(a \leq b) \rightarrow ((a - b) = 0)$

THEOREM: difference-add1-arg2
$(a - (1 + b))$
$=$ **if** $b < a$ **then** $(a - b) - 1$
else 0 **endif**

THEOREM: difference-sub1-arg2
$(a - (b - 1))$
$=$ **if** $b \simeq 0$ **then** fix (a)
elseif $a < b$ **then** 0
else $1 + (a - b)$ **endif**

THEOREM: difference-difference-arg1
$((x - y) - z) = (x - (y + z))$

THEOREM: difference-difference-arg2
$(a - (b - c))$
$=$ **if** $b < c$ **then** fix (a)
else $(a + c) - b$ **endif**

THEOREM: difference-x-x
$(x - x) = 0$

```
; ---------- LESSP ----------
```

THEOREM: lessp-difference-cancellation
$((a - c) < (b - c))$
$=$ **if** $c \leq a$ **then** $a < b$
else $c < b$ **endif**

EVENT: Disable lessp-difference-cancellation.

```
; CANCEL-LESSP-PLUS cancels LESSP terms whose arguments are sums.
; Examples:
; (LESSP (PLUS A B C) (PLUS A C D)) -> (LESSP (FIX B) (FIX D))
; (LESSP A (PLUS A B))  -> (NOT (ZEROP (FIX B)))
; (LESSP (PLUS A B) A)  -> F
```

DEFINITION:
cancel-lessp-plus (x)
$=$ **if** listp $(x) \wedge$ (car $(x) =$ 'lessp)
then if listp (cadr (x))
$\wedge$ (caadr $(x) =$ 'plus)
$\wedge$ listp (caddr (x))

```
        ∧   (caaddr (x) = 'plus)
      then list ('lessp,
                 plus-tree (bagdiff (plus-fringe (cadr (x)),
                                     bagint (plus-fringe (cadr (x)),
                                             plus-fringe (caddr (x))))),
                 plus-tree (bagdiff (plus-fringe (caddr (x)),
                                     bagint (plus-fringe (cadr (x)),
                                             plus-fringe (caddr (x))))))
      elseif listp (cadr (x))
             ∧   (caadr (x) = 'plus)
             ∧   (caddr (x) ∈ plus-fringe (cadr (x)))
      then list ('quote, f)
      elseif listp (caddr (x))
             ∧   (caaddr (x) = 'plus)
             ∧   (cadr (x) ∈ plus-fringe (caddr (x)))
      then list ('not,
                 list ('zerop,
                       plus-tree (delete (cadr (x), plus-fringe (caddr (x))))))
      else x endif
  else x endif
```

THEOREM: correctness-of-cancel-lessp-plus
eval$ (**t**, x, a) = eval$ (**t**, cancel-lessp-plus (x), a)

```
; Define the available theory of addition.  To get the list of events to
; put in the theory, evaluate the following form in NQTHM at this point
; in the script. This form lists all  lemmas which are globally enabled,
; and which have non-null lemma type.
;
; (remove-if-not (function (lambda (x)
;     (and (member x (lemmas))
;  (not (assoc x disabled-lemmas))
;  (not (null (nth 2 (get x 'event)))))))
;         chronology)
```

EVENT: Let us define the theory *addition* to consist of the following events: equal-plus-0, equal-difference-0, commutativity-of-plus, commutativity2-of-plus, plus-zero-arg2, plus-add1-arg2, plus-add1-arg1, associativity-of-plus, plus-difference-arg1, plus-difference-arg2, diff-diff-arg1, diff-diff-arg2, correctness-of-cancel-equal-plus, correctness-of-cancel-difference-plus, difference-elim, difference-leq-arg1, difference-add1-arg2, difference-sub1-arg2, difference-difference-arg1, difference-difference-arg2, difference-x-x, correctness-of-cancel-lessp-plus.

```
; ---------- TIMES ----------
```

THEOREM: equal-times-0
$((x * y) = 0) = ((x \simeq 0) \vee (y \simeq 0))$

THEOREM: equal-times-1
$((a * b) = 1) = ((a = 1) \wedge (b = 1))$

```
;(lemma equal-sub1-times-0 (rewrite)
;       (equal (equal (sub1 (times a b)) 0)
;       (or (zerop a)
;   (zerop b)
;   (and (equal a 1) (equal b 1)))))
```

THEOREM: equal-sub1-0
$((x - 1) = 0) = ((x \simeq 0) \vee (x = 1))$

THEOREM: times-zero
$(y \simeq 0) \rightarrow ((x * y) = 0)$

THEOREM: times-add1
$(x * (1 + y))$
$=$ **if** $y \in \mathbf{N}$ **then** $x + (x * y)$
else $\mathrm{fix}\,(x)$ **endif**

THEOREM: commutativity-of-times
$(y * x) = (x * y)$

THEOREM: times-distributes-over-plus-proof
$(x * (y + z)) = ((x * y) + (x * z))$

THEOREM: times-distributes-over-plus
$((x * (y + z)) = ((x * y) + (x * z)))$
$\wedge$ $(((x + y) * z) = ((x * z) + (y * z)))$

THEOREM: commutativity2-of-times
$(x * y * z) = (y * x * z)$

THEOREM: associativity-of-times
$((x * y) * z) = (x * y * z)$

THEOREM: times-distributes-over-difference-proof
$((a - b) * c) = ((a * c) - (b * c))$

THEOREM: times-distributes-over-difference
$(((a - b) * c) = ((a * c) - (b * c)))$
$\wedge \quad ((a * (b - c)) = ((a * b) - (a * c)))$

THEOREM: times-quotient-proof
$((x \not\simeq \mathtt{0}) \wedge ((y \ \mathbf{mod}\ x) = \mathtt{0})) \rightarrow (((y \div x) * x) = \mathrm{fix}(y))$

THEOREM: times-quotient
$((y \not\simeq \mathtt{0}) \wedge ((x \ \mathbf{mod}\ y) = \mathtt{0}))$
$\rightarrow \quad ((((x \div y) * y) = \mathrm{fix}(x)) \wedge ((y * (x \div y)) = \mathrm{fix}(x)))$

THEOREM: times-1-arg1
$(\mathtt{1} * x) = \mathrm{fix}(x)$

THEOREM: lessp-times1-proof
$((a < b) \wedge (c \not\simeq \mathtt{0})) \rightarrow ((a < (b * c)) = \mathbf{t})$

THEOREM: lessp-times1
$((a < b) \wedge (c \not\simeq \mathtt{0}))$
$\rightarrow \quad (((a < (b * c)) = \mathbf{t}) \wedge ((a < (c * b)) = \mathbf{t}))$

THEOREM: lessp-times2-proof
$((a \leq b) \wedge (c \not\simeq \mathtt{0})) \rightarrow (((b * c) < a) = \mathbf{f})$

THEOREM: lessp-times2
$((a \leq b) \wedge (c \not\simeq \mathtt{0}))$
$\rightarrow \quad ((((b * c) < a) = \mathbf{f}) \wedge (((c * b) < a) = \mathbf{f}))$

THEOREM: lessp-times3-proof1
$((a \not\simeq \mathtt{0}) \wedge (\mathtt{1} < b)) \rightarrow (a < (a * b))$

THEOREM: lessp-times3-proof2
$(a < (a * b)) \rightarrow ((a \not\simeq \mathtt{0}) \wedge (\mathtt{1} < b))$

THEOREM: lessp-times3
$((a < (a * b)) = ((a \not\simeq \mathtt{0}) \wedge (\mathtt{1} < b)))$
$\wedge \quad ((a < (b * a)) = ((a \not\simeq \mathtt{0}) \wedge (\mathtt{1} < b)))$

THEOREM: lessp-times-cancellation-proof
$((x * z) < (y * z)) = ((z \not\simeq \mathtt{0}) \wedge (x < y))$

THEOREM: lessp-times-cancellation1
$(((x * z) < (y * z)) = ((z \not\simeq \mathtt{0}) \wedge (x < y)))$
$\wedge \quad (((z * x) < (y * z)) = ((z \not\simeq \mathtt{0}) \wedge (x < y)))$
$\wedge \quad (((x * z) < (z * y)) = ((z \not\simeq \mathtt{0}) \wedge (x < y)))$
$\wedge \quad (((z * x) < (z * y)) = ((z \not\simeq \mathtt{0}) \wedge (x < y)))$

EVENT: Disable lessp-times-cancellation1.

THEOREM: lessp-plus-times-proof
$(x < a) \rightarrow (((x + (a * b)) < (a * c)) = (b < c))$

THEOREM: lessp-plus-times1
$(((a + (b * c)) < b) = ((a < b) \wedge (c \simeq \mathtt{0})))$
$\wedge \quad (((a + (c * b)) < b) = ((a < b) \wedge (c \simeq \mathtt{0})))$
$\wedge \quad ((((c * b) + a) < b) = ((a < b) \wedge (c \simeq \mathtt{0})))$
$\wedge \quad ((((b * c) + a) < b) = ((a < b) \wedge (c \simeq \mathtt{0})))$

THEOREM: lessp-plus-times2
$((a \not\simeq \mathtt{0}) \wedge (x < a))$
$\rightarrow \quad ((((x + (a * b)) < (a * c)) = (b < c))$
$\quad \wedge \quad (((x + (b * a)) < (a * c)) = (b < c))$
$\quad \wedge \quad (((x + (a * b)) < (c * a)) = (b < c))$
$\quad \wedge \quad (((x + (b * a)) < (c * a)) = (b < c))$
$\quad \wedge \quad ((((a * b) + x) < (a * c)) = (b < c))$
$\quad \wedge \quad ((((b * a) + x) < (a * c)) = (b < c))$
$\quad \wedge \quad ((((a * b) + x) < (c * a)) = (b < c))$
$\quad \wedge \quad ((((b * a) + x) < (c * a)) = (b < c)))$

THEOREM: lessp-1-times
$(\mathtt{1} < (a * b))$
$= \quad (\neg\ ((a \simeq \mathtt{0}) \vee (b \simeq \mathtt{0}) \vee ((a = \mathtt{1}) \wedge (b = \mathtt{1}))))$

```
;;; meta lemmas to cancel lessp-times and equal-times expressions
;; examples
;; (lessp (times b (times d a)) (times b (times e (times a f)))) ->
;;                                                 (and (and (not (zerop a))
;;                                                           (not (zerop b)))
;;                                                      (lessp (fix d) (times e f)))
;;
;; (equal (times b (times c d)) (times b d))    ->
;;                                               (or (or (zerop b) (zerop d))
;;                                                   (equal (fix c) 1))
```

DEFINITION:
times-tree (x)
$=$ **if** $x \simeq$ **nil** **then** `''1`
elseif cdr $(x) \simeq$ **nil** **then** list (`'fix`, car (x))
elseif cddr $(x) \simeq$ **nil** **then** list (`'times`, car (x), cadr (x))
else list (`'times`, car (x), times-tree (cdr (x))) **endif**

DEFINITION:
times-fringe (x)
= **if** listp $(x) \wedge$ (car (x) = 'times)
then append (times-fringe (cadr (x)), times-fringe (caddr (x)))
else cons $(x$, **nil**$)$ **endif**

DEFINITION:
or-zerop-tree (x)
= **if** $x \simeq$ **nil then** '(false)
elseif cdr $(x) \simeq$ **nil then** list ('zerop, car (x))
elseif cddr $(x) \simeq$ **nil**
then list ('or, list ('zerop, car (x)), list ('zerop, cadr (x)))
else list ('or, list ('zerop, car (x)), or-zerop-tree (cdr (x))) **endif**

DEFINITION:
and-not-zerop-tree (x)
= **if** $x \simeq$ **nil then** '(true)
elseif cdr $(x) \simeq$ **nil then** list ('not, list ('zerop, car (x)))
else list ('and,
list ('not, list ('zerop, car (x))),
and-not-zerop-tree (cdr (x))) **endif**

THEOREM: numberp-eval$-times
(car (x) = 'times) $\rightarrow$ (eval$ (**t**, x, a) $\in$ **N**)

EVENT: Disable numberp-eval$-times.

THEOREM: eval$-times
(car (x) = 'times)
$\rightarrow$ (eval$ (**t**, x, a) = (eval$ (**t**, cadr (x), a) $*$ eval$ (**t**, caddr (x), a)))

EVENT: Disable eval$-times.

THEOREM: eval$-or
(car (x) = 'or)
$\rightarrow$ (eval$ (**t**, x, a) = (eval$ (**t**, cadr (x), a) $\vee$ eval$ (**t**, caddr (x), a)))

EVENT: Disable eval$-or.

THEOREM: eval$-equal
(car (x) = 'equal)
$\rightarrow$ (eval$ (**t**, x, a) = (eval$ (**t**, cadr (x), a) = eval$ (**t**, caddr (x), a)))

EVENT: Disable eval$-equal.

THEOREM: eval$-lessp
$(\text{car}\,(x) = \texttt{'lessp})$
$\rightarrow \quad (\text{eval\$}\,(\mathbf{t},\ x,\ a) = (\text{eval\$}\,(\mathbf{t},\ \text{cadr}\,(x),\ a) < \text{eval\$}\,(\mathbf{t},\ \text{caddr}\,(x),\ a)))$

EVENT: Disable eval$-lessp.

THEOREM: eval$-quotient
$(\text{car}\,(x) = \texttt{'quotient})$
$\rightarrow \quad (\text{eval\$}\,(\mathbf{t},\ x,\ a) = (\text{eval\$}\,(\mathbf{t},\ \text{cadr}\,(x),\ a) \div \text{eval\$}\,(\mathbf{t},\ \text{caddr}\,(x),\ a)))$

EVENT: Disable eval$-quotient.

THEOREM: eval$-if
$(\text{car}\,(x) = \texttt{'if})$
$\rightarrow \quad (\text{eval\$}\,(\mathbf{t},\ x,\ a)$
$\quad = \quad \textbf{if}\ \text{eval\$}\,(\mathbf{t},\ \text{cadr}\,(x),\ a)\ \ \textbf{then}\ \text{eval\$}\,(\mathbf{t},\ \text{caddr}\,(x),\ a)$
$\quad\quad\ \ \textbf{else}\ \text{eval\$}\,(\mathbf{t},\ \text{cadddr}\,(x),\ a)\ \textbf{endif})$

EVENT: Disable eval$-if.

THEOREM: numberp-eval$-times-tree
$\text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(x),\ a) \in \mathbf{N}$

EVENT: Disable numberp-eval$-times-tree.

THEOREM: lessp-times-arg1
$(a \not\simeq \texttt{0}) \rightarrow (((a * x) \not< (a * y)) = (x \not< y))$

THEOREM: infer-equality-from-not-lessp
$((a \in \mathbf{N}) \wedge (b \in \mathbf{N})) \rightarrow (((a \not< b) \wedge (b \not< a)) = (a = b))$

THEOREM: equal-times-arg1
$(a \not\simeq \texttt{0}) \rightarrow (((a * x) = (a * y)) = (\text{fix}\,(x) = \text{fix}\,(y)))$

EVENT: Disable equal-times-arg1.

THEOREM: equal-times-bridge
$((a * b) = (c * (a * d))) = ((a \simeq \texttt{0}) \vee (\text{fix}\,(b) = (c * d)))$

EVENT: Disable equal-times-bridge.

THEOREM: eval$-times-member
$(e \in x)$
$\rightarrow \quad (\text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(x),\ a)$
$\quad = \quad (\text{eval\$}\,(\mathbf{t},\ e,\ a) * \text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(\text{delete}\,(e,\ x)),\ a)))$

EVENT: Disable eval$-times-member.

THEOREM: zerop-makes-times-tree-zero
((¬ eval$ (**t**, and-not-zerop-tree (*x*), *a*)) ∧ subbagp (*x*, *y*))
→ (eval$ (**t**, times-tree (*y*), *a*) = `0`)

EVENT: Disable zerop-makes-times-tree-zero.

THEOREM: or-zerop-tree-is-not-zerop-tree
eval$ (**t**, or-zerop-tree (*x*), *a*) = (¬ eval$ (**t**, and-not-zerop-tree (*x*), *a*))

EVENT: Disable or-zerop-tree-is-not-zerop-tree.

THEOREM: zerop-makes-times-tree-zero2
(eval$ (**t**, or-zerop-tree (*x*), *a*) ∧ subbagp (*x*, *y*))
→ (eval$ (**t**, times-tree (*y*), *a*) = `0`)

EVENT: Disable zerop-makes-times-tree-zero2.

THEOREM: times-tree-append
eval$ (**t**, times-tree (append (*x*, *y*)), *a*)
= (eval$ (**t**, times-tree (*x*), *a*) ∗ eval$ (**t**, times-tree (*y*), *a*))

EVENT: Disable times-tree-append.

THEOREM: times-tree-of-times-fringe
eval$ (**t**, times-tree (times-fringe (*x*)), *a*) = fix (eval$ (**t**, *x*, *a*))

EVENT: Disable times-tree-of-times-fringe.

DEFINITION:
cancel-lessp-times (*x*)
= **if** (car (*x*) = `'lessp`)
∧ (caadr (*x*) = `'times`)
∧ (caaddr (*x*) = `'times`)
then let *inboth* **be** bagint (times-fringe (cadr (*x*)),
times-fringe (caddr (*x*)))
in
if listp (*inboth*)
then list (`'and`,
and-not-zerop-tree (*inboth*),
list (`'lessp`,

```
                    times-tree (bagdiff (times-fringe (cadr (x)),
                                         inboth)),
                    times-tree (bagdiff (times-fringe (caddr (x)),
                                         inboth))))
        else x endif endlet
    else x endif
```

THEOREM: eval$-lessp-times-tree-bagdiff
$(\text{subbagp}(x, y) \wedge \text{subbagp}(x, z) \wedge \text{eval\$}(\mathbf{t}, \text{and-not-zerop-tree}(x), a))$
$\rightarrow \quad ((\text{eval\$}(\mathbf{t}, \text{times-tree}(\text{bagdiff}(y, x)), a)$
$< \quad \text{eval\$}(\mathbf{t}, \text{times-tree}(\text{bagdiff}(z, x)), a))$
$= \quad (\text{eval\$}(\mathbf{t}, \text{times-tree}(y), a) < \text{eval\$}(\mathbf{t}, \text{times-tree}(z), a)))$

EVENT: Disable eval$-lessp-times-tree-bagdiff.

THEOREM: zerop-makes-lessp-false-bridge
$((\text{car}(x) = \texttt{'times})$
$\wedge \quad (\text{car}(y) = \texttt{'times})$
$\wedge \quad (\neg\, \text{eval\$}(\mathbf{t},$
and-not-zerop-tree (bagint (times-fringe (x), times-fringe (y))),
$a)))$
$\rightarrow \quad (((\text{eval\$}(\mathbf{t}, \text{cadr}(x), a) * \text{eval\$}(\mathbf{t}, \text{caddr}(x), a))$
$< \quad (\text{eval\$}(\mathbf{t}, \text{cadr}(y), a) * \text{eval\$}(\mathbf{t}, \text{caddr}(y), a)))$
$= \quad \mathbf{f})$

EVENT: Disable zerop-makes-lessp-false-bridge.

THEOREM: correctness-of-cancel-lessp-times
$\text{eval\$}(\mathbf{t}, x, a) = \text{eval\$}(\mathbf{t}, \text{cancel-lessp-times}(x), a)$

DEFINITION:

```
cancel-equal-times (x)
=   if (car (x) = 'equal)
      ∧  (caadr (x) = 'times)
      ∧  (caaddr (x) = 'times)
    then let inboth be bagint (times-fringe (cadr (x)),
                               times-fringe (caddr (x)))
         in
         if listp (inboth)
         then list ('or,
                    or-zerop-tree (inboth),
                    list ('equal,
                          times-tree (bagdiff (times-fringe (cadr (x)),
                                               inboth)),
```

```
                                  times-tree (bagdiff (times-fringe (caddr (x)),
                                                       inboth))))
          else x endif endlet
    else x endif
```

THEOREM: zerop-makes-equal-true-bridge

$$
\begin{array}{ll}
((\text{car}\,(x) = \texttt{'times}) & \\
\wedge & (\text{car}\,(y) = \texttt{'times}) \\
\wedge & \text{eval\$}\,(\mathbf{t},\ \text{or-zerop-tree}\,(\text{bagint}\,(\text{times-fringe}\,(x),\ \text{times-fringe}\,(y))),\ a)) \\
\rightarrow & (((\text{eval\$}\,(\mathbf{t},\ \text{cadr}\,(x),\ a) * \text{eval\$}\,(\mathbf{t},\ \text{caddr}\,(x),\ a)) \\
& \quad = \quad (\text{eval\$}\,(\mathbf{t},\ \text{cadr}\,(y),\ a) * \text{eval\$}\,(\mathbf{t},\ \text{caddr}\,(y),\ a))) \\
& \quad = \quad \mathbf{t})
\end{array}
$$

EVENT: Disable zerop-makes-equal-true-bridge.

THEOREM: eval$-equal-times-tree-bagdiff

$$
\begin{array}{ll}
(\text{subbagp}\,(x,\ y) \wedge \text{subbagp}\,(x,\ z) \wedge (\neg\ \text{eval\$}\,(\mathbf{t},\ \text{or-zerop-tree}\,(x),\ a))) & \\
\rightarrow & ((\text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(\text{bagdiff}\,(y,\ x)),\ a) \\
& \quad = \quad \text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(\text{bagdiff}\,(z,\ x)),\ a)) \\
& \quad = \quad (\text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(y),\ a) = \text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(z),\ a)))
\end{array}
$$

EVENT: Disable eval$-equal-times-tree-bagdiff.

THEOREM: cancel-equal-times-preserves-inequality

$$
\begin{array}{ll}
(\text{subbagp}\,(z,\ x) & \\
\wedge & \text{subbagp}\,(z,\ y) \\
\wedge & (\text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(x),\ a) \neq \text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(y),\ a))) \\
\rightarrow & (\text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(\text{bagdiff}\,(x,\ z)),\ a) \\
& \quad \neq \quad \text{eval\$}\,(\mathbf{t},\ \text{times-tree}\,(\text{bagdiff}\,(y,\ z)),\ a))
\end{array}
$$

EVENT: Disable cancel-equal-times-preserves-inequality.

THEOREM: cancel-equal-times-preserves-inequality-bridge

```
((car (x) = 'times)
 ∧   (car (y) = 'times)
 ∧   ((eval$ (t, cadr (x), a) * eval$ (t, caddr (x), a))
      ≠   (eval$ (t, cadr (y), a) * eval$ (t, caddr (y), a))))
→   (eval$ (t,
              times-tree (bagdiff (times-fringe (x),
                                   bagint (times-fringe (x), times-fringe (y)))),
              a)
     ≠   eval$ (t,
                  times-tree (bagdiff (times-fringe (y),
```

bagint (times-fringe (x),
times-fringe (y))),
a))

EVENT: Disable cancel-equal-times-preserves-inequality-bridge.

THEOREM: correctness-of-cancel-equal-times
eval$ (**t**, x, a) = eval$ (**t**, cancel-equal-times (x), a)

```
; Define the available  theory of  multiplication.   To  get the list of
; events to put in the theory,  evaluate the following  form in NQTHM at
; this point  in  the  script. This   form  lists all lemmas   which are
; globally enabled, and which have non-null lemma type.
;
; (remove-if-not (function (lambda (x)
;     (and (member x (lemmas))
;  (not (assoc x disabled-lemmas))
;  (not (null (nth 2 (get x 'event))))
;  (not (member x (nth 2 (get 'addition 'event)))))))
;         chronology)
```

EVENT: Let us define the theory *multiplication* to consist of the following events: equal-times-0, equal-times-1, equal-sub1-0, times-zero, times-add1, commutativity-of-times, times-distributes-over-plus, commutativity2-of-times, associativity-of-times, times-distributes-over-difference, times-quotient, times-1-arg1, lessp-times1, lessp-times2, lessp-times3, lessp-plus-times1, lessp-plus-times2, lessp-1-times, correctness-of-cancel-lessp-times, correctness-of-cancel-equal-times.

```
; ---------- REMAINDER ----------
```

THEOREM: lessp-remainder
$((x \textbf{ mod } y) < y) = (y \not\simeq 0)$

THEOREM: remainder-noop
$(a < b) \rightarrow ((a \textbf{ mod } b) = \text{fix}\,(a))$

THEOREM: remainder-of-non-number
$(a \notin \textbf{N}) \rightarrow ((a \textbf{ mod } n) = (0 \textbf{ mod } n))$

THEOREM: remainder-zero
$(x \simeq 0) \rightarrow ((y \textbf{ mod } x) = \text{fix}\,(y))$

THEOREM: plus-remainder-times-quotient
$((x \bmod y) + (y * (x \div y))) = \mathrm{fix}(x)$

EVENT: Disable plus-remainder-times-quotient.

THEOREM: remainder-quotient-elim
$((y \not\simeq 0) \wedge (x \in \mathbf{N})) \rightarrow (((x \bmod y) + (y * (x \div y))) = x)$

```
; (lemma remainder-sub1 (rewrite)
;        (implies (and (not (zerop a))
;       (not (zerop b)))
;  (equal (remainder (sub1 a) b)
;         (if (equal (remainder a b) 0)
;     (sub1 b)
;     (sub1 (remainder a b)))))
;        ((enable lessp-remainder
;  remainder-noop
;  remainder-quotient-elim)
;  (enable-theory addition)
;  (induct (remainder a b))))
```

THEOREM: remainder-add1
$((a \bmod b) = 0) \rightarrow (((1 + a) \bmod b) = (1 \bmod b))$

THEOREM: remainder-plus-proof
$((b \bmod c) = 0) \rightarrow (((a + b) \bmod c) = (a \bmod c))$

THEOREM: remainder-plus
$((a \bmod c) = 0)$
$\rightarrow \quad ((((a + b) \bmod c) = (b \bmod c))$
$\quad \wedge \quad (((b + a) \bmod c) = (b \bmod c))$
$\quad \wedge \quad (((x + y + a) \bmod c) = ((x + y) \bmod c)))$

THEOREM: equal-remainder-plus-0-proof
$((a \bmod c) = 0) \rightarrow ((((a + b) \bmod c) = 0) = ((b \bmod c) = 0))$

THEOREM: equal-remainder-plus-0
$((a \bmod c) = 0)$
$\rightarrow \quad (((((a + b) \bmod c) = 0) = ((b \bmod c) = 0))$
$\quad \wedge \quad ((((b + a) \bmod c) = 0) = ((b \bmod c) = 0))$
$\quad \wedge \quad ((((x + y + a) \bmod c) = 0)$
$\quad\quad = \quad (((x + y) \bmod c) = 0)))$

THEOREM: equal-remainder-plus-remainder-proof
$(a < c) \rightarrow ((((a + b) \bmod c) = (b \bmod c)) = (a \simeq 0))$

THEOREM: equal-remainder-plus-remainder
$(a < c)$
$\rightarrow \quad (((((a + b)\ \mathbf{mod}\ c) = (b\ \mathbf{mod}\ c)) = (a \simeq \mathtt{0}))$
$\quad \wedge \quad ((((b + a)\ \mathbf{mod}\ c) = (b\ \mathbf{mod}\ c)) = (a \simeq \mathtt{0})))$

EVENT: Disable equal-remainder-plus-remainder.

THEOREM: remainder-times1-proof
$((b\ \mathbf{mod}\ c) = \mathtt{0}) \rightarrow (((a * b)\ \mathbf{mod}\ c) = \mathtt{0})$

THEOREM: remainder-times1
$((b\ \mathbf{mod}\ c) = \mathtt{0})$
$\rightarrow \quad ((((a * b)\ \mathbf{mod}\ c) = \mathtt{0}) \wedge (((b * a)\ \mathbf{mod}\ c) = \mathtt{0}))$

THEOREM: remainder-times1-instance-proof
$((x * y)\ \mathbf{mod}\ y) = \mathtt{0}$

THEOREM: remainder-times1-instance
$(((x * y)\ \mathbf{mod}\ y) = \mathtt{0}) \wedge (((x * y)\ \mathbf{mod}\ x) = \mathtt{0})$

THEOREM: remainder-times-times-proof
$((x * y)\ \mathbf{mod}\ (x * z)) = (x * (y\ \mathbf{mod}\ z))$

THEOREM: remainder-times-times
$(((x * y)\ \mathbf{mod}\ (x * z)) = (x * (y\ \mathbf{mod}\ z)))$
$\wedge \quad (((x * z)\ \mathbf{mod}\ (y * z)) = ((x\ \mathbf{mod}\ y) * z))$

EVENT: Disable remainder-times-times.

THEOREM: remainder-times2-proof
$((a\ \mathbf{mod}\ z) = \mathtt{0}) \rightarrow ((a\ \mathbf{mod}\ (z * y)) = (z * ((a \div z)\ \mathbf{mod}\ y)))$

THEOREM: remainder-times2
$((a\ \mathbf{mod}\ z) = \mathtt{0})$
$\rightarrow \quad (((a\ \mathbf{mod}\ (y * z)) = (z * ((a \div z)\ \mathbf{mod}\ y)))$
$\quad \wedge \quad ((a\ \mathbf{mod}\ (z * y)) = (z * ((a \div z)\ \mathbf{mod}\ y))))$

THEOREM: remainder-times2-instance
$(((x * y)\ \mathbf{mod}\ (x * z)) = (x * (y\ \mathbf{mod}\ z)))$
$\wedge \quad (((x * z)\ \mathbf{mod}\ (y * z)) = ((x\ \mathbf{mod}\ y) * z))$

THEOREM: remainder-difference1
$((a\ \mathbf{mod}\ c) = (b\ \mathbf{mod}\ c))$
$\rightarrow \quad (((a - b)\ \mathbf{mod}\ c) = ((a\ \mathbf{mod}\ c) - (b\ \mathbf{mod}\ c)))$

DEFINITION:
double-remainder-induction $(a,\ b,\ c)$
$=$ **if** $c \simeq \mathtt{0}$ **then** $\mathtt{0}$
elseif $a < c$ **then** $\mathtt{0}$
elseif $b < c$ **then** $\mathtt{0}$
else double-remainder-induction $(a - c,\ b - c,\ c)$ **endif**

THEOREM: remainder-difference2
$(((a \textbf{ mod } c) = \mathtt{0}) \wedge ((b \textbf{ mod } c) \neq \mathtt{0}))$
$\rightarrow$ $(((a - b) \textbf{ mod } c)$
$=$ **if** $b < a$ **then** $c - (b \textbf{ mod } c)$
else $\mathtt{0}$ **endif**)

THEOREM: remainder-difference3
$(((b \textbf{ mod } c) = \mathtt{0}) \wedge ((a \textbf{ mod } c) \neq \mathtt{0}))$
$\rightarrow$ $(((a - b) \textbf{ mod } c)$
$=$ **if** $b < a$ **then** $a \textbf{ mod } c$
else $\mathtt{0}$ **endif**)

EVENT: Disable remainder-difference3.

THEOREM: equal-remainder-difference-0
$(((a - b) \textbf{ mod } c) = \mathtt{0})$
$=$ **if** $b \leq a$ **then** $(a \textbf{ mod } c) = (b \textbf{ mod } c)$
else t endif

EVENT: Disable equal-remainder-difference-0.

THEOREM: lessp-plus-fact
$(((b \textbf{ mod } x) = \mathtt{0}) \wedge ((c \textbf{ mod } x) = \mathtt{0}) \wedge (b < c) \wedge (a < x))$
$\rightarrow$ $(((a + b) < c) = \mathbf{t})$

EVENT: Disable lessp-plus-fact.

THEOREM: remainder-plus-fact
$(((b \textbf{ mod } x) = \mathtt{0}) \wedge ((c \textbf{ mod } x) = \mathtt{0}) \wedge (a < x))$
$\rightarrow$ $(((a + b) \textbf{ mod } c) = (a + (b \textbf{ mod } c)))$

THEOREM: remainder-plus-times-times-proof
$(a < b)$
$\rightarrow$ $(((a + (b * c)) \textbf{ mod } (b * d))$
$=$ $(a + ((b * c) \textbf{ mod } (b * d))))$

THEOREM: remainder-plus-times-times
$(a < b)$
$\rightarrow \quad ((((a + (b * c)) \textbf{ mod } (b * d))$
$\quad = \quad (a + ((b * c) \textbf{ mod } (b * d))))$
$\quad \wedge \quad (((a + (c * b)) \textbf{ mod } (d * b))$
$\quad = \quad (a + ((c * b) \textbf{ mod } (d * b)))))$

```
; REMAINDER-PLUS-TIMES-TIMES-INSTANCE is the completion of the rules
; TIMES-DISTRIBUTES-OVER-PLUS, REMAINDER-TIMES-TIMES and REMAINDER-PLUS-TIMES-TIMES
```

THEOREM: remainder-plus-times-times-instance
$(a < b)$
$\rightarrow \quad ((((a + (b * c) + (b * d)) \textbf{ mod } (b * e))$
$\quad = \quad (a + (b * ((c + d) \textbf{ mod } e))))$
$\quad \wedge \quad (((a + (c * b) + (d * b)) \textbf{ mod } (e * b))$
$\quad = \quad (a + (b * ((c + d) \textbf{ mod } e)))))$

THEOREM: remainder-remainder
$((b \textbf{ mod } a) = 0) \rightarrow (((n \textbf{ mod } b) \textbf{ mod } a) = (n \textbf{ mod } a))$

THEOREM: remainder-1-arg1
$(1 \textbf{ mod } x)$
$= \quad \textbf{if } x = 1 \textbf{ then } 0$
$\quad \textbf{else } 1 \textbf{ endif}$

THEOREM: remainder-1-arg2
$(y \textbf{ mod } 1) = 0$

THEOREM: remainder-x-x
$(x \textbf{ mod } x) = 0$

THEOREM: transitivity-of-divides
$(((a \textbf{ mod } b) = 0) \wedge ((b \textbf{ mod } c) = 0)) \rightarrow ((a \textbf{ mod } c) = 0)$

```
; Define the available  theory of  remainder.   To  get the list of
; events to put in the theory,  evaluate the following  form in NQTHM at
; this point  in  the  script. This   form  lists all lemmas   which are
; globally enabled, and which have non-null lemma type.
;
;
; (let ((lemmas (lemmas)))
;   (remove-if-not (function (lambda (x)
;       (and (member x lemmas)
;    (not (assoc x disabled-lemmas))
```

```
;    (not (null (nth 2 (get x 'event))))
;    (not (member x (nth 2 (get 'addition 'event))))
;    (not (member x (nth 2 (get 'multiplication 'event)))))))
;   chronology))
```

EVENT: Let us define the theory *remainders* to consist of the following events: lessp-remainder, remainder-noop, remainder-of-non-number, remainder-zero, remainder-quotient-elim, remainder-add1, remainder-plus, equal-remainder-plus-0, remainder-times1, remainder-times1-instance, remainder-times2, remainder-times2-instance, remainder-difference1, remainder-difference2, remainder-plus-times-times, remainder-plus-times-times-instance, remainder-remainder, remainder-1-arg1, remainder-1-arg2, remainder-x-x.

```
; ---------- QUOTIENT, DIVIDES  ----------
```

THEOREM: quotient-noop
$(b = \mathtt{1}) \rightarrow ((a \div b) = \mathrm{fix}(a))$

THEOREM: quotient-of-non-number
$(a \notin \mathbf{N}) \rightarrow ((a \div n) = (\mathtt{0} \div n))$

THEOREM: quotient-zero
$(x \simeq \mathtt{0}) \rightarrow ((y \div x) = \mathtt{0})$

THEOREM: quotient-add1
$((a \textbf{ mod } b) = \mathtt{0})$
$\rightarrow \quad (((1 + a) \div b)$
$\quad = \quad \textbf{if } b = \mathtt{1} \textbf{ then } 1 + (a \div b)$
$\quad\quad \textbf{else } a \div b \textbf{ endif})$

THEOREM: equal-quotient-0
$((a \div b) = \mathtt{0}) = ((b \simeq \mathtt{0}) \vee (a < b))$

THEOREM: quotient-sub1
$((a \not\simeq \mathtt{0}) \wedge (b \not\simeq \mathtt{0}))$
$\rightarrow \quad (((a - 1) \div b)$
$\quad = \quad \textbf{if } (a \textbf{ mod } b) = \mathtt{0} \textbf{ then } (a \div b) - 1$
$\quad\quad \textbf{else } a \div b \textbf{ endif})$

THEOREM: quotient-plus-proof
$((b \textbf{ mod } c) = \mathtt{0}) \rightarrow (((a + b) \div c) = ((a \div c) + (b \div c)))$

THEOREM: quotient-plus
$((a \textbf{ mod } c) = 0)$
$\rightarrow \quad ((((a + b) \div c) = ((a \div c) + (b \div c)))$
$\wedge \quad (((b + a) \div c) = ((a \div c) + (b \div c)))$
$\wedge \quad (((x + y + a) \div c)$
$= \quad (((x + y) \div c) + (a \div c))))$

```
; I need QUOTIENT-TIMES-INSTANCE to prove the more general QUOTIENT-TIMES,
; but I want QUOTIENT-TIMES-INSTANCE to be tried first (i.e. come after
; QUOTIENT-TIMES in the event list.) So first, prove QUOTIENT-TIMES-INSTANCE-TEMP,
; then prove QUOTIENT-TIMES, and finally give QUOTIENT-TIMES-INSTANCE.
```

THEOREM: quotient-times-instance-temp-proof
$((y * x) \div y)$
$= \quad \textbf{if } y \simeq 0 \textbf{ then } 0$
$\textbf{else } \mathrm{fix}(x) \textbf{ endif}$

THEOREM: quotient-times-instance-temp
$(((y * x) \div y)$
$= \quad \textbf{if } y \simeq 0 \textbf{ then } 0$
$\textbf{else } \mathrm{fix}(x) \textbf{ endif})$
$\wedge \quad (((x * y) \div y)$
$= \quad \textbf{if } y \simeq 0 \textbf{ then } 0$
$\textbf{else } \mathrm{fix}(x) \textbf{ endif})$

EVENT: Disable quotient-times-instance-temp.

THEOREM: quotient-times-proof
$((a \textbf{ mod } c) = 0) \rightarrow (((a * b) \div c) = (b * (a \div c)))$

THEOREM: quotient-times
$((a \textbf{ mod } c) = 0)$
$\rightarrow \quad ((((a * b) \div c) = (b * (a \div c)))$
$\wedge \quad (((b * a) \div c) = (b * (a \div c))))$

THEOREM: quotient-times-instance
$(((y * x) \div y)$
$= \quad \textbf{if } y \simeq 0 \textbf{ then } 0$
$\textbf{else } \mathrm{fix}(x) \textbf{ endif})$
$\wedge \quad (((x * y) \div y)$
$= \quad \textbf{if } y \simeq 0 \textbf{ then } 0$
$\textbf{else } \mathrm{fix}(x) \textbf{ endif})$

THEOREM: quotient-times-times-proof
$((x * y) \div (x * z))$
$=$ **if** $x \simeq 0$ **then** 0
else $y \div z$ **endif**

THEOREM: quotient-times-times
$(((x * y) \div (x * z))$
$=$ **if** $x \simeq 0$ **then** 0
else $y \div z$ **endif**)
$\wedge$ $(((x * z) \div (y * z))$
$=$ **if** $z \simeq 0$ **then** 0
else $x \div y$ **endif**)

EVENT: Disable quotient-times-times.

THEOREM: quotient-difference1
$((a \textbf{ mod } c) = (b \textbf{ mod } c))$
$\rightarrow$ $(((a - b) \div c) = ((a \div c) - (b \div c)))$

THEOREM: quotient-lessp-arg1
$(a < b) \rightarrow ((a \div b) = 0)$

THEOREM: quotient-difference2
$(((a \textbf{ mod } c) = 0) \wedge ((b \textbf{ mod } c) \neq 0))$
$\rightarrow$ $(((a - b) \div c)$
$=$ **if** $b < a$ **then** $(a \div c) - (1 + (b \div c))$
else 0 **endif**)

THEOREM: quotient-difference3
$(((b \textbf{ mod } c) = 0) \wedge ((a \textbf{ mod } c) \neq 0))$
$\rightarrow$ $(((a - b) \div c)$
$=$ **if** $b < a$ **then** $(a \div c) - (b \div c)$
else 0 **endif**)

THEOREM: remainder-equals-its-first-argument
$(a = (a \textbf{ mod } b)) = ((a \in \mathbf{N}) \wedge ((b \simeq 0) \vee (a < b)))$

EVENT: Disable remainder-equals-its-first-argument.

THEOREM: quotient-remainder-times
$((x \textbf{ mod } (a * b)) \div a) = ((x \div a) \textbf{ mod } b)$

THEOREM: quotient-remainder
$((c \textbf{ mod } a) = 0) \rightarrow (((b \textbf{ mod } c) \div a) = ((b \div a) \textbf{ mod } (c \div a)))$

THEOREM: quotient-remainder-instance
$((x \textbf{ mod } (a * b)) \div a) = ((x \div a) \textbf{ mod } b)$

THEOREM: quotient-plus-fact
$(((b \textbf{ mod } x) = 0) \wedge ((c \textbf{ mod } x) = 0) \wedge (a < x))$
$\rightarrow \quad (((a + b) \div c) = (b \div c))$

THEOREM: quotient-plus-times-times-proof
$(a < b)$
$\rightarrow \quad (((a + (b * c)) \div (b * d)) = ((b * c) \div (b * d)))$

THEOREM: quotient-plus-times-times
$(a < b)$
$\rightarrow \quad ((((a + (b * c)) \div (b * d)) = ((b * c) \div (b * d)))$
$\quad \wedge \quad (((a + (b * c)) \div (b * d))$
$\quad\quad = \quad ((b * c) \div (b * d))))$

```
; QUOTIENT-PLUS-TIMES-TIMES-INSTANCE is the completion of the rules
; QUOTIENT-TIMES-TIMES, QUOTIENT-PLUS-TIMES-TIMES and TIMES-DISTRIBUTES-OVER-PLUS
```

THEOREM: quotient-plus-times-times-instance
$(a < b)$
$\rightarrow \quad ((((a + (b * c) + (b * d)) \div (b * e))$
$\quad = \quad$ **if** $b \simeq 0$ **then** 0
$\quad\quad$ **else** $(c + d) \div e$ **endif**)
$\quad \wedge \quad (((a + (c * b) + (d * b)) \div (e * b))$
$\quad\quad = \quad$ **if** $b \simeq 0$ **then** 0
$\quad\quad\quad$ **else** $(d + c) \div e$ **endif**))

THEOREM: quotient-quotient
$((b \div a) \div c) = (b \div (a * c))$

THEOREM: leq-quotient
$(a < b) \rightarrow ((a \div c) \leq (b \div c))$

THEOREM: quotient-1-arg2
$(n \div 1) = \text{fix}(n)$

THEOREM: quotient-1-arg1-casesplit
$(n \simeq 0) \vee (n = 1) \vee (1 < n)$

THEOREM: quotient-1-arg1
$(1 \div n)$
$=$ **if** $n = 1$ **then** 1
else 0 **endif**

THEOREM: quotient-x-x
$(x \not\simeq \texttt{0}) \rightarrow ((x \div x) = \texttt{1})$

THEOREM: lessp-quotient
$((i \div j) < i) = ((i \not\simeq \texttt{0}) \wedge (j \neq \texttt{1}))$

```
;; Metalemma to cancel quotient-times expressions

;; ex.
;; (quotient (times a b) (times c (times d a)))  ->
;;                                                (if (not (zerop a))
;;                                                    (quotient (fix b) (times c d))
;;                                                 (zero))
;;
```

DEFINITION:
cancel-quotient-times (x)
= **if** (car (x) = `'quotient`)
∧ (caadr (x) = `'times`)
∧ (caaddr (x) = `'times`)
then let *inboth* **be** bagint (times-fringe (cadr (x)),
times-fringe (caddr (x))))
in
if listp (*inboth*)
then list (`'if`,
and-not-zerop-tree (*inboth*),
list (`'quotient`,
times-tree (bagdiff (times-fringe (cadr (x)),
inboth)),
times-tree (bagdiff (times-fringe (caddr (x)),
inboth))),
`'(zero)`)
else x **endif endlet**
else x **endif**

THEOREM: zerop-makes-quotient-zero-bridge
((car (x) = `'times`)
∧ (car (y) = `'times`)
∧ (¬ eval$ (**t**,
and-not-zerop-tree (bagint (times-fringe (x), times-fringe (y))),
a)))
→ (((eval$ (**t**, cadr (x), a) ∗ eval$ (**t**, caddr (x), a))
÷ (eval$ (**t**, cadr (y), a) ∗ eval$ (**t**, caddr (y), a)))
= **0**)

EVENT: Disable zerop-makes-quotient-zero-bridge.

THEOREM: eval$-quotient-times-tree-bagdiff
(subbagp (*x*, *y*) ∧ subbagp (*x*, *z*) ∧ eval$ (**t**, and-not-zerop-tree (*x*), *a*))
→ ((eval$ (**t**, times-tree (bagdiff (*y*, *x*)), *a*)
÷ eval$ (**t**, times-tree (bagdiff (*z*, *x*)), *a*))
= (eval$ (**t**, times-tree (*y*), *a*) ÷ eval$ (**t**, times-tree (*z*), *a*)))

EVENT: Disable eval$-quotient-times-tree-bagdiff.

THEOREM: correctness-of-cancel-quotient-times
eval$ (**t**, *x*, *a*) = eval$ (**t**, cancel-quotient-times (*x*), *a*)

```
; Define the available theory of quotient.  To get the list of events to
; put in the theory, evaluate the following form in NQTHM at  this point
; in the script. This form lists all  lemmas which are globally enabled,
; and which have non-null lemma type.
;
;
; (let ((lemmas (lemmas)))
;   (remove-if-not (function (lambda (x)
;       (and (member x lemmas)
;    (not (assoc x disabled-lemmas))
;    (not (null (nth 2 (get x 'event))))
;    (not (member x (nth 2 (get 'addition 'event))))
;    (not (member x (nth 2 (get 'multiplication 'event))))
;    (not (member x (nth 2 (get 'remainders 'event)))))))
;   chronology))
```

EVENT: Let us define the theory *quotients* to consist of the following events: quotient-noop, quotient-of-non-number, quotient-zero, quotient-add1, equal-quotient-0, quotient-sub1, quotient-plus, quotient-times, quotient-times-instance, quotient-difference1, quotient-lessp-arg1, quotient-difference2, quotient-difference3, quotient-remainder-times, quotient-remainder, quotient-remainder-instance, quotient-plus-times-times, quotient-plus-times-times-instance, quotient-quotient, quotient-1-arg2, quotient-1-arg1, quotient-x-x, lessp-quotient, correctness-of-cancel-quotient-times.

```
;;; exp, log, and gcd
```

DEFINITION:
$\exp(i, j)$
$=$ **if** $j \simeq \mathtt{0}$ **then** $\mathtt{1}$
else $i * \exp(i, j - 1)$ **endif**

DEFINITION:
$\log(\mathit{base}, n)$
$=$ **if** $\mathit{base} < \mathtt{2}$ **then** $\mathtt{0}$
elseif $n \simeq \mathtt{0}$ **then** $\mathtt{0}$
else $1 + \log(\mathit{base}, n \div \mathit{base})$ **endif**

DEFINITION:
$\gcd(x, y)$
$=$ **if** $x \simeq \mathtt{0}$ **then** $\mathrm{fix}(y)$
elseif $y \simeq \mathtt{0}$ **then** x
elseif $x < y$ **then** $\gcd(x, y - x)$
else $\gcd(x - y, y)$ **endif**

THEOREM: remainder-exp
$(k \not\simeq \mathtt{0}) \rightarrow ((\exp(n, k) \textbf{ mod } n) = \mathtt{0})$

DEFINITION:
double-number-induction (i, j)
$=$ **if** $i \simeq \mathtt{0}$ **then** $\mathtt{0}$
elseif $j \simeq \mathtt{0}$ **then** $\mathtt{0}$
else double-number-induction $(i - 1, j - 1)$ **endif**

THEOREM: remainder-exp-exp
$(i \leq j) \rightarrow ((\exp(a, j) \textbf{ mod } \exp(a, i)) = \mathtt{0})$

THEOREM: quotient-exp
$(k \not\simeq \mathtt{0})$
$\rightarrow$ $((\exp(n, k) \div n)$
$=$ **if** $n \simeq \mathtt{0}$ **then** $\mathtt{0}$
else $\exp(n, k - 1)$ **endif**$)$

THEOREM: exp-zero
$(k \simeq \mathtt{0}) \rightarrow (\exp(n, k) = \mathtt{1})$

THEOREM: exp-add1
$\exp(n, 1 + k) = (n * \exp(n, k))$

THEOREM: exp-plus
$\exp(i, j + k) = (\exp(i, j) * \exp(i, k))$

THEOREM: exp-0-arg1
$\exp(\mathtt{0}, k)$
$=$ **if** $k \simeq \mathtt{0}$ **then** $\mathtt{1}$
else $\mathtt{0}$ **endif**

THEOREM: exp-1-arg1
$\exp(\mathtt{1}, k) = \mathtt{1}$

THEOREM: exp-0-arg2
$\exp(n, \mathtt{0}) = \mathtt{1}$

THEOREM: exp-times
$\exp(i * j, k) = (\exp(i, k) * \exp(j, k))$

THEOREM: exp-exp
$\exp(\exp(i, j), k) = \exp(i, j * k)$

THEOREM: equal-exp-0
$(\exp(n, k) = \mathtt{0}) = ((n \simeq \mathtt{0}) \wedge (k \not\simeq \mathtt{0}))$

THEOREM: equal-exp-1
$(\exp(n, k) = \mathtt{1})$
$=$ **if** $k \simeq \mathtt{0}$ **then t**
else $n = \mathtt{1}$ **endif**

THEOREM: exp-difference
$((c \leq b) \wedge (a \not\simeq \mathtt{0})) \rightarrow (\exp(a, b - c) = (\exp(a, b) \div \exp(a, c)))$

EVENT: Let us define the theory *exponentiation* to consist of the following events: equal-exp-0, equal-exp-1, exp-exp, exp-add1, exp-times, exp-1-arg1, exp-zero, exp-0-arg2, exp-0-arg1, exp-difference, exp-plus, quotient-exp, remainder-exp-exp, remainder-exp.

THEOREM: equal-log-0
$(\log(base, n) = \mathtt{0}) = ((base < \mathtt{2}) \vee (n \simeq \mathtt{0}))$

THEOREM: log-0
$(n \simeq \mathtt{0}) \rightarrow (\log(base, n) = \mathtt{0})$

THEOREM: log-1
$(\mathtt{1} < base) \rightarrow (\log(base, \mathtt{1}) = \mathtt{1})$

DEFINITION:
double-log-induction$(base, a, b)$
$=$ **if** $base < \mathtt{2}$ **then** $\mathtt{0}$
elseif $a \simeq \mathtt{0}$ **then** $\mathtt{0}$
elseif $b \simeq \mathtt{0}$ **then** $\mathtt{0}$
else double-log-induction$(base, a \div base, b \div base)$ **endif**

THEOREM: leq-log-log
$(n \leq m) \rightarrow (\log(c,\ n) \leq \log(c,\ m))$

THEOREM: log-quotient
$(1 < c) \rightarrow (\log(c,\ n \div c) = (\log(c,\ n) - 1))$

THEOREM: log-quotient-times-proof
$(1 < c) \rightarrow (\log(c,\ n \div (c * m)) = (\log(c,\ n \div m) - 1))$

THEOREM: log-quotient-times
$(1 < c)$
$\rightarrow \quad ((\log(c,\ n \div (c * m)) = (\log(c,\ n \div m) - 1))$
$\quad \wedge \quad (\log(c,\ n \div (m * c)) = (\log(c,\ n \div m) - 1)))$

THEOREM: log-quotient-exp
$(1 < c) \rightarrow (\log(c,\ n \div \exp(c,\ m)) = (\log(c,\ n) - m))$

THEOREM: log-times-proof
$((1 < c) \wedge (n \not\simeq 0)) \rightarrow (\log(c,\ c * n) = (1 + \log(c,\ n)))$

THEOREM: log-times
$((1 < c) \wedge (n \not\simeq 0))$
$\rightarrow \quad ((\log(c,\ c * n) = (1 + \log(c,\ n)))$
$\quad \wedge \quad (\log(c,\ n * c) = (1 + \log(c,\ n))))$

THEOREM: log-times-exp-proof
$((1 < c) \wedge (n \not\simeq 0)) \rightarrow (\log(c,\ n * \exp(c,\ m)) = (\log(c,\ n) + m))$

THEOREM: log-times-exp
$((1 < c) \wedge (n \not\simeq 0))$
$\rightarrow \quad ((\log(c,\ n * \exp(c,\ m)) = (\log(c,\ n) + m))$
$\quad \wedge \quad (\log(c,\ \exp(c,\ m) * n) = (\log(c,\ n) + m)))$

THEOREM: log-exp
$(1 < c) \rightarrow (\log(c,\ \exp(c,\ n)) = (1 + n))$

EVENT: Let us define the theory *logs* to consist of the following events: log-exp, log-times-exp, log-times, log-quotient-exp, log-quotient-times, log-quotient, log-1, log-0, equal-log-0, exp-exp.

THEOREM: commutativity-of-gcd
$\gcd(b,\ a) = \gcd(a,\ b)$

DEFINITION:
single-number-induction(n)
$=$ **if** $n \simeq 0$ **then** 0
else single-number-induction$(n - 1)$ **endif**

THEOREM: gcd-0
$(\gcd(\mathtt{0},\ x) = \mathrm{fix}(x)) \wedge (\gcd(x,\ \mathtt{0}) = \mathrm{fix}(x))$

THEOREM: gcd-1
$(\gcd(\mathtt{1},\ x) = \mathtt{1}) \wedge (\gcd(x,\ \mathtt{1}) = \mathtt{1})$

THEOREM: equal-gcd-0
$(\gcd(a,\ b) = \mathtt{0}) = ((a \simeq \mathtt{0}) \wedge (b \simeq \mathtt{0}))$

THEOREM: lessp-gcd
$(b \not\simeq \mathtt{0}) \rightarrow (((b < \gcd(a,\ b)) = \mathbf{f}) \wedge ((b < \gcd(b,\ a)) = \mathbf{f}))$

THEOREM: gcd-plus-instance-temp-proof
$\gcd(a,\ a + b) = \gcd(a,\ b)$

THEOREM: gcd-plus-instance-temp
$(\gcd(a,\ a + b) = \gcd(a,\ b)) \wedge (\gcd(a,\ b + a) = \gcd(a,\ b))$

THEOREM: gcd-plus-proof
$((b\ \mathbf{mod}\ a) = \mathtt{0}) \rightarrow (\gcd(a,\ b + c) = \gcd(a,\ c))$

THEOREM: gcd-plus
$((b\ \mathbf{mod}\ a) = \mathtt{0})$
$\rightarrow \quad ((\gcd(a,\ b + c) = \gcd(a,\ c))$
$\wedge \quad (\gcd(a,\ c + b) = \gcd(a,\ c))$
$\wedge \quad (\gcd(b + c,\ a) = \gcd(a,\ c))$
$\wedge \quad (\gcd(c + b,\ a) = \gcd(a,\ c)))$

THEOREM: gcd-plus-instance
$(\gcd(a,\ a + b) = \gcd(a,\ b)) \wedge (\gcd(a,\ b + a) = \gcd(a,\ b))$

THEOREM: remainder-gcd
$((a\ \mathbf{mod}\ \gcd(a,\ b)) = \mathtt{0}) \wedge ((b\ \mathbf{mod}\ \gcd(a,\ b)) = \mathtt{0})$

THEOREM: distributivity-of-times-over-gcd-proof
$\gcd(x * z,\ y * z) = (z * \gcd(x,\ y))$

THEOREM: distributivity-of-times-over-gcd
$(\gcd(x * z,\ y * z) = (z * \gcd(x,\ y)))$
$\wedge \quad (\gcd(z * x,\ y * z) = (z * \gcd(x,\ y)))$
$\wedge \quad (\gcd(x * z,\ z * y) = (z * \gcd(x,\ y)))$
$\wedge \quad (\gcd(z * x,\ z * y) = (z * \gcd(x,\ y)))$

THEOREM: gcd-is-the-greatest
$((x \not\simeq \mathtt{0}) \wedge (y \not\simeq \mathtt{0}) \wedge ((x\ \mathbf{mod}\ z) = \mathtt{0}) \wedge ((y\ \mathbf{mod}\ z) = \mathtt{0}))$
$\rightarrow \quad (z \leq \gcd(x,\ y))$

THEOREM: common-divisor-divides-gcd
$(((x \textbf{ mod } z) = 0) \wedge ((y \textbf{ mod } z) = 0)) \rightarrow ((\gcd(x, y) \textbf{ mod } z) = 0)$

```
; We prove ASSOCIATIVITY-OF-GCD and COMMUTATIVITY2-OF-GCD roughly the same way.
; Use GCD-IS-THE-GREATEST twice to show that each side of the equality is
; less than or equal to the other side.
```

THEOREM: associativity-of-gcd-zero-case
$((a \simeq 0) \vee (b \simeq 0) \vee (c \simeq 0)) \rightarrow (\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c)))$

THEOREM: associativity-of-gcd
$\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$

THEOREM: commutativity2-of-gcd-zero-case
$((a \simeq 0) \vee (b \simeq 0) \vee (c \simeq 0)) \rightarrow (\gcd(b, \gcd(a, c)) = \gcd(a, \gcd(b, c)))$

THEOREM: commutativity2-of-gcd
$\gcd(b, \gcd(a, c)) = \gcd(a, \gcd(b, c))$

THEOREM: gcd-x-x
$\gcd(x, x) = \mathrm{fix}(x)$

THEOREM: gcd-idempotence
$(\gcd(x, \gcd(x, y)) = \gcd(x, y)) \wedge (\gcd(y, \gcd(x, y)) = \gcd(x, y))$

EVENT: Let us define the theory *gcds* to consist of the following events: commutativity2-of-gcd, associativity-of-gcd, common-divisor-divides-gcd, distributivity-of-times-over-gcd, lessp-gcd, equal-gcd-0, gcd-0, gcd-idempotence, gcd-x-x, remainder-gcd, gcd-plus, gcd-plus-instance, gcd-1, commutativity-of-gcd.

EVENT: Let us define the theory *naturals* to consist of the following events: addition, multiplication, remainders, quotients, exponentiation, logs, gcds.

```
; ------------------------------------------------------------
; was integers.events
; ------------------------------------------------------------


;; By Matt Kaufmann, modified from earlier integer library of Bill
;; Bevier and Matt Wilding.  A few functions (even ILESSP) have
;; been changed, but I expect the functionality of this library to
;; include all the functionality of the old one in most or even all
;; cases.
```

```
;; Modified from /local/src/nqthm-libs/integers.events to get ILEQ
;; expressed in terms of ILESSP and IDIFFERENCE in terms of INEG and
;; IPLUS.  There are other changes too.  The highlights are the new
;; metalemmas.

;; I'm going to leave the eval$ rules on that are proved here, and
;; leave eval$ off.

;; My intention is that this library be used in a mode in which ILEQ
;; and IDIFFERENCE are left enabled.  Otherwise, the aforementioned
;; meta lemmas may not be very useful, and also a number of additional
;; replacement rules may be needed.

;; There are three theories created by this library.  INTEGER-DEFNS is
;; a list of definitions of all integer functions (not including the
;; cancellation metafunctions and their auxiliaries, though), except
;; that ILEQ and IDIFFERENCE have been omitted.  This is a useful
;; theory for an ENABLE-THEORY hint when one simply wants to blast all
;; integer functions open, and it's also useful if one wants to close
;; them down with a DISABLE-THEORY hint (perhaps to go with an
;; (ENABLE-THEORY T) hint).  Second, ALL-INTEGER-DEFNS is the same as
;; INTEGER-DEFNS except that ILEQ and IDIFFERENCE are included in this
;; one.  Finally, INTEGERS is a list of all events to be "exported as
;; enabled" from this file when working in a mode where everything not
;; enabled by an ENABLE-THEORY hint is to be disabled.  Notice that
;; some rewrite rules have been included that might appear to be
;; unnecessary in light of the metalemmas; that's because metalemmas
;; only work on tame terms.  However, there's no guarantee that the
;; rewrite rules alone will prove very useful (on non-tame terms).
;; Also notice that INTEGER-DEFNS is disjoint from INTEGERS, since we
;; expect the basic definitions (other than ILEQ and IDIFFERENCE) to
;; remain disabled.

;; It's easy to see what I have and haven't placed in INTEGERS, since
;; I'll simply comment out the event names that I want to exclude (see
;; end of this file).

;; One might wish to consider changing (fix-int (minus ...)) in some
;; of the definitions below to (ineg ...).

;; The following meta rules are in this library.
;; (A little documentation added by Matt Wilding July 90)
;;
```

```
;; CORRECTNESS-OF-CANCEL-INEG
;;  cancel the first argument of an iplus term with a member of the second
;;  argument.
;;
;;  ex:  (iplus (ineg y) (iplus (ineg x) (iplus y z)))
;;     -->
;;       (iplus (ineg x) (fix-int z))
;;
;; CORRECTNESS-OF-CANCEL-IPLUS
;;  cancel the sides of an equality of iplus sums
;;
;;  ex:  (equal (iplus x (iplus y z)) (iplus a (iplus z x)))
;;     -->
;;       (equal (fix-int y) (fix-int a))
;;
;; CORRECTNESS-OF-CANCEL-IPLUS-ILESSP
;;  cancel the sides of an ilessp inequality of sums
;;
;;  ex:  (ilessp (iplus x (iplus y z)) (iplus a (iplus z x)))
;;     -->
;;       (ilessp y a)
;;
;; CORRECTNESS-OF-CANCEL-ITIMES
;;  cancel the sides of an equality of itimes products
;;
;;  ex: (equal (itimes x (itimes y z)) (itimes a (itimes z x)))
;;     -->
;;      (if (equal (itimes x z) '0)
;;        t
;;        (equal (fix-int y) (fix-int a)))
;;
;; CORRECTNESS-OF-CANCEL-ITIMES-ILESSP
;;  cancel the sides of an inequality of itimes products
;;
;;  ex: (ilessp (itimes x (itimes y z)) (itimes a (itimes z x)))
;;     -->
;;      (if (ilessp (itimes x z) '0)
;;         (ilessp a y)
;;        (if (ilessp 0 (itimes x z))
;;           (ilessp y a)
;;           f))
;;
;; CORRECTNESS-OF-CANCEL-ITIMES-FACTORS
;;  cancel factors in equality terms
```

```
;;  ex: (equal (iplus (itimes x y) x) (itimes z x))
;;     -->
;;      (if (equal (fix-int x) '0)
;;          t
;;        (equal (fix-int (plus y 1)) (fix-int z)))
;;
;; CORRECTNESS-OF-CANCEL-ITIMES-ILESSP-FACTORS
;;  cancel factors in ilessp terms
;;  ex: (equal (iplus (itimes x y) x) (itimes z x))
;;     -->
;;      (if (ilessp x '0)
;;          (ilessp z (iplus y 1))
;;        (if (ilessp '0 x)
;;            (ilessp (iplus y '1) z)
;;          f))
;;
;; CORRECTNESS-OF-CANCEL-FACTORS-0
;;  factor one side of equality when other side is constant 0
;;
;;  ex: (equal (iplus x (itimes x y)) '0)
;;     -->
;;      (or (equal (fix-int (iplus '1 y)) '0)
;;          (equal (fix-int x) '0))
;;
;; CORRECTNESS-OF-CANCEL-FACTORS-ILESSP-0
;;  factor one side of inequality when other side is constant 0
;;
;;  ex: (ilessp (iplus x (itimes x y)) '0)
;;     -->
;;      (or (and (ilessp (iplus '1 y) '0)
;;               (ilessp '0 x))
;;          (and (ilessp '0 (iplus '1 y))
;;               (ilessp x '0)))
;;
;; CORRECTNESS-OF-CANCEL-INEG-TERMS-FROM-EQUALITY
;;  rewrite equality to remove ineg terms
;;
;;  ex: (equal (iplus (ineg x) (ineg y)) (iplus (ineg z) w))
;;     -->
;;      (equal (fix-int z) (iplus x (iplus y w)))
;;
;; CORRECTNESS-OF-CANCEL-INEG-TERMS-FROM-INEQUALITY
;;  rewrite inequalities to remove ineg terms
;;
```

```
;;  ex: (ilessp (iplus (ineg x) (ineg y)) (iplus (ineg z) w))
;;     -->
;;      (ilessp (fix-int z) (iplus x (iplus y w)))


;(note-lib "/local/src/nqthm-libs/naturals")

;(compile-uncompiled-defns "xxx")

; ------------------------------------------------------------------------------
; Integers
; ------------------------------------------------------------------------------

#| The function below has no AND or OR, for efficiency
(defn integerp (x)
  (or (numberp x)
      (and (negativep x)
   (not (zerop (negative-guts x))))))
|#
```

DEFINITION:
integerp (x)
= **if** $x \in \mathbf{N}$ **then t**
elseif negativep (x) **then** negative-guts $(x) \not\simeq$ 0
else f endif

DEFINITION:
fix-int (x)
= **if** integerp (x) **then** x
else 0 **endif**

```
;; Even though I'll include a definition for izerop here, I'll
;; often avoid using it.
```

DEFINITION: izerop (i) = (fix-int (i) = 0)

```
#| old version:
(defn izerop (i)
  (if (integerp i)
      (equal i 0)
      t))
|#
```

DEFINITION:
ilessp $(i,\ j)$
$=$ **if** negativep (i)
then if negativep (j) **then** negative-guts (j) $<$ negative-guts (i)
elseif $i = (-\ 0)$ **then** $0 < j$
else t endif
elseif negativep (j) **then f**
else $i < j$ **endif**

DEFINITION: ileq $(i,\ j) = (\neg$ ilessp $(j,\ i))$

DEFINITION:
iplus $(x,\ y)$
$=$ **if** negativep (x)
then if negativep (y)
then if (negative-guts $(x) \simeq 0) \wedge$ (negative-guts $(y) \simeq 0)$ **then** 0
else $-$ (negative-guts (x) + negative-guts (y)) **endif**
elseif $y <$ negative-guts (x) **then** $-$ (negative-guts (x) $-$ $y)$
else $y -$ negative-guts (x) **endif**
elseif negativep (y)
then if $x <$ negative-guts (y) **then** $-$ (negative-guts (y) $-$ $x)$
else $x -$ negative-guts (y) **endif**
else $x + y$ **endif**

DEFINITION:
ineg (x)
$=$ **if** negativep (x) **then** negative-guts (x)
elseif $x \simeq 0$ **then** 0
else $-\ x$ **endif**

DEFINITION: idifference $(x,\ y)$ = iplus $(x,$ ineg $(y))$

DEFINITION:
iabs (i)
$=$ **if** negativep (i) **then** negative-guts (i)
else fix (i) **endif**

DEFINITION:
itimes $(i,\ j)$
$=$ **if** negativep (i)
then if negativep (j) **then** negative-guts $(i) *$ negative-guts (j)
else fix-int $(-$ (negative-guts $(i) * j))$ **endif**
elseif negativep (j) **then** fix-int $(-\ (i *$ negative-guts $(j)))$
else $i * j$ **endif**

DEFINITION:
iquotient (i, j)
= **if** fix-int $(j) =$ `0` **then** `0`
elseif negativep (i)
then if negativep (j)
then if (negative-guts (i) **mod** negative-guts $(j)) =$ `0`
then negative-guts $(i) \div$ negative-guts (j)
else $1 +$ (negative-guts $(i) \div$ negative-guts $(j))$ **endif**
elseif (negative-guts (i) **mod** $j) =$ `0`
then fix-int $(-$ (negative-guts $(i) \div j))$
else fix-int $(-$ $(1 +$ (negative-guts $(i) \div j)))$ **endif**
elseif negativep (j) **then** fix-int $(-$ $(i \div$ negative-guts $(j)))$
else $i \div j$ **endif**

DEFINITION:
iremainder (i, j) = idifference $(i,$ itimes $(j,$ iquotient $(i, j)))$

DEFINITION:
idiv (i, j)
= **if** fix-int $(j) =$ `0` **then** `0`
elseif negativep (i)
then if negativep (j) **then** negative-guts $(i) \div$ negative-guts (j)
elseif (negative-guts (i) **mod** $j) =$ `0`
then fix-int $(-$ (negative-guts $(i) \div j))$
else fix-int $(-$ $(1 +$ (negative-guts $(i) \div j)))$ **endif**
elseif negativep (j)
then if $(i$ **mod** negative-guts $(j)) =$ `0`
then fix-int $(-$ $(i \div$ negative-guts $(j)))$
else fix-int $(-$ $(1 +$ $(i \div$ negative-guts $(j))))$ **endif**
else $i \div j$ **endif**

DEFINITION:
imod (i, j) = idifference (fix-int (i), itimes $(j,$ idiv $(i, j)))$

DEFINITION:
iquo (i, j)
= **if** fix-int $(j) =$ `0` **then** `0`
elseif negativep (i)
then if negativep (j) **then** negative-guts $(i) \div$ negative-guts (j)
else fix-int $(-$ (negative-guts $(i) \div j))$ **endif**
elseif negativep (j) **then** fix-int $(-$ $(i \div$ negative-guts $(j)))$
else $i \div j$ **endif**

DEFINITION:
irem (i, j) = idifference (fix-int (i), itimes $(j,$ iquo $(i, j)))$

```
; ---------- DEFTHEORY events for definitions ----------
```

EVENT: Let us define the theory *integer-defns* to consist of the following events: integerp, fix-int, ilessp, iplus, ineg, iabs, itimes, iquotient, iremainder, idiv, imod, iquo, irem.

EVENT: Let us define the theory *all-integer-defns* to consist of the following events: integerp, fix-int, izerop, ilessp, ileq, iplus, ineg, idifference, iabs, itimes, iquotient, iremainder, idiv, imod, iquo, irem.

EVENT: Disable integerp.

EVENT: Disable fix-int.

EVENT: Disable ilessp.

EVENT: Disable iplus.

EVENT: Disable ineg.

EVENT: Disable iabs.

EVENT: Disable itimes.

```
;; I've disabled the rest later in the file, just because the lemmas
;; about division were (re-)proved with the remaining functions enabled.

; ---------- INTEGERP ----------
```

THEOREM: integerp-fix-int
integerp (fix-int (x))

THEOREM: integerp-iplus
integerp (iplus $(x,\ y)$)

THEOREM: integerp-idifference
integerp (idifference $(x,\ y)$)

THEOREM: integerp-ineg
integerp (ineg (*x*))

THEOREM: integerp-iabs
integerp (iabs (*x*))

THEOREM: integerp-itimes
integerp (itimes (*x*, *y*))

```
; ---------- FIX-INT ----------

;; The first of these, FIX-INT-REMOVER, is potentially dangerous from
;; a backchaining point of view, but I believe it's necessary.  At least
;; the lemmas below it should go a long way toward preventing its application.
```

THEOREM: fix-int-remover
integerp (*x*) → (fix-int (*x*) = *x*)

THEOREM: fix-int-fix-int
fix-int (fix-int (*x*)) = fix-int (*x*)

THEOREM: fix-int-iplus
fix-int (iplus (*a*, *b*)) = iplus (*a*, *b*)

THEOREM: fix-int-idifference
fix-int (idifference (*a*, *b*)) = idifference (*a*, *b*)

THEOREM: fix-int-ineg
fix-int (ineg (*x*)) = ineg (*x*)

THEOREM: fix-int-iabs
fix-int (iabs (*x*)) = iabs (*x*)

THEOREM: fix-int-itimes
fix-int (itimes (*x*, *y*)) = itimes (*x*, *y*)

```
; ---------- INEG ----------
```

THEOREM: ineg-iplus
ineg (iplus (*a*, *b*)) = iplus (ineg (*a*), ineg (*b*))

THEOREM: ineg-ineg
ineg (ineg (*x*)) = fix-int (*x*)

THEOREM: ineg-fix-int
ineg (fix-int (*x*)) = ineg (*x*)

THEOREM: ineg-of-non-integerp
(¬ integerp (*x*)) → (ineg (*x*) = 0)

```
;; I don't want the backchaining to slow down the prover.
```

EVENT: Disable ineg-of-non-integerp.

THEOREM: ineg-0
ineg (0) = 0

```
; ---------- IPLUS ----------

;; The first two of these really aren't necessary, in light
;; of the cancellation metalemma.
```

THEOREM: iplus-left-id
(¬ integerp (*x*)) → (iplus (*x*, *y*) = fix-int (*y*))

```
;; I don't want the backchaining to slow down the prover.
```

EVENT: Disable iplus-left-id.

THEOREM: iplus-right-id
(¬ integerp (*y*)) → (iplus (*x*, *y*) = fix-int (*x*))

```
;; I don't want the backchaining to slow down the prover.
```

EVENT: Disable iplus-right-id.

THEOREM: iplus-0-left
iplus (0, *x*) = fix-int (*x*)

THEOREM: iplus-0-right
iplus (*x*, 0) = fix-int (*x*)

THEOREM: commutativity2-of-iplus
iplus (*x*, iplus (*y*, *z*)) = iplus (*y*, iplus (*x*, *z*))

THEOREM: commutativity-of-iplus
iplus (*x*, *y*) = iplus (*y*, *x*)

THEOREM: associativity-of-iplus
iplus (iplus (*x*, *y*), *z*) = iplus (*x*, iplus (*y*, *z*))

THEOREM: iplus-cancellation-1
(iplus (*a*, *b*) = iplus (*a*, *c*)) = (fix-int (*b*) = fix-int (*c*))

THEOREM: iplus-cancellation-2
(iplus (*b*, *a*) = iplus (*c*, *a*)) = (fix-int (*b*) = fix-int (*c*))

THEOREM: iplus-ineg1
iplus (ineg (*a*), *a*) = 0

THEOREM: iplus-ineg2
iplus (*a*, ineg (*a*)) = 0

THEOREM: iplus-fix-int1
iplus (fix-int (*a*), *b*) = iplus (*a*, *b*)

THEOREM: iplus-fix-int2
iplus (*a*, fix-int (*b*)) = iplus (*a*, *b*)

```
; ---------- IDIFFERENCE ----------

;; mostly omitted, but I'll keep a few
```

THEOREM: idifference-fix-int1
idifference (fix-int (*a*), *b*) = idifference (*a*, *b*)

THEOREM: idifference-fix-int2
idifference (*a*, fix-int (*b*)) = idifference (*a*, *b*)

```
; ------------------------------------------------------------------------------
; Cancel INEG
; ------------------------------------------------------------------------------

;; We assume that the given term (IPLUS x y) has the property that y has already
;; been reduced and x is not an iplus-term.  So, the only question is whether
;; or not the formal negative of x appears in the fringe of y.

#| The function below has no AND or OR, for efficiency
(defn cancel-ineg-aux (x y)
  ;; returns nil or else a new term provably equal to (IPLUS x y)
  (if (and (listp x)
   (equal (car x) 'ineg))
      (cond
```

```
       ((equal y (cadr x))
''0)
       ((and (listp y)
     (equal (car y) 'iplus))
(let ((y1 (cadr y)) (y2 (caddr y)))
  (if (equal y1 (cadr x))
      (list 'fix-int y2)
    (let ((z (cancel-ineg-aux x y2)))
      (if z
  (list 'iplus y1 z)
f)))))
       (t f))
    (cond
     ((nlistp y)
      f)
     ((equal (car y) 'ineg)
      (if (equal x (cadr y))
  ''0
f))
     ((equal (car y) 'iplus)
      (let ((y1 (cadr y)) (y2 (caddr y)))
(if (and (listp y1)
 (equal (car y1) 'ineg)
 (equal x (cadr y1)))
    (list 'fix-int y2)
  (let ((z (cancel-ineg-aux x y2)))
    (if z
(list 'iplus y1 z)
      f)))))
     (t f))))
|#
```

DEFINITION:
cancel-ineg-aux $(x,\ y)$
= **if** listp (x)
then if car (x) = 'ineg
then if y = cadr (x) **then** ''0
elseif listp (y)
then if car (y) = 'iplus
then if cadr (y) = cadr (x)
then list ('fix-int, caddr (y))
elseif cancel-ineg-aux $(x,$ caddr $(y))$
then list ('iplus,

```
                                    cadr (y),
                                    cancel-ineg-aux (x, caddr (y)))
                            else f endif
                    else f endif
            else f endif
        elseif y ≃ nil then f
        elseif car (y) = 'ineg
        then if x = cadr (y) then ''0
            else f endif
        elseif car (y) = 'iplus
        then if listp (cadr (y))
            then if caadr (y) = 'ineg
                then if x = cadadr (y)
                    then list ('fix-int, caddr (y))
                    elseif cancel-ineg-aux (x, caddr (y))
                    then list ('iplus,
                                    cadr (y),
                                    cancel-ineg-aux (x, caddr (y)))
                    else f endif
                elseif cancel-ineg-aux (x, caddr (y))
                then list ('iplus, cadr (y), cancel-ineg-aux (x, caddr (y)))
                else f endif
            elseif cancel-ineg-aux (x, caddr (y))
            then list ('iplus, cadr (y), cancel-ineg-aux (x, caddr (y)))
            else f endif
        else f endif
elseif y ≃ nil then f
elseif car (y) = 'ineg
then if x = cadr (y) then ''0
        else f endif
elseif car (y) = 'iplus
then if listp (cadr (y))
        then if caadr (y) = 'ineg
            then if x = cadadr (y) then list ('fix-int, caddr (y))
                elseif cancel-ineg-aux (x, caddr (y))
                then list ('iplus, cadr (y), cancel-ineg-aux (x, caddr (y)))
                else f endif
            elseif cancel-ineg-aux (x, caddr (y))
            then list ('iplus, cadr (y), cancel-ineg-aux (x, caddr (y)))
            else f endif
        elseif cancel-ineg-aux (x, caddr (y))
        then list ('iplus, cadr (y), cancel-ineg-aux (x, caddr (y)))
        else f endif
else f endif
```

```
#| The function below has no AND or OR, for efficiency
(defn cancel-ineg (x)
  (if (and (listp x)
   (equal (car x) 'iplus))
      (let ((temp (cancel-ineg-aux (cadr x) (caddr x))))
(if temp
    temp
  x))
    x))
|#
```

DEFINITION:
cancel-ineg (x)
= **if** listp (x)
then if car (x) = `'iplus`
then if cancel-ineg-aux (cadr (x), caddr (x))
then cancel-ineg-aux (cadr (x), caddr (x))
else x **endif**
else x **endif**
else x **endif**

```
;; It seems a big win to turn off eval$.  I'll leave the recursive step out in
;; hopes that rewrite-eval$ handles it OK.
```

THEOREM: eval$-list-cons
eval$ (`'list`, cons (x, y), a) = cons (eval$ (**t**, x, a), eval$ (`'list`, y, a))

THEOREM: eval$-list-nlistp
$(x \simeq$ **nil**$) \rightarrow$ (eval$ (`'list`, x, a) = **nil**)

THEOREM: eval$-litatom
litatom $(x) \rightarrow$ (eval$ (**t**, x, a) = cdr (assoc (x, a)))

```
#|
(prove-lemma eval$-quotep (rewrite)
  (equal (eval$ t (list 'quote x) a)
 x))
|#

;; In place of the above I'll do the following, from
;; the naturals library.
```

EVENT: Enable eval$-quote.

THEOREM: eval$-other
$((\neg \text{litatom}(x)) \wedge (x \simeq \textbf{nil})) \rightarrow (\text{eval\$}(\textbf{t}, x, a) = x)$

EVENT: Disable eval$.

```
;; What I'd like to do is say what (eval$ t (cancel-ineg-aux x y) a),
;; but a rewrite rule will loop because of the recursion.  So I
;; introduce a silly auxiliary function so that the opening-up
;; heuristics can help me.  The function body has (listp y) tests
;; so that it can be accepted.
```

DEFINITION:
eval$-cancel-ineg-aux-fn (x, y, a)
= **if** listp $(x) \wedge (\text{car}(x) =$ `'ineg`$)$
then if $y = \text{cadr}(x)$ **then** `0`
else let *y1* **be** cadr (y),
y2 **be** caddr (y)
in
if *y1* $= \text{cadr}(x)$ **then** fix-int (eval$ $(\textbf{t}, y2, a)$)
elseif listp (y)
then iplus (eval$ $(\textbf{t}, y1, a)$,
eval$-cancel-ineg-aux-fn $(x, y2, a)$)
else `0` **endif endlet endif**
else if car $(y) =$ `'ineg` **then** `0`
else let *y1* **be** cadr (y),
y2 **be** caddr (y)
in
if listp $(y1)$
$\wedge$ $(\text{car}(y1) =$ `'ineg`$)$
$\wedge$ $(x = \text{cadr}(y1))$
then fix-int (eval$ $(\textbf{t}, y2, a)$)
elseif listp (y)
then iplus (eval$ $(\textbf{t}, y1, a)$,
eval$-cancel-ineg-aux-fn $(x,$
$y2,$
$a))$
else `0` **endif endlet endif endif**

THEOREM: eval$-cancel-ineg-aux-is-its-fn
(cancel-ineg-aux $(x, y) \neq \textbf{f}$)
$\rightarrow$ (eval$ $(\textbf{t}$, cancel-ineg-aux (x, y), $a) =$ eval$-cancel-ineg-aux-fn (x, y, a))

THEOREM: iplus-ineg3
iplus (ineg (x), iplus $(x, y)) =$ fix-int (y)

THEOREM: iplus-ineg4
$\text{iplus}(x, \text{iplus}(\text{ineg}(x), y)) = \text{fix-int}(y)$

THEOREM: iplus-ineg-promote
$\text{iplus}(y, \text{ineg}(x)) = \text{iplus}(\text{ineg}(x), y)$

THEOREM: iplus-x-y-ineg-x
$\text{iplus}(x, \text{iplus}(y, \text{ineg}(x))) = \text{fix-int}(y)$

EVENT: Disable iplus-ineg-promote.

THEOREM: correctness-of-cancel-ineg-aux
$(\text{cancel-ineg-aux}(x, y) \neq \mathbf{f})$
$\rightarrow \quad (\text{eval\$-cancel-ineg-aux-fn}(x, y, a) = \text{iplus}(\text{eval\$}(\mathbf{t}, x, a), \text{eval\$}(\mathbf{t}, y, a)))$

THEOREM: correctness-of-cancel-ineg
$\text{eval\$}(\mathbf{t}, x, a) = \text{eval\$}(\mathbf{t}, \text{cancel-ineg}(x), a)$

EVENT: Disable correctness-of-cancel-ineg-aux.

```
; ------------------------------------------------------------------------------
; Cancel IPLUS
; ------------------------------------------------------------------------------

;; All I do here is cancel like terms from both sides.  The problem of handling
;; INEG cancellation IS handled completely separately above.  That hasn't always
;; been the case -- in my first try I attempted to integrate the operations.
;; But now I see that for things like (equal z (iplus x (iplus y (ineg x))))
;; the integrated approach will fail.  Also, thanks to Matt Wilding, for pointing
;; out that the "four squares" example that Bill Pase sent me ran faster with
;; the newer approach (on his previously-implemented version for the rationals).

#| The function below has no AND or OR, for efficiency
(defn iplus-fringe (x)
  (if (and (listp x)
   (equal (car x)
  (quote iplus)))
      (append (iplus-fringe (cadr x))
      (iplus-fringe (caddr x)))
      (cons x nil)))
|#
```

DEFINITION:

iplus-fringe (x)
= **if** listp (x)
  then if car (x) = `'iplus`
    then append (iplus-fringe (cadr (x)), iplus-fringe (caddr (x)))
    else list (x) **endif**
  else list (x) **endif**

THEOREM: lessp-count-listp-cdr
listp (cdr (x)) → (count (cdr (x)) < count (x))

DEFINITION:
iplus-tree-rec (l)
= **if** cdr (l) ≃ **nil** **then** car (l)
  else list (`'iplus`, car (l), iplus-tree-rec (cdr (l))) **endif**

DEFINITION:
iplus-tree (l)
= **if** listp (l)
  then if listp (cdr (l)) **then** iplus-tree-rec (l)
    else list (`'fix-int`, car (l)) **endif**
  else `''0` **endif**

DEFINITION:
iplus-list (x)
= **if** listp (x) **then** iplus (car (x), iplus-list (cdr (x)))
  else `0` **endif**

THEOREM: integerp-iplus-list
integerp (iplus-list (x))

THEOREM: eval$-iplus-tree-rec
eval$ (**t**, iplus-tree-rec (x), a)
= **if** listp (x)
  then if listp (cdr (x)) **then** iplus-list (eval$ (`'list`, x, a))
    else eval$ (**t**, car (x), a) **endif**
  else `0` **endif**

THEOREM: eval$-iplus-tree
eval$ (**t**, iplus-tree (x), a) = iplus-list (eval$ (`'list`, x, a))

THEOREM: eval$-list-append
eval$ (`'list`, append (x, y), a)
= append (eval$ (`'list`, x, a), eval$ (`'list`, y, a))

```
#| The function below has no AND or OR, for efficiency
(defn cancel-iplus (x)
  (if (and (listp x)
   (equal (car x) (quote equal)))
      (if (and (listp (cadr x))
       (equal (caadr x) (quote iplus))
       (listp (caddr x))
       (equal (caaddr x) (quote iplus)))
  (let ((xs (iplus-fringe (cadr x)))
(ys (iplus-fringe (caddr x))))
    (let ((bagint (bagint xs ys)))
      (if (listp bagint)
  (list (quote equal)
(iplus-tree (bagdiff xs bagint))
(iplus-tree (bagdiff ys bagint)))
x)))
(if (and (listp (cadr x))
 (equal (caadr x) (quote iplus))
 ;; We don't want to introduce the IF below unless something
 ;; is "gained", or else we may get into an infinite rewriting loop.
 (member (caddr x) (iplus-fringe (cadr x))))
    (list (quote if)
  (list (quote integerp) (caddr x))
  (list (quote equal)
(iplus-tree (delete (caddr x) (iplus-fringe (cadr x))))
''0)
  (list (quote quote) f))
  (if (and (listp (caddr x))
   (equal (caaddr x) (quote iplus))
   (member (cadr x) (iplus-fringe (caddr x))))
      (list (quote if)
    (list (quote integerp) (cadr x))
    (list (quote equal)
  ''0
  (iplus-tree (delete (cadr x) (iplus-fringe (caddr x)))))
    (list (quote quote) f))
    x)))
    x))
|#
```

Definition:
cancel-iplus (x)
= **if** listp (x)

```
then if car (x) = 'equal
     then if listp (cadr (x))
          then if caadr (x) = 'iplus
               then if listp (caddr (x))
                    then if caaddr (x) = 'iplus
                         then if listp (bagint (iplus-fringe (cadr (x)),
                                                iplus-fringe (caddr (x))))
                              then list ('equal,
                                         iplus-tree (bagdiff (iplus-fringe (cadr (x)),
                                                              bagint (iplus-fringe (cadr (x)),
                                                                      iplus-fringe (caddr (x))))),
                                         iplus-tree (bagdiff (iplus-fringe (caddr (x)),
                                                              bagint (iplus-fringe (cadr (x)),
                                                                      iplus-fringe (caddr (x))))))
                              else x endif
                         elseif caddr (x) ∈ iplus-fringe (cadr (x))
                         then list ('if,
                                    list ('integerp, caddr (x)),
                                    cons ('equal,
                                          cons (iplus-tree (delete (caddr (x),
                                                                    iplus-fringe (cadr (x)))),
                                                '('0))),
                                    list ('quote, f))
                         else x endif
                    elseif caddr (x) ∈ iplus-fringe (cadr (x))
                    then list ('if,
                               list ('integerp, caddr (x)),
                               cons ('equal,
                                     cons (iplus-tree (delete (caddr (x),
                                                               iplus-fringe (cadr (x)))),
                                           '('0))),
                               list ('quote, f))
                    else x endif
               elseif listp (caddr (x))
               then if caaddr (x) = 'iplus
                    then if cadr (x) ∈ iplus-fringe (caddr (x))
                         then list ('if,
                                    list ('integerp, cadr (x)),
                                    list ('equal,
                                          ''0,
                                          iplus-tree (delete (cadr (x),
                                                              iplus-fringe (caddr (x))))),
                                    list ('quote, f))
                         else x endif
```

```
                    else x endif
               else x endif
          elseif listp(caddr(x))
          then if caaddr(x) = 'iplus
               then if cadr(x) ∈ iplus-fringe(caddr(x))
                    then list('if,
                              list('integerp, cadr(x)),
                              list('equal,
                                   ''0,
                                   iplus-tree(delete(cadr(x),
                                                     iplus-fringe(caddr(x))))),
                              list('quote, f))
                    else x endif
               else x endif
          else x endif
     else x endif
```

THEOREM: eval$-cancel-iplus

```
eval$(t, cancel-iplus(x), a)
=    if listp(x) ∧ (car(x) = 'equal)
     then if listp(cadr(x))
               ∧   (caadr(x) = 'iplus)
               ∧   listp(caddr(x))
               ∧   (caaddr(x) = 'iplus)
          then let xs be iplus-fringe(cadr(x)),
                   ys be iplus-fringe(caddr(x))
               in
               let bagint be bagint(xs, ys)
               in
               if listp(bagint)
               then iplus-list(eval$('list,
                                     bagdiff(xs,
                                             bagint(xs, ys)),
                                     a))
                    =   iplus-list(eval$('list,
                                         bagdiff(ys,
                                                 bagint(xs,
                                                        ys)),
                                         a))
               else eval$(t, x, a) endif endlet endlet
          elseif listp(cadr(x))
               ∧   (caadr(x) = 'iplus)
               ∧   (caddr(x) ∈ iplus-fringe(cadr(x)))
```

```
        then if integerp(eval$(t, caddr(x), a))
             then iplus-list(eval$('list,
                                   delete(caddr(x), iplus-fringe(cadr(x))),
                                   a))
                  = 0
             else f endif
        elseif listp(caddr(x))
             ∧ (caaddr(x) = 'iplus)
             ∧ (cadr(x) ∈ iplus-fringe(caddr(x)))
        then if integerp(eval$(t, cadr(x), a))
             then 0 = iplus-list(eval$('list,
                                       delete(cadr(x),
                                              iplus-fringe(caddr(x))),
                                       a))
             else f endif
        else eval$(t, x, a) endif
   else eval$(t, x, a) endif
```

EVENT: Disable cancel-iplus.

THEOREM: eval$-iplus-list-delete
$(z \in y)$
$\rightarrow$ (iplus-list (eval$ ('list, delete (z, y), a))
= idifference (iplus-list (eval$ ('list, y, a)), eval$ (t, z, a)))

THEOREM: eval$-iplus-list-bagdiff
subbagp (x, y)
$\rightarrow$ (iplus-list (eval$ ('list, bagdiff (y, x), a))
= idifference (iplus-list (eval$ ('list, y, a)),
iplus-list (eval$ ('list, x, a))))

THEOREM: iplus-list-append
iplus-list (append (x, y)) = iplus (iplus-list (x), iplus-list (y))

EVENT: Disable iplus-tree.

```
;; because we want to use EVAL$-IPLUS-TREE for now
```

THEOREM: not-integerp-implies-not-equal-iplus
$(\neg$ integerp $(a)) \rightarrow ((a =$ iplus $(b, c)) =$ f)

THEOREM: iplus-list-eval$-fringe
iplus-list (eval$ ('list, iplus-fringe (x), a)) = fix-int (eval$ (t, x, a))

```
;; The following two lemmas aren't needed but they sure do
;; shorten the total proof time!!!
```

THEOREM: iplus-ineg5-lemma-1
integerp (x)
$\rightarrow$ $((x = \text{iplus}(y, \text{iplus}(\text{ineg}(z), w)))$
$=$ $(x = \text{iplus}(\text{ineg}(z), \text{iplus}(y, w))))$

THEOREM: iplus-ineg5-lemma-2
$(\text{integerp}(x) \wedge \text{integerp}(v))$
$\rightarrow$ $((x = \text{iplus}(\text{ineg}(z), v)) = (\text{iplus}(x, z) = v))$

THEOREM: iplus-ineg5
integerp (x)
$\rightarrow$ $((x = \text{iplus}(y, \text{iplus}(\text{ineg}(z), w))) = (\text{iplus}(x, z) = \text{iplus}(y, w)))$

EVENT: Disable iplus-ineg5-lemma-1.

EVENT: Disable iplus-ineg5-lemma-2.

THEOREM: iplus-ineg6
integerp (x)
$\rightarrow$ $((x = \text{iplus}(y, \text{iplus}(w, \text{ineg}(z)))) = (\text{iplus}(x, z) = \text{iplus}(y, w)))$

THEOREM: eval$-iplus
$(\text{listp}(x) \wedge (\text{car}(x) = \texttt{'iplus}))$
$\rightarrow$ $(\text{eval\$}(\mathbf{t}, x, a) = \text{iplus}(\text{eval\$}(\mathbf{t}, \text{cadr}(x), a), \text{eval\$}(\mathbf{t}, \text{caddr}(x), a)))$

THEOREM: iplus-ineg7
$(0 = \text{iplus}(x, \text{ineg}(y))) = (\text{fix-int}(y) = \text{fix-int}(x))$

THEOREM: correctness-of-cancel-iplus
$\text{eval\$}(\mathbf{t}, x, a) = \text{eval\$}(\mathbf{t}, \text{cancel-iplus}(x), a)$

EVENT: Disable iplus-ineg5.

EVENT: Disable iplus-ineg6.

```
; ------------------------------------------------------------------------------
; Cancel IPLUS from ILESSP
; ------------------------------------------------------------------------------
```

```
;; This is similar to the cancellation of IPLUS terms from equalities,
;; handled above, and uses many of the same lemmas.  A small but definite
;; difference however is that for ILESSP we don't have to fix integers.

;; By luck we have that iplus-tree-rec is appropriate here, since
;; the lemma eval$-iplus-tree-rec shows that it (accidentally) behaves
;; properly on the empty list.
```

THEOREM: ilessp-fix-int-1
ilessp (fix-int (x), y) = ilessp (x, y)

THEOREM: ilessp-fix-int-2
ilessp (x, fix-int (y)) = ilessp (x, y)

```
;; Perhaps the easiest approach is to do everything with respect to the
;; same IPLUS-TREE function that we used before, and then once the
;; supposed meta-lemma is proved, go back and show that we get the
;; same answer if we use a version that doesn't fix-int singleton fringes.
```

DEFINITION:
make-cancel-iplus-inequality-1 (x, y)
= list ('ilessp,
iplus-tree (bagdiff (x, bagint (x, y))),
iplus-tree (bagdiff (y, bagint (x, y))))

```
#| The function below has no AND or OR, for efficiency
(defn cancel-iplus-ilessp-1 (x)
  (if (and (listp x)
   (equal (car x) (quote ilessp)))
      (make-cancel-iplus-inequality-1 (iplus-fringe (cadr x))
      (iplus-fringe (caddr x)))
    x))
|#
```

DEFINITION:
cancel-iplus-ilessp-1 (x)
= **if** listp (x)
then if car (x) = 'ilessp
then make-cancel-iplus-inequality-1 (iplus-fringe (cadr (x)),
iplus-fringe (caddr (x)))
else x **endif**
else x **endif**

```
;; Notice that IPLUS-TREE-NO-FIX-INT is currently enabled, which is
;; good since we want to use EVAL$-IPLUS-TREE-NO-FIX-INT for now.
```

THEOREM: lessp-difference-plus-arg1
$(w < ((w + y) - x)) = (x < y)$

THEOREM: lessp-difference-plus-arg1-commuted
$(w < ((y + w) - x)) = (x < y)$

THEOREM: iplus-cancellation-1-for-ilessp
ilessp (iplus (a, b), iplus (a, c)) = ilessp (b, c)

THEOREM: iplus-cancellation-2-for-ilessp
ilessp (iplus (b, a), iplus (c, a)) = ilessp (b, c)

THEOREM: correctness-of-cancel-iplus-ilessp-lemma
eval$ (**t**, x, a) = eval$ (**t**, cancel-iplus-ilessp-1 (x), a)

DEFINITION:
iplus-tree-no-fix-int (l)
= **if** listp (l) **then** iplus-tree-rec (l)
else ''0 **endif**

THEOREM: eval$-ilessp-iplus-tree-no-fix-int
ilessp (eval$ (**t**, iplus-tree-no-fix-int (x), a),
eval$ (**t**, iplus-tree-no-fix-int (y), a))
= ilessp (eval$ (**t**, iplus-tree (x), a), eval$ (**t**, iplus-tree (y), a))

EVENT: Disable iplus-tree-no-fix-int.

THEOREM: make-cancel-iplus-inequality-simplifier
eval$ (**t**, make-cancel-iplus-inequality-1 (x, y), a)
= eval$ (**t**,
list ('ilessp,
iplus-tree-no-fix-int (bagdiff (x, bagint (x, y))),
iplus-tree-no-fix-int (bagdiff (y, bagint (x, y)))),
a)

```
#| The function below has no AND or OR, for efficiency
(defn cancel-iplus-ilessp (x)
  (if (and (listp x)
   (equal (car x) (quote ilessp)))
      (let ((x1 (iplus-fringe (cadr x)))
    (y1 (iplus-fringe (caddr x))))
```

```
(let ((bagint (bagint x1 y1)))
  (if (listp bagint)
      ;; I check (listp bagint) only for efficiency
      (list (quote ilessp)
    (iplus-tree-no-fix-int (bagdiff x1 bagint))
    (iplus-tree-no-fix-int (bagdiff y1 bagint)))
    x)))
    x))
|#

;; **** Should perhaps check that some argument of the ILESSP has function
;; symbol IPLUS, or else we may wind up dealing with (ILESSP 0 0).  That should
;; be harmless enough, though, even if *1*IPLUS is disabled; we'll just get the
;; same term back, the hard way.
```

DEFINITION:
cancel-iplus-ilessp (x)
= **if** listp (x)
then if car (x) = 'ilessp
then if listp (bagint (iplus-fringe (cadr (x)),
iplus-fringe (caddr (x)))))
then list ('ilessp,
iplus-tree-no-fix-int (bagdiff (iplus-fringe (cadr (x))),
bagint (iplus-fringe (cadr (x))),
iplus-fringe (caddr (x))))),
iplus-tree-no-fix-int (bagdiff (iplus-fringe (caddr (x))),
bagint (iplus-fringe (cadr (x))),
iplus-fringe (caddr (x)))))))
else x **endif**
else x **endif**
else x **endif**

EVENT: Disable make-cancel-iplus-inequality-1.

THEOREM: correctness-of-cancel-iplus-ilessp
eval$ (**t**, x, a) = eval$ (**t**, cancel-iplus-ilessp (x), a)

```
; ---------- Multiplication ----------
```

THEOREM: itimes-zero1
(fix-int (x) = 0) → (itimes (x, y) = 0)

THEOREM: itimes-0-left
itimes (0, y) = 0

```
;; I don't want the backchaining to slow down the prover.
```

EVENT: Disable itimes-zero1.

THEOREM: itimes-zero2
(fix-int (y) = 0) → (itimes (x, y) = 0)

THEOREM: itimes-0-right
itimes (x, 0) = 0

```
;; I don't want the backchaining to slow down the prover.
```

EVENT: Disable itimes-zero2.

THEOREM: itimes-fix-int1
itimes (fix-int (a), b) = itimes (a, b)

THEOREM: itimes-fix-int2
itimes (a, fix-int (b)) = itimes (a, b)

THEOREM: commutativity-of-itimes
itimes (x, y) = itimes (y, x)

THEOREM: itimes-distributes-over-iplus-proof
itimes (x, iplus (y, z)) = iplus (itimes (x, y), itimes (x, z))

THEOREM: itimes-distributes-over-iplus
(itimes (x, iplus (y, z)) = iplus (itimes (x, y), itimes (x, z)))
∧ (itimes (iplus (x, y), z) = iplus (itimes (x, z), itimes (y, z)))

THEOREM: commutativity2-of-itimes
itimes (x, itimes (y, z)) = itimes (y, itimes (x, z))

THEOREM: associativity-of-itimes
itimes (itimes (x, y), z) = itimes (x, itimes (y, z))

THEOREM: equal-itimes-0
(itimes (x, y) = 0) = ((fix-int (x) = 0) ∨ (fix-int (y) = 0))

THEOREM: equal-itimes-1
(itimes (a, b) = 1)
= ((((a = 1) ∧ (b = 1)) ∨ ((a = -1) ∧ (b = -1)))

THEOREM: equal-itimes-minus-1
(itimes (*a*, *b*) = -1)
= (((*a* = -1) ∧ (*b* = 1)) ∨ ((*a* = 1) ∧ (*b* = -1)))

THEOREM: itimes-1-arg1
itimes (1, *x*) = fix-int (*x*)

```
; ---------- Division ----------
```

THEOREM: quotient-remainder-uniqueness
((*a* = (*r* + (*b* ∗ *q*))) ∧ (*r* < *b*))
→ ((fix (*r*) = (*a* **mod** *b*)) ∧ (fix (*q*) = (*a* ÷ *b*)))

```
; We want to define IQUOTIENT and IREMAINDER. The standard approach to
; integer division derives from from the following theorem.
;
; Division Theorem:
; For all integers i,j, j not 0, there exist unique integers q and r
; which satisfy  i = jq + r,  0 <= r < |j|.
;
; The functions IQUOTIENT and IREMAINDER are intended to compute q and r.
; Therefore, to be satisfied that we have the right definitions, we must
; prove the above theorem.
```

THEOREM: division-theorem-part1
integerp (*i*) → (iplus (iremainder (*i*, *j*), itimes (*j*, iquotient (*i*, *j*))) = *i*)

THEOREM: division-theorem-part2
(integerp (*j*) ∧ (*j* ≠ 0)) → (¬ ilessp (iremainder (*i*, *j*), 0))

THEOREM: division-theorem-part3
(integerp (*j*) ∧ (*j* ≠ 0)) → ilessp (iremainder (*i*, *j*), iabs (*j*))

THEOREM: division-theorem
(integerp (*i*) ∧ integerp (*j*) ∧ (*j* ≠ 0))
→ ((iplus (iremainder (*i*, *j*), itimes (*j*, iquotient (*i*, *j*))) = *i*)
∧ (¬ ilessp (iremainder (*i*, *j*), 0))
∧ ilessp (iremainder (*i*, *j*), iabs (*j*)))

THEOREM: quotient-difference-lessp-arg2
(((*a* **mod** *c*) = 0) ∧ (*b* < *c*))
→ (((*a* − *b*) ÷ *c*)
= **if** *b* ≃ 0 **then** *a* ÷ *c*
elseif *b* < *a* **then** (*a* ÷ *c*) − (1 + (*b* ÷ *c*))
else 0 **endif**)

THEOREM: iquotient-iremainder-uniqueness
(integerp (*i*)
 ∧ integerp (*j*)
 ∧ integerp (*r*)
 ∧ integerp (*q*)
 ∧ (*j* ≠ 0)
 ∧ (*i* = iplus (*r*, itimes (*j*, *q*)))
 ∧ (¬ ilessp (*r*, 0))
 ∧ ilessp (*r*, iabs (*j*)))
→ ((*r* = iremainder (*i*, *j*)) ∧ (*q* = iquotient (*i*, *j*)))

```
; It turns out that in computer arithmetic, notions of division other than that
; given by the division theorem are used. Two in particular, called
; "truncate towards negative infinity" and "truncate towards zero" are common.
; We present their definitions here.

; Division Theorem (truncate towards negative infinity variant):
;
; For all integers i,j, j not 0, there exist unique integers q and r
; which satisfy
;                i = jq + r,  0 <= r <  j  (j > 0)
;                             j <  r <= 0  (j < 0)
;
; In this version the integer quotient of two integers is the integer floor
; of the real quotient of the integers. The remainder has the sign of the
; divisor. The functions IDIV and IMOD are intended to compute q and r.
; Therefore, to be satisfied that we have the right definitions, we must
; prove the above theorem.
```

THEOREM: division-theorem-for-truncate-to-neginf-part1
integerp (*i*) → (iplus (imod (*i*, *j*), itimes (*j*, idiv (*i*, *j*))) = *i*)

THEOREM: division-theorem-for-truncate-to-neginf-part2
ilessp (0, *j*) → ((¬ ilessp (imod (*i*, *j*), 0)) ∧ ilessp (imod (*i*, *j*), *j*))

THEOREM: division-theorem-for-truncate-to-neginf-part3
(integerp (*j*) ∧ ilessp (*j*, 0))
→ ((¬ ilessp (0, imod (*i*, *j*))) ∧ ilessp (*j*, imod (*i*, *j*)))

THEOREM: division-theorem-for-truncate-to-neginf
(integerp (*i*) ∧ integerp (*j*) ∧ (*j* ≠ 0))
→ ((iplus (imod (*i*, *j*), itimes (*j*, idiv (*i*, *j*))) = *i*)
 ∧ **if** ilessp (0, *j*)
 then (¬ ilessp (imod (*i*, *j*), 0)) ∧ ilessp (imod (*i*, *j*), *j*)

else (¬ ilessp (0, imod (*i*, *j*)))
∧ ilessp (*j*, imod (*i*, *j*)) **endif**)

THEOREM: idiv-imod-uniqueness
(integerp (*i*)
∧ integerp (*j*)
∧ integerp (*r*)
∧ integerp (*q*)
∧ (*j* ≠ 0)
∧ (*i* = iplus (*r*, itimes (*j*, *q*)))
∧ **if** ilessp (0, *j*) **then** (¬ ilessp (*r*, 0)) ∧ ilessp (*r*, *j*)
else (¬ ilessp (0, *r*)) ∧ ilessp (*j*, *r*) **endif**)
→ ((*r* = imod (*i*, *j*)) ∧ (*q* = idiv (*i*, *j*)))

```
; Division Theorem (truncate towards zero variant):
;
; For all integers i,j, j not 0, there exist unique integers q and r
; which satisfy
;                i = jq + r,     0 <=  r  <  |j|  (i => 0)
;                              -|j| <   r  <= 0    (i < 0)
;
; In this version (iquo, irem), the integer quotient of two integers is the integer floor
; of the real quotient of the integers, if the real quotient is positive. If the
; real quotient is negative, the integer quotient is the integer ceiling of the
; real quotient. The remainder has the sign of the dividend. The functions IQUO
; and IREM are intended to compute q and r. Therefore, to be satisfied that we
; have the right definitions, we must prove the above theorem.
```

THEOREM: division-theorem-for-truncate-to-zero-part1
integerp (*i*) → (iplus (irem (*i*, *j*), itimes (*j*, iquo (*i*, *j*))) = *i*)

THEOREM: division-theorem-for-truncate-to-zero-part2
(integerp (*i*) ∧ integerp (*j*) ∧ (*j* ≠ 0) ∧ (¬ ilessp (*i*, 0)))
→ ((¬ ilessp (irem (*i*, *j*), 0)) ∧ ilessp (irem (*i*, *j*), iabs (*j*)))

THEOREM: division-theorem-for-truncate-to-zero-part3
(integerp (*i*) ∧ integerp (*j*) ∧ (*j* ≠ 0) ∧ ilessp (*i*, 0))
→ ((¬ ilessp (0, irem (*i*, *j*))) ∧ ilessp (ineg (iabs (*j*)), irem (*i*, *j*)))

THEOREM: division-theorem-for-truncate-to-zero
(integerp (*i*) ∧ integerp (*j*) ∧ (*j* ≠ 0))
→ ((iplus (irem (*i*, *j*), itimes (*j*, iquo (*i*, *j*))) = *i*)
∧ **if** ¬ ilessp (*i*, 0)
then (¬ ilessp (irem (*i*, *j*), 0))

$\wedge$ ilessp (irem (i, j), iabs (j))
else ($\neg$ ilessp (0, irem (i, j)))
$\wedge$ ilessp (ineg (iabs (j)), irem (i, j)) **endif**)

THEOREM: iquo-irem-uniqueness
(integerp (i)
$\wedge$ integerp (j)
$\wedge$ integerp (r)
$\wedge$ integerp (q)
$\wedge$ ($j \neq$ 0)
$\wedge$ (i = iplus (r, itimes (j, q)))
$\wedge$ **if** $\neg$ ilessp (i, 0) **then** ($\neg$ ilessp (r, 0)) $\wedge$ ilessp (r, iabs (j))
else ($\neg$ ilessp (0, r)) $\wedge$ ilessp (ineg (iabs (j)), r) **endif**)
$\rightarrow$ ((r = irem (i, j)) $\wedge$ (q = iquo (i, j))))

; ---------- Multiplication Facts

THEOREM: itimes-ineg-1
itimes (ineg (x), y) = ineg (itimes (x, y))

THEOREM: itimes-ineg-2
itimes (x, ineg (y)) = ineg (itimes (x, y))

THEOREM: itimes-cancellation-1
(itimes (a, b) = itimes (a, c))
= ((fix-int (a) = 0) $\vee$ (fix-int (b) = fix-int (c)))

THEOREM: itimes-cancellation-2
(itimes (b, a) = itimes (c, a))
= ((fix-int (a) = 0) $\vee$ (fix-int (b) = fix-int (c)))

THEOREM: itimes-cancellation-3
(itimes (a, b) = itimes (c, a))
= ((fix-int (a) = 0) $\vee$ (fix-int (b) = fix-int (c)))

; ---------- Division Facts

THEOREM: integerp-iquotient
integerp (iquotient (i, j))

THEOREM: integerp-iremainder
integerp (iremainder (i, j))

THEOREM: integerp-idiv
integerp (idiv (i, j))

THEOREM: integerp-imod
integerp (imod (*i*, *j*))

THEOREM: integerp-iquo
integerp (iquo (*i*, *j*))

THEOREM: integerp-irem
integerp (irem (*i*, *j*))

THEOREM: iquotient-fix-int1
iquotient (fix-int (*i*), *j*) = iquotient (*i*, *j*)

THEOREM: iquotient-fix-int2
iquotient (*i*, fix-int (*j*)) = iquotient (*i*, *j*)

THEOREM: iremainder-fix-int1
iremainder (fix-int (*i*), *j*) = iremainder (*i*, *j*)

THEOREM: iremainder-fix-int2
iremainder (*i*, fix-int (*j*)) = iremainder (*i*, *j*)

THEOREM: idiv-fix-int1
idiv (fix-int (*i*), *j*) = idiv (*i*, *j*)

THEOREM: idiv-fix-int2
idiv (*i*, fix-int (*j*)) = idiv (*i*, *j*)

THEOREM: imod-fix-int1
imod (fix-int (*i*), *j*) = imod (*i*, *j*)

THEOREM: imod-fix-int2
imod (*i*, fix-int (*j*)) = imod (*i*, *j*)

THEOREM: iquo-fix-int1
iquo (fix-int (*i*), *j*) = iquo (*i*, *j*)

THEOREM: iquo-fix-int2
iquo (*i*, fix-int (*j*)) = iquo (*i*, *j*)

THEOREM: irem-fix-int1
irem (fix-int (*i*), *j*) = irem (*i*, *j*)

THEOREM: irem-fix-int2
irem (*i*, fix-int (*j*)) = irem (*i*, *j*)

THEOREM: fix-int-iquotient
fix-int (iquotient (*i*, *j*)) = iquotient (*i*, *j*)

THEOREM: fix-int-iremainder
fix-int (iremainder (*i*, *j*)) = iremainder (*i*, *j*)

THEOREM: fix-int-idiv
fix-int (idiv (*i*, *j*)) = idiv (*i*, *j*)

THEOREM: fix-int-imod
fix-int (imod (*i*, *j*)) = imod (*i*, *j*)

THEOREM: fix-int-iquo
fix-int (iquo (*i*, *j*)) = iquo (*i*, *j*)

THEOREM: fix-int-irem
fix-int (irem (*i*, *j*)) = irem (*i*, *j*)

EVENT: Disable iquotient.

EVENT: Disable iremainder.

EVENT: Disable idiv.

EVENT: Disable imod.

EVENT: Disable iquo.

EVENT: Disable irem.

```
; ---------- Meta lemma for itimes cancellation

;; I tried to adapt this somewhat from corresponding meta lemmas in
;; naturals library, but it seemed to get hairy.  So instead I'll try
;; to parallel the development I gave for IPLUS.  I'll be lazier here
;; about efficiency, so I'll use a completely analogous definition of
;; itimes-tree.  Notice that I've avoided the IZEROP-TREE approach
;; from the naturals version, in that I simply create the appropriate
;; common fringe into a product and say that this product is non-zero
;; when dividing both sides by it.  It can then be up to the user whether
;; or not to enable the (meta or rewrite) rule that says that izerop of a product reduces
;; to the disjunction of izerop of the factors.

#| The function below has no AND or OR, for efficiency
(defn itimes-fringe (x)
```

```
  (if (and (listp x)
   (equal (car x)
  (quote itimes)))
      (append (itimes-fringe (cadr x))
      (itimes-fringe (caddr x)))
      (cons x nil)))
|#
```

DEFINITION:
itimes-fringe (x)
= **if** listp (x)
then if car (x) = `'itimes`
then append (itimes-fringe (cadr (x)), itimes-fringe (caddr (x)))
else list (x) **endif**
else list (x) **endif**

DEFINITION:
itimes-tree-rec (l)
= **if** cdr $(l) \simeq$ **nil** **then** car (l)
else list (`'itimes`, car (l), itimes-tree-rec (cdr (l))) **endif**

DEFINITION:
itimes-tree (l)
= **if** listp (l)
then if listp (cdr (l)) **then** itimes-tree-rec (l)
else list (`'fix-int`, car (l)) **endif**
else `''1` **endif**

DEFINITION:
itimes-list (x)
= **if** listp (x) **then** itimes (car (x), itimes-list (cdr (x)))
else `1` **endif**

THEOREM: integerp-itimes-list
integerp (itimes-list (x))

THEOREM: eval$-itimes-tree-rec
listp (x)
→ (eval$ (**t**, itimes-tree-rec (x), a)
= **if** listp (cdr (x)) **then** itimes-list (eval$ (`'list`, x, a))
else eval$ (**t**, car (x), a) **endif**)

```
;; The following allows us to pretty much ignore itimes-tree forever.  (Notice
;; that it is disabled immediately below.)
```

THEOREM: eval$-itimes-tree
eval$ (**t**, itimes-tree (*x*), *a*) = itimes-list (eval$ (`'list`, *x*, *a*))

EVENT: Disable itimes-tree.

```
;; because we want to use EVAL$-ITIMES-TREE for now
```

DEFINITION:
make-cancel-itimes-equality (*x*, *y*, *in-both*)
= list (`'if`,
list (`'equal`, itimes-tree (*in-both*), `''0`),
list (`'quote`, **t**),
list (`'equal`,
itimes-tree (bagdiff (*x*, *in-both*)),
itimes-tree (bagdiff (*y*, *in-both*))))

```
#| The function below has no AND or OR, for efficiency
(defn cancel-itimes (x)
  (if (and (listp x)
   (equal (car x) (quote equal)))
      (if (and (listp (cadr x))
       (equal (caadr x) (quote itimes))
       (listp (caddr x))
       (equal (caaddr x) (quote itimes)))
  (if (listp (bagint (itimes-fringe (cadr x)) (itimes-fringe (caddr x))))
      (make-cancel-itimes-equality (itimes-fringe (cadr x))
   (itimes-fringe (caddr x))
   (bagint (itimes-fringe (cadr x)) (itimes-fringe (caddr x))))
    x)
(if (and (listp (cadr x))
 (equal (caadr x) (quote itimes)))
    ;; We don't want to introduce the IF below unless something
    ;; is "gained", or else we may get into an infinite rewriting loop.
    (if (member (caddr x) (itimes-fringe (cadr x)))
(list (quote if)
      (list (quote integerp) (caddr x))
      (make-cancel-itimes-equality (itimes-fringe (cadr x))
   (list (caddr x))
   (list (caddr x)))
      (list (quote quote) f))
      x)
  (if (and (listp (caddr x))
   (equal (caaddr x) (quote itimes)))
```

```
      (if (member (cadr x) (itimes-fringe (caddr x)))
  (list (quote if)
(list (quote integerp) (cadr x))
(make-cancel-itimes-equality (list (cadr x))
     (itimes-fringe (caddr x))
     (list (cadr x)))
(list (quote quote) f))
x)
    x)))
    x))
|#
```

DEFINITION:
cancel-itimes (x)
= **if** listp (x)
then if car (x) = `'equal`
then if listp (cadr (x))
then if caadr (x) = `'itimes`
then if listp (caddr (x))
then if caaddr (x) = `'itimes`
then if listp (bagint (itimes-fringe (cadr (x)),
itimes-fringe (caddr (x))))
then make-cancel-itimes-equality (itimes-fringe (cadr (x)),
itimes-fringe (caddr (x)),
bagint (itimes-fringe (cadr (x))),
itimes-fringe (caddr (x))))
else x **endif**
elseif caddr (x) ∈ itimes-fringe (cadr (x))
then list (`'if`,
list (`'integerp`, caddr (x)),
make-cancel-itimes-equality (itimes-fringe (cadr (x)),
list (caddr (x)),
list (caddr (x))),
list (`'quote`, **f**))
else x **endif**
elseif caddr (x) ∈ itimes-fringe (cadr (x))
then list (`'if`,
list (`'integerp`, caddr (x)),
make-cancel-itimes-equality (itimes-fringe (cadr (x)),
list (caddr (x)),
list (caddr (x))),
list (`'quote`, **f**))
else x **endif**

```
                    elseif listp (caddr (x))
                    then if caaddr (x) = 'itimes
                         then if cadr (x) ∈ itimes-fringe (caddr (x))
                              then list ('if,
                                         list ('integerp, cadr (x)),
                                         make-cancel-itimes-equality (list (cadr (x)),
                                                                      itimes-fringe (caddr (x)),
                                                                      list (cadr (x))),
                                         list ('quote, f))
                              else x endif
                         else x endif
                    elseif listp (caddr (x))
                    then if caaddr (x) = 'itimes
                         then if cadr (x) ∈ itimes-fringe (caddr (x))
                              then list ('if,
                                         list ('integerp, cadr (x)),
                                         make-cancel-itimes-equality (list (cadr (x)),
                                                                      itimes-fringe (caddr (x)),
                                                                      list (cadr (x))),
                                         list ('quote, f))
                              else x endif
                         else x endif
                    else x endif
               else x endif
          else x endif
```

THEOREM: itimes-list-append
itimes-list (append (x, y)) = itimes (itimes-list (x), itimes-list (y))

THEOREM: itimes-list-eval$-fringe
itimes-list (eval$ ('list, itimes-fringe (x), a)) = fix-int (eval$ (t, x, a))

THEOREM: integerp-eval$-itimes
(car (x) = 'itimes) → integerp (eval$ (t, x, a))

THEOREM: not-integerp-implies-not-equal-itimes
(¬ integerp (a)) → ((a = itimes (b, c)) = f)

THEOREM: itimes-list-eval$-delete
(z ∈ y)
→ (itimes-list (eval$ ('list, y, a))
= itimes (eval$ (t, z, a), itimes-list (eval$ ('list, delete (z, y), a))))

THEOREM: itimes-list-bagdiff

subbagp (x, y)
→ (itimes-list (eval$ ('list, y, a))
= itimes (itimes-list (eval$ ('list, bagdiff (y, x), a)),
itimes-list (eval$ ('list, x, a))))

THEOREM: equal-itimes-list-eval$-list-delete
(($c \in y$) ∧ (fix-int (eval$ (**t**, c, a)) ≠ 0))
→ ((x = itimes-list (eval$ ('list, delete (c, y), a)))
= (integerp (x)
∧ (itimes (x, eval$ (**t**, c, a))
= itimes-list (eval$ ('list, y, a)))))

EVENT: Disable itimes-list-eval$-delete.

```
;; I had trouble with the clausifier (thanks, J, for pointing that out
;; as the source of my trouble) in the proof of the meta lemma -- it's
;; getting rid of a case split.  So, I'll proceed by reducing
;; cancel-itimes in each case; see lemma eval$-make-cancel-itimes-equality
;; (and its -1 and -2 versions).
```

THEOREM: member-append
($a \in$ append (x, y)) = (($a \in x$) ∨ ($a \in y$))

THEOREM: member-izerop-itimes-fringe
(($z \in$ itimes-fringe (x)) ∧ (fix-int (eval$ (**t**, z, a)) = 0))
→ (fix-int (eval$ (**t**, x, a)) = 0)

THEOREM: correctness-of-cancel-itimes-hack-1
(($w \in$ itimes-fringe (cons ('itimes, *x1*)))
∧ (fix-int (eval$ (**t**, w, a)) = 0)
∧ (fix-int (eval$ (**t**, car (*x1*), a)) ≠ 0))
→ (fix-int (eval$ (**t**, cadr (*x1*), a)) = 0)

EVENT: Enable eval$-equal.

THEOREM: eval$-make-cancel-itimes-equality
eval$ (**t**, make-cancel-itimes-equality (x, y, *in-both*), a)
= **if** eval$ (**t**, list ('equal, itimes-tree (*in-both*), ''0), a) **then t**
else itimes-list (eval$ ('list, bagdiff (x, *in-both*), a))
= itimes-list (eval$ ('list, bagdiff (y, *in-both*), a)) **endif**

EVENT: Disable make-cancel-itimes-equality.

```
;; Here's a special case that I hope helps with the clausifier problem.
;; The lemma above seems necessary for its proof.
```

THEOREM: eval$-make-cancel-itimes-equality-1
eval$ (**t**, make-cancel-itimes-equality (list (x), y, list (x)), a)
= **if** fix-int (eval$ (**t**, x, a)) = `0` **then t**
else `1` = itimes-list (eval$ (`'list`, delete (x, y), a)) **endif**

THEOREM: equal-fix-int
(fix-int (x) = x) = integerp (x)

```
;; Here's another special case that I hope helps with the clausifier problem.
```

THEOREM: eval$-make-cancel-itimes-equality-2
eval$ (**t**, make-cancel-itimes-equality (x, list (y), list (y)), a)
= **if** fix-int (eval$ (**t**, y, a)) = `0` **then t**
else `1` = itimes-list (eval$ (`'list`, delete (y, x), a)) **endif**

THEOREM: eval$-equal-itimes-tree-itimes-fringe-0
(eval$ (**t**, list (`'equal`, itimes-tree (itimes-fringe (x)), `''0`), a)
∧ (car (x) = `'itimes`))
→ (eval$ (**t**, x, a) = `0`)

THEOREM: izerop-eval-of-member-implies-itimes-list-0
(($z \in y$) ∧ (fix-int (eval$ (**t**, z, a)) = `0`))
→ (itimes-list (eval$ (`'list`, y, a)) = `0`)

```
#| The function below has no AND or OR, for efficiency
(defn subsetp (x y)
  (if (nlistp x)
      t
    (and (member (car x) y)
 (subsetp (cdr x) y))))
|#
```

DEFINITION:
subsetp (x, y)
= **if** $x \simeq$ **nil then t**
elseif car (x) $\in$ y **then** subsetp (cdr (x), y)
else f endif

THEOREM: subsetp-implies-itimes-list-eval$-equals-0
(subsetp (x, y) ∧ (itimes-list (eval$ (`'list`, x, a)) = `0`))
→ (itimes-list (eval$ (`'list`, y, a)) = `0`)

THEOREM: subbagp-subsetp
subbagp $(x,\ y) \rightarrow$ subsetp $(x,\ y)$

THEOREM: equal-0-itimes-list-eval$-bagint-1
(itimes-list (eval$ ('list, bagint $(x,\ y)$, a)) = 0)
$\rightarrow$ (itimes-list (eval$ ('list, x, a)) = 0)

THEOREM: equal-0-itimes-list-eval$-bagint-2
(itimes-list (eval$ ('list, bagint $(x,\ y)$, a)) = 0)
$\rightarrow$ (itimes-list (eval$ ('list, y, a)) = 0)

THEOREM: correctness-of-cancel-itimes-hack-2
(listp (u)
$\wedge$ (car (u) = 'itimes)
$\wedge$ listp (v)
$\wedge$ (car (v) = 'itimes)
$\wedge$ (eval$ (**t**, u, a) $\neq$ eval$ (**t**, v, a)))
$\rightarrow$ (itimes-list (eval$ ('list,
bagdiff (itimes-fringe (u),
bagint (itimes-fringe (u), itimes-fringe (v))),
a))
$\neq$ itimes-list (eval$ ('list,
bagdiff (itimes-fringe (v),
bagint (itimes-fringe (u),
itimes-fringe (v))),
a)))

THEOREM: correctness-of-cancel-itimes-hack-3-lemma
((u = itimes $(a,\ b)$) $\wedge$ (fix-int (a) $\neq$ 0))
$\rightarrow$ ((u = itimes $(a,\ c)$) = (fix-int (b) = fix-int (c)))

THEOREM: correctness-of-cancel-itimes-hack-3
(listp (u)
$\wedge$ (car (u) = 'itimes)
$\wedge$ listp (v)
$\wedge$ (car (v) = 'itimes)
$\wedge$ (eval$ (**t**, u, a) = eval$ (**t**, v, a))
$\wedge$ ($\neg$ eval$ (**t**,
list ('equal,
itimes-tree (bagint (itimes-fringe (u), itimes-fringe (v))),
''0),
a)))
$\rightarrow$ ((itimes-list (eval$ ('list,
bagdiff (itimes-fringe (u),
bagint (itimes-fringe (u), itimes-fringe (v))),

a))
= itimes-list (eval$ ('list,
bagdiff (itimes-fringe (v),
bagint (itimes-fringe (u),
itimes-fringe (v))),
a)))
= **t**)

EVENT: Disable correctness-of-cancel-itimes-hack-3-lemma.

THEOREM: correctness-of-cancel-itimes
eval$ (**t**, x, a) = eval$ (**t**, cancel-itimes (x), a)

```
; ---------- Meta lemma for itimes cancellation on ilessp terms

;; I'll try to keep this similar to the approach for equalities above,
;; modified as in the iplus case (i.e. no fix-int is necessary).

;; EVAL$-EQUAL is currently enabled, but that's OK.
```

DEFINITION:
itimes-tree-no-fix-int (l)
= **if** listp (l) **then** itimes-tree-rec (l)
else ''1 **endif**

```
;; The following allows us to pretty much ignore
;; itimes-tree-no-fix-int forever.  (Notice that it is disabled
;; immediately below.)
```

THEOREM: eval$-itimes-tree-no-fix-int-1
ilessp (eval$ (**t**, itimes-tree-no-fix-int (x), a), y)
= ilessp (eval$ (**t**, itimes-tree (x), a), y)

THEOREM: eval$-itimes-tree-no-fix-int-2
ilessp (y, eval$ (**t**, itimes-tree-no-fix-int (x), a))
= ilessp (y, eval$ (**t**, itimes-tree (x), a))

EVENT: Disable itimes-tree-no-fix-int.

```
;; We want to use EVAL$-ITIMES-TREE, and ITIMES-TREE is still disabled
;; so we're in good shape.
```

DEFINITION:
make-cancel-itimes-inequality (x, y, *in-both*)
= list (`'if`,
list (`'ilessp`, itimes-tree-no-fix-int (*in-both*), `''0`),
list (`'ilessp`,
itimes-tree-no-fix-int (bagdiff (y, *in-both*)),
itimes-tree-no-fix-int (bagdiff (x, *in-both*))),
list (`'if`,
list (`'ilessp`, `''0`, itimes-tree-no-fix-int (*in-both*)),
list (`'ilessp`,
itimes-tree-no-fix-int (bagdiff (x, *in-both*)),
itimes-tree-no-fix-int (bagdiff (y, *in-both*))),
`'(false)`))

```
#| The function below has no AND or OR, for efficiency
(defn cancel-itimes-ilessp (x)
  (if (and (listp x)
   (equal (car x) (quote ilessp))
   (listp (bagint (itimes-fringe (cadr x)) (itimes-fringe (caddr x)))))
      (make-cancel-itimes-inequality (itimes-fringe (cadr x))
     (itimes-fringe (caddr x))
     (bagint (itimes-fringe (cadr x))
     (itimes-fringe (caddr x))))
    x))
|#
```

DEFINITION:
cancel-itimes-ilessp (x)
= **if** listp (x)
then if car (x) = `'ilessp`
then if listp (bagint (itimes-fringe (cadr (x)),
itimes-fringe (caddr (x))))
then make-cancel-itimes-inequality (itimes-fringe (cadr (x)),
itimes-fringe (caddr (x)),
bagint (itimes-fringe (cadr (x)),
itimes-fringe (caddr (x))))
else x **endif**
else x **endif**
else x **endif**

THEOREM: eval$-make-cancel-itimes-inequality
eval$ (**t**, make-cancel-itimes-inequality (x, y, *in-both*), a)
= **if** eval$ (**t**, list (`'ilessp`, itimes-tree-no-fix-int (*in-both*), `''0`), a)
then ilessp (eval$ (**t**, itimes-tree-no-fix-int (bagdiff (y, *in-both*)), a),

eval$ (**t**, itimes-tree-no-fix-int (bagdiff (*x*, *in-both*)), *a*))
elseif eval$ (**t**, list (`'ilessp`, `''0`, itimes-tree-no-fix-int (*in-both*)), *a*)
then ilessp (eval$ (**t**, itimes-tree-no-fix-int (bagdiff (*x*, *in-both*)), *a*),
eval$ (**t**, itimes-tree-no-fix-int (bagdiff (*y*, *in-both*)), *a*))
else f endif

EVENT: Disable make-cancel-itimes-inequality.

THEOREM: listp-bagint-with-singleton-implies-member
listp (bagint (*y*, list (*z*))) → (*z* ∈ *y*)

THEOREM: itimes-list-eval$-list-0
(`0` ∈ *x*) → (itimes-list (eval$ (`'list`, *x*, *a*)) = `0`)

THEOREM: ilessp-itimes-right-positive
ilessp (`0`, *x*) → (ilessp (*y*, *z*) = ilessp (itimes (*y*, *x*), itimes (*z*, *x*)))

THEOREM: correctness-of-cancel-itimes-ilessp-hack-1
(subbagp (*bag*, *x*)
∧ subbagp (*bag*, *y*)
∧ ilessp (`0`, itimes-list (eval$ (`'list`, *bag*, *a*))))
→ (ilessp (itimes-list (eval$ (`'list`, bagdiff (*x*, *bag*), *a*)),
itimes-list (eval$ (`'list`, bagdiff (*y*, *bag*), *a*)))
= ilessp (itimes-list (eval$ (`'list`, *x*, *a*)),
itimes-list (eval$ (`'list`, *y*, *a*))))

THEOREM: listp-bagint-with-singleton-member
listp (bagint (*y*, list (*z*))) = (*z* ∈ *y*)

THEOREM: correctness-of-cancel-itimes-ilessp-hack-2-lemma
(`0` ∈ itimes-fringe (*w*)) → (eval$ (**t**, *w*, *a*) = `0`)

THEOREM: correctness-of-cancel-itimes-ilessp-hack-2
(`0` ∈ itimes-fringe (*w*)) → (¬ ilessp (eval$ (**t**, *w*, *a*), `0`))

EVENT: Disable correctness-of-cancel-itimes-ilessp-hack-2-lemma.

```
;;; Now hack-3 and hack-4 below are all that's left to prove before the
;;; main result.
```

THEOREM: ilessp-trichotomy
(¬ ilessp (*x*, *y*)) → (ilessp (*y*, *x*) = (fix-int (*x*) ≠ fix-int (*y*)))

THEOREM: correctness-of-cancel-itimes-ilessp-hack-3-lemma-1
((0 = itimes-list (eval$ ('list, *bag*, *a*))) ∧ subsetp (*bag*, *z*))
→ (itimes-list (eval$ ('list, *z*, *a*)) = 0)

THEOREM: correctness-of-cancel-itimes-ilessp-hack-3-lemma-2
((0 = itimes-list (eval$ ('list, *bag*, *a*))) ∧ subsetp (*bag*, itimes-fringe (*x*)))
→ (fix-int (eval$ (**t**, *x*, *a*)) = 0)

THEOREM: same-fix-int-implies-not-ilessp
(fix-int (*x*) = fix-int (*y*)) → (¬ ilessp (*x*, *y*))

THEOREM: correctness-of-cancel-itimes-ilessp-hack-3
((¬ ilessp (itimes-list (eval$ ('list, *bag*, *a*)), 0))
∧ (¬ ilessp (0, itimes-list (eval$ ('list, *bag*, *a*))))
∧ subbagp (*bag*, itimes-fringe (*w*))
∧ subbagp (*bag*, itimes-fringe (*v*)))
→ (¬ ilessp (eval$ (**t**, *w*, *a*), eval$ (**t**, *v*, *a*)))

THEOREM: ilessp-itimes-right-negative
ilessp (*x*, 0) → (ilessp (*y*, *z*) = ilessp (itimes (*z*, *x*), itimes (*y*, *x*)))

THEOREM: correctness-of-cancel-itimes-ilessp-hack-4
(subbagp (*bag*, *x*)
∧ subbagp (*bag*, *y*)
∧ ilessp (itimes-list (eval$ ('list, *bag*, *a*)), 0))
→ (ilessp (itimes-list (eval$ ('list, bagdiff (*x*, *bag*), *a*)),
itimes-list (eval$ ('list, bagdiff (*y*, *bag*), *a*)))
= ilessp (itimes-list (eval$ ('list, *y*, *a*)),
itimes-list (eval$ ('list, *x*, *a*))))

EVENT: Disable ilessp-trichotomy.

EVENT: Disable same-fix-int-implies-not-ilessp.

THEOREM: correctness-of-cancel-itimes-ilessp
eval$ (**t**, *x*, *a*) = eval$ (**t**, cancel-itimes-ilessp (*x*), *a*)

```
;; I think that the following lemma is safe because it won't be
;; called at all during relieve-hyps.
```

THEOREM: ilessp-strict
ilessp (*x*, *y*) → (¬ ilessp (*y*, *x*))

```
; ---------- Setting up the State ----------

;; I'll close by disabling (or enabling) those rules and definitions
;; whose status as left over from above isn't quite what I'd like.
;; I'm going to leave the eval$ rules on and eval$ off.
```

EVENT: Disable eval$-cancel-iplus.

EVENT: Disable eval$-iplus.

EVENT: Disable lessp-count-listp-cdr.

EVENT: Disable eval$-iplus-tree-rec.

EVENT: Disable eval$-iplus-tree.

```
;;(disable eval$-list-append) ;; Nice rule -- I'll keep it enabled
```

EVENT: Disable iplus-list-eval$-fringe.

EVENT: Disable eval$-iplus-list-bagdiff.

EVENT: Disable lessp-difference-plus-arg1.

EVENT: Disable lessp-difference-plus-arg1-commuted.

EVENT: Disable correctness-of-cancel-iplus-ilessp-lemma.

EVENT: Disable eval$-ilessp-iplus-tree-no-fix-int.

EVENT: Disable make-cancel-iplus-inequality-simplifier.

EVENT: Disable quotient-difference-lessp-arg2.

EVENT: Disable eval$-itimes-tree-rec.

EVENT: Disable eval$-itimes-tree.

EVENT: Disable itimes-list-eval$-fringe.

EVENT: Disable integerp-eval$-itimes.

EVENT: Disable itimes-list-bagdiff.

EVENT: Disable equal-itimes-list-eval$-list-delete.

EVENT: Disable member-izerop-itimes-fringe.

EVENT: Disable correctness-of-cancel-itimes-hack-1.

EVENT: Disable eval$-make-cancel-itimes-equality.

EVENT: Disable eval$-make-cancel-itimes-equality-1.

EVENT: Disable eval$-make-cancel-itimes-equality-2.

EVENT: Disable eval$-equal-itimes-tree-itimes-fringe-0.

EVENT: Disable izerop-eval-of-member-implies-itimes-list-0.

EVENT: Disable subsetp-implies-itimes-list-eval$-equals-0.

EVENT: Disable equal-0-itimes-list-eval$-bagint-1.

EVENT: Disable equal-0-itimes-list-eval$-bagint-2.

EVENT: Disable correctness-of-cancel-itimes-hack-2.

EVENT: Disable correctness-of-cancel-itimes-hack-3-lemma.

EVENT: Disable correctness-of-cancel-itimes-hack-3.

EVENT: Disable eval$-itimes-tree-no-fix-int-1.

EVENT: Disable eval$-itimes-tree-no-fix-int-2.

EVENT: Disable eval$-make-cancel-itimes-inequality.

EVENT: Disable listp-bagint-with-singleton-implies-member.

EVENT: Disable itimes-list-eval$-list-0.

EVENT: Disable correctness-of-cancel-itimes-ilessp-hack-1.

EVENT: Disable listp-bagint-with-singleton-member.

EVENT: Disable correctness-of-cancel-itimes-ilessp-hack-2.

EVENT: Disable correctness-of-cancel-itimes-ilessp-hack-3-lemma-1.

EVENT: Disable correctness-of-cancel-itimes-ilessp-hack-3-lemma-2.

EVENT: Disable correctness-of-cancel-itimes-ilessp-hack-3.

EVENT: Disable correctness-of-cancel-itimes-ilessp-hack-4.

```
;; The last one is a tough call, but I think it's OK.
;; (disable ilessp-strict)

;;;;;; ***** EXTRA META STUFF ***** ;;;;;;

;; The next goal is to improve itimes cancellation so that it looks
;; for common factors, and hence works on equations like
;;    x*y + x = x*z
;; and, for that matter,
;; a*x + -b*x = 0.

;; Rather than changing the existing cancel-itimes function, I'll
;; leave that one but disable its metalemma at the end.  Then if the
;; new version, which I'll call cancel-itimes-factors, is found to be
;; too slow, one can always disable its metalemma and re-enable the
;; metalemma for cancel-itimes.
```

```
;; Notice, by the way, that the existing cancel-itimes function is
;; useless for something like the following, since there's no special
;; treatment for INEG.  I'll remedy that in this version.

#|
(IMPLIES (AND (NOT (IZEROP X))
      (EQUAL (ITIMES A X) (INEG (ITIMES B X))))
 (EQUAL (FIX-INT A) (INEG B)))
|#
```

DEFINITION:
itimes-tree-ineg (l)
= **if** listp (l)
then if car (l) = list (`'quote`, `-1`)
then if listp (cdr (l)) **then** list (`'ineg`, itimes-tree-rec (cdr (l)))
else car (l) **endif**
else `''1` **endif**
else `''1` **endif**

DEFINITION:
itimes-factors (x)
= **if** listp (x)
then if car (x) = `'itimes`
then append (itimes-factors (cadr (x)), itimes-factors (caddr (x)))
elseif car (x) = `'iplus`
then let *bag1* **be** itimes-factors (cadr (x)),
bag2 **be** itimes-factors (caddr (x))
in
let *inboth* **be** bagint (*bag1*, *bag2*)
in
if listp (*inboth*)
then cons (list (`'iplus`,
itimes-tree-ineg (bagdiff (*bag1*,
inboth)),
itimes-tree-ineg (bagdiff (*bag2*,
inboth))),
inboth)
else list (x) **endif endlet endlet**
elseif car (x) = `'ineg`
then cons (list (`'quote`, `-1`), itimes-factors (cadr (x)))
else list (x) **endif**
else list (x) **endif**

THEOREM: itimes–1
itimes (-1, x) = ineg (x)

```
;; I'll need the following lemma because it's simplest not to deal with
;; e.g. (equal x x), where x is a variable, in the meta thing.  I'll do
;; the one after it too, simply because I'm thinking of it now.
```

THEOREM: equal-ineg-ineg
(ineg (x) = ineg (y)) = (fix-int (x) = fix-int (y))

THEOREM: ilessp-ineg-ineg
ilessp (ineg (x), ineg (y)) = ilessp (y, x)

THEOREM: fix-int-eval$-itimes-tree-rec
listp (x)
→ (fix-int (eval$ (**t**, itimes-tree-rec (x), a))
= itimes-list (eval$ ('list, x, a)))

THEOREM: eval$-itimes-tree-ineg
fix-int (eval$ (**t**, itimes-tree-ineg (x), a)) = itimes-list (eval$ ('list, x, a))

```
;; Now I want the above lemma to apply, but it doesn't, so the
;; following three lemmas are used instead.
```

THEOREM: ineg-eval$-itimes-tree-ineg
ineg (eval$ (**t**, itimes-tree-ineg (x), a))
= ineg (itimes-list (eval$ ('list, x, a)))

THEOREM: iplus-eval$-itimes-tree-ineg
(iplus (eval$ (**t**, itimes-tree-ineg (x), a), y)
= iplus (itimes-list (eval$ ('list, x, a)), y))
∧ (iplus (y, eval$ (**t**, itimes-tree-ineg (x), a))
= iplus (y, itimes-list (eval$ ('list, x, a))))

THEOREM: itimes-eval$-itimes-tree-ineg
(itimes (eval$ (**t**, itimes-tree-ineg (x), a), y)
= itimes (itimes-list (eval$ ('list, x, a)), y))
∧ (itimes (y, eval$ (**t**, itimes-tree-ineg (x), a))
= itimes (y, itimes-list (eval$ ('list, x, a))))

EVENT: Disable itimes-tree-ineg.

```
#| ****** The following definitions are for efficient execution of
   metafunctions.  They should probably be applied to all the metafunctions
```

```
   with fns arguments AND and OR.

(defmacro nqthm-macroexpand (defn &rest fns)
  `(nqthm-macroexpand-fn ',defn ',fns))

(defun nqthm-macroexpand-fn (defn fns)
  (iterate for fn in fns
   when (not (get fn 'sdefn))
   do (er soft (fn) |Sorry| |,| |but| |there| |is| |no| SDEFN
  |for| (!ppr fn (quote |.|))))
  (let (name args body)
    (match! defn (defn name args body))
    (let ((arity-alist (cons (cons name (length args)) arity-alist)))
      (list 'defn name args
    (untranslate (normalize-ifs
  (nqthm-macroexpand-term (translate body) fns)
  nil nil nil))))))

(defun nqthm-macroexpand-term (term fns)
  (cond
   ((or (variablep term) (fquotep term))
    term)
   ((member-eq (ffn-symb term) fns)
    (let ((sdefn (get (ffn-symb term) 'sdefn)))
      (sub-pair-var (cadr sdefn)
    (iterate for arg in (fargs term)
     collect (nqthm-macroexpand-term arg fns))
    (caddr sdefn))))
   (t (fcons-term (ffn-symb term)
  (iterate for arg in (fargs term)
   collect (nqthm-macroexpand-term arg fns))))))

|#

;; I "macroexpand" away the following below, so it's not really needed except
;; for the proof.  That is, I use it in the definition of cancel-itimes-factors,
;; but then get rid of it for cancel-itimes-factors-expanded, and although I
;; reason about the former, I USE the latter, for efficiency.
```

DEFINITION:
iplus-or-itimes-term (x)
= **if** listp (x)
then case on car (x):
case = *iplus*

```
        then t
        case = itimes
         then t
        case = ineg
         then if listp (cadr (x)) then car (cadr (x)) = 'itimes
                 else f endif
        otherwise f endcase
    else f endif
```

DEFINITION:

```
cancel-itimes-factors (x)
=   if listp (x) ∧ (car (x) = 'equal)
    then let bagint be bagint (itimes-factors (cadr (x)),
                               itimes-factors (caddr (x)))
         in
         let new-equality be make-cancel-itimes-equality (itimes-factors (cadr (x)),
                                                          itimes-factors (caddr (x)),
                                                          bagint)
         in
         if iplus-or-itimes-term (cadr (x))
         then if listp (bagint)
                 then if iplus-or-itimes-term (caddr (x))
                         then new-equality
                         else list ('if,
                                    list ('integerp,
                                          caddr (x)),
                                    new-equality,
                                    list ('quote, f)) endif
                 else x endif
         elseif iplus-or-itimes-term (caddr (x))
         then if listp (bagint)
                 then list ('if,
                            list ('integerp, cadr (x)),
                            new-equality,
                            list ('quote, f))
                 else x endif
         else x endif endlet endlet
    else x endif
```

```
;; The following was generated with the nqthm-macroexpand macro defined above.



;; The following was generated with the nqthm-macroexpand macro defined above.
(DEFN CANCEL-ITIMES-FACTORS-expanded (X)
```

```
  (IF (LISTP X)
      (IF (EQUAL (CAR X) 'EQUAL)
  (COND
   ((LISTP (CADR X))
    (CASE (CAR (CAR (CDR X)))
  (IPLUS (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
     (IF (LISTP (CADDR X))
 (CASE (CAR (CAR (CDR (CDR X))))
       (IPLUS
(MAKE-CANCEL-ITIMES-EQUALITY
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))
 (BAGINT (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X)))))
       (ITIMES
(MAKE-CANCEL-ITIMES-EQUALITY
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))
 (BAGINT (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X)))))
       (INEG
(IF (LISTP (CADADDR X))
    (IF (EQUAL (CAADADDR X) 'ITIMES)
(MAKE-CANCEL-ITIMES-EQUALITY
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))
 (BAGINT
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))))
(LIST 'IF
      (LIST 'INTEGERP (CADDR X))
      (MAKE-CANCEL-ITIMES-EQUALITY
       (ITIMES-FACTORS (CADR X))
       (ITIMES-FACTORS (CADDR X))
       (BAGINT
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))))
      (LIST 'QUOTE F)))
    (LIST 'IF
  (LIST 'INTEGERP (CADDR X))
  (MAKE-CANCEL-ITIMES-EQUALITY
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))
```

```
   (BAGINT
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
  (LIST 'QUOTE F))))
       (OTHERWISE
(LIST 'IF
      (LIST 'INTEGERP (CADDR X))
      (MAKE-CANCEL-ITIMES-EQUALITY
       (ITIMES-FACTORS (CADR X))
       (ITIMES-FACTORS (CADDR X))
       (BAGINT
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))))
      (LIST 'QUOTE F))))
 (LIST 'IF (LIST 'INTEGERP (CADDR X))
       (MAKE-CANCEL-ITIMES-EQUALITY
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))
(BAGINT
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))))
       (LIST 'QUOTE F)))
     X))
  (ITIMES (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))))
      (IF (LISTP (CADDR X))
  (CASE (CAR (CAR (CDR (CDR X))))
(IPLUS
 (MAKE-CANCEL-ITIMES-EQUALITY
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))
  (BAGINT
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X)))))
(ITIMES
 (MAKE-CANCEL-ITIMES-EQUALITY
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))
  (BAGINT
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X)))))
(INEG
 (IF (LISTP (CADADDR X))
     (IF (EQUAL (CAADADDR X) 'ITIMES)
```

```
 (MAKE-CANCEL-ITIMES-EQUALITY
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))
  (BAGINT
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))))
 (LIST 'IF
       (LIST 'INTEGERP (CADDR X))
       (MAKE-CANCEL-ITIMES-EQUALITY
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))
(BAGINT
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))))
       (LIST 'QUOTE F)))
     (LIST 'IF
   (LIST 'INTEGERP (CADDR X))
   (MAKE-CANCEL-ITIMES-EQUALITY
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))
    (BAGINT
     (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))))
   (LIST 'QUOTE F))))
(OTHERWISE
 (LIST 'IF
       (LIST 'INTEGERP (CADDR X))
       (MAKE-CANCEL-ITIMES-EQUALITY
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))
(BAGINT
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))))
       (LIST 'QUOTE F))))
  (LIST 'IF (LIST 'INTEGERP (CADDR X))
(MAKE-CANCEL-ITIMES-EQUALITY
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))
 (BAGINT (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))))
(LIST 'QUOTE F)))
      X))
  (INEG (COND
 ((LISTP (CADADR X))
```

```
  (COND
   ((EQUAL (CAADADR X) 'ITIMES)
    (IF (LISTP
 (BAGINT (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))))
(IF (LISTP (CADDR X))
                                    (CASE (CAR (CAR (CDR (CDR X))))
  (IPLUS
   (MAKE-CANCEL-ITIMES-EQUALITY
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))
    (BAGINT
     (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X)))))
  (ITIMES
   (MAKE-CANCEL-ITIMES-EQUALITY
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))
    (BAGINT
     (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X)))))
  (INEG
   (IF (LISTP (CADADDR X))
       (IF
(EQUAL (CAADADDR X) 'ITIMES)
(MAKE-CANCEL-ITIMES-EQUALITY
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))
 (BAGINT
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))))
(LIST 'IF
      (LIST 'INTEGERP (CADDR X))
      (MAKE-CANCEL-ITIMES-EQUALITY
       (ITIMES-FACTORS (CADR X))
       (ITIMES-FACTORS (CADDR X))
       (BAGINT
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))))
      (LIST 'QUOTE F)))
       (LIST 'IF
     (LIST 'INTEGERP (CADDR X))
     (MAKE-CANCEL-ITIMES-EQUALITY
      (ITIMES-FACTORS (CADR X))
```

```
      (ITIMES-FACTORS (CADDR X))
      (BAGINT
       (ITIMES-FACTORS (CADR X))
       (ITIMES-FACTORS (CADDR X))))
     (LIST 'QUOTE F))))
  (OTHERWISE
   (LIST 'IF
 (LIST 'INTEGERP (CADDR X))
 (MAKE-CANCEL-ITIMES-EQUALITY
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))
  (BAGINT
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))))
 (LIST 'QUOTE F))))
                                    (LIST 'IF
  (LIST 'INTEGERP (CADDR X))
  (MAKE-CANCEL-ITIMES-EQUALITY
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))
   (BAGINT (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))))
  (LIST 'QUOTE F)))
X))
   ((LISTP (CADDR X))
    (CASE (CAR (CAR (CDR (CDR X))))
  (IPLUS (IF
  (LISTP
   (BAGINT
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
  (LIST 'IF
(LIST 'INTEGERP (CADR X))
(MAKE-CANCEL-ITIMES-EQUALITY
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))
 (BAGINT
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))))
(LIST 'QUOTE F))
  X))
  (ITIMES (IF
   (LISTP
    (BAGINT
```

```
     (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))))
   (LIST 'IF
 (LIST 'INTEGERP (CADR X))
 (MAKE-CANCEL-ITIMES-EQUALITY
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))
  (BAGINT
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))))
 (LIST 'QUOTE F))
   X))
  (INEG (IF (LISTP (CADADDR X))
    (IF
     (EQUAL (CAADADDR X) 'ITIMES)
     (IF
      (LISTP
       (BAGINT
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))))
      (LIST 'IF
    (LIST 'INTEGERP (CADR X))
    (MAKE-CANCEL-ITIMES-EQUALITY
     (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))
     (BAGINT
      (ITIMES-FACTORS (CADR X))
      (ITIMES-FACTORS (CADDR X))))
    (LIST 'QUOTE F))
      X)
     X)
    X))
  (OTHERWISE X)))
   (T X)))
 ((LISTP (CADDR X))
  (CASE (CAR (CAR (CDR (CDR X))))
(IPLUS (IF
(LISTP
 (BAGINT
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))))
(LIST 'IF
      (LIST 'INTEGERP (CADR X))
      (MAKE-CANCEL-ITIMES-EQUALITY
```

```
       (ITIMES-FACTORS (CADR X))
       (ITIMES-FACTORS (CADDR X))
       (BAGINT
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))))
      (LIST 'QUOTE F))
X))
(ITIMES (IF
 (LISTP
  (BAGINT
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))))
 (LIST 'IF
       (LIST 'INTEGERP (CADR X))
       (MAKE-CANCEL-ITIMES-EQUALITY
(ITIMES-FACTORS (CADR X))
(ITIMES-FACTORS (CADDR X))
(BAGINT
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))))
       (LIST 'QUOTE F))
 X))
(INEG (IF (LISTP (CADADDR X))
  (IF (EQUAL (CAADADDR X) 'ITIMES)
      (IF
       (LISTP
(BAGINT
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))))
       (LIST 'IF
     (LIST 'INTEGERP (CADR X))
     (MAKE-CANCEL-ITIMES-EQUALITY
      (ITIMES-FACTORS (CADR X))
      (ITIMES-FACTORS (CADDR X))
      (BAGINT
       (ITIMES-FACTORS (CADR X))
       (ITIMES-FACTORS (CADDR X))))
     (LIST 'QUOTE F))
       X)
      X)
  X))
(OTHERWISE X)))
 (T X)))
  (OTHERWISE
```

```
 (IF (LISTP (CADDR X))
     (CASE (CAR (CAR (CDR (CDR X))))
                           (IPLUS (IF
                                   (LISTP
                                    (BAGINT (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
                                   (LIST 'IF
 (LIST 'INTEGERP (CADR X))
 (MAKE-CANCEL-ITIMES-EQUALITY
  (ITIMES-FACTORS (CADR X))
  (ITIMES-FACTORS (CADDR X))
  (BAGINT
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))))
 (LIST 'QUOTE F))
                                  X))
                           (ITIMES (IF
                                    (LISTP
                                     (BAGINT
                                      (ITIMES-FACTORS (CADR X))
                                      (ITIMES-FACTORS (CADDR X))))
                                    (LIST 'IF
  (LIST 'INTEGERP (CADR X))
  (MAKE-CANCEL-ITIMES-EQUALITY
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))
   (BAGINT
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
  (LIST 'QUOTE F))
                                   X))
                           (INEG (IF (LISTP (CADADDR X))
     (IF (EQUAL (CAADADDR X) 'ITIMES)
 (IF
  (LISTP
   (BAGINT
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
  (LIST 'IF
(LIST 'INTEGERP (CADR X))
(MAKE-CANCEL-ITIMES-EQUALITY
 (ITIMES-FACTORS (CADR X))
 (ITIMES-FACTORS (CADDR X))
 (BAGINT
```

```
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
  (LIST 'QUOTE F))
    X)
   X)
       X))
                              (OTHERWISE X))
       X))))
   ((LISTP (CADDR X))
    (CASE (CAR (CAR (CDR (CDR X))))
  (IPLUS (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
     (LIST 'IF (LIST 'INTEGERP (CADR X))
   (MAKE-CANCEL-ITIMES-EQUALITY
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))
    (BAGINT (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
   (LIST 'QUOTE F))
     X))
  (ITIMES (IF (LISTP (BAGINT (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))))
      (LIST 'IF (LIST 'INTEGERP (CADR X))
    (MAKE-CANCEL-ITIMES-EQUALITY
     (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))
     (BAGINT (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))))
    (LIST 'QUOTE F))
      X))
  (INEG (IF (LISTP (CADADDR X))
    (IF (EQUAL (CAADADDR X) 'ITIMES)
(IF (LISTP
     (BAGINT (ITIMES-FACTORS (CADR X))
     (ITIMES-FACTORS (CADDR X))))
    (LIST 'IF
  (LIST 'INTEGERP (CADR X))
  (MAKE-CANCEL-ITIMES-EQUALITY
   (ITIMES-FACTORS (CADR X))
   (ITIMES-FACTORS (CADDR X))
   (BAGINT
    (ITIMES-FACTORS (CADR X))
    (ITIMES-FACTORS (CADDR X))))
  (LIST 'QUOTE F))
```

```
    X)
X)
    X))
  (OTHERWISE X)))
   (T X))
  X)
      X))
```

THEOREM: cancel-itimes-factors-expanded-cancel-itimes-factors
cancel-itimes-factors-expanded (x) = cancel-itimes-factors (x)

EVENT: Disable cancel-itimes-factors-expanded.

EVENT: Disable iplus-or-itimes-term.

THEOREM: equal-itimes-list-eval$-list-delete-new-1
(fix-int (eval$ (**t**, *elt*, *a*)) ≠ 0)
→ ((*x* = itimes-list (eval$ ('list, delete (*elt*, *bag*), *a*)))
= **if** *elt* ∈ *bag*
then integerp (*x*)
∧ (itimes (*x*, eval$ (**t**, *elt*, *a*))
= itimes-list (eval$ ('list, *bag*, *a*)))
else *x* = itimes-list (eval$ ('list, *bag*, *a*)) **endif**)

THEOREM: equal-itimes-list-eval$-list-delete-new-2
(fix-int (eval$ (**t**, *elt*, *a*)) ≠ 0)
→ ((itimes-list (eval$ ('list, delete (*elt*, *bag*), *a*)) = *x*)
= **if** *elt* ∈ *bag*
then integerp (*x*)
∧ (itimes (*x*, eval$ (**t**, *elt*, *a*))
= itimes-list (eval$ ('list, *bag*, *a*)))
else *x* = itimes-list (eval$ ('list, *bag*, *a*)) **endif**)

THEOREM: itimes-itimes-list-eval$-list-delete
(*x* ∈ *bag*)
→ (itimes (eval$ (**t**, *x*, *a*), itimes-list (eval$ ('list, delete (*x*, *bag*), *a*)))
= itimes-list (eval$ ('list, *bag*, *a*)))

THEOREM: equal-itimes-list-eval$-list-bagdiff
(subbagp (*in-both*, *bag1*)
∧ subbagp (*in-both*, *bag2*)
∧ (itimes-list (eval$ ('list, *in-both*, *a*)) ≠ 0))
→ ((itimes-list (eval$ ('list, bagdiff (*bag1*, *in-both*), *a*))
= itimes-list (eval$ ('list, bagdiff (*bag2*, *in-both*), *a*)))

```
    =   (itimes-list (eval$ ('list, bag1, a))
          =   itimes-list (eval$ ('list, bag2, a))))
```

THEOREM: membership-of-0-implies-itimes-list-is-0
$(0 \in x) \rightarrow (\text{itimes-list}\,(x) = 0)$

THEOREM: member-0-eval$-list
$(0 \in x) \rightarrow (0 \in \text{eval\$}\,(\text{'list}, x, a))$

THEOREM: itimes-list-eval$-factors-lemma
itimes (itimes-list (eval$ ('list, bagint (*bag1*, *bag2*), *a*)),
itimes-list (eval$ ('list, bagdiff (*bag2*, bagint (*bag1*, *bag2*)), *a*)))
= itimes-list (eval$ ('list, *bag2*, *a*))

THEOREM: itimes-list-eval$-factors-lemma-prime
itimes (itimes-list (eval$ ('list, bagint (*bag1*, *bag2*), *a*)),
itimes-list (eval$ ('list, bagdiff (*bag1*, bagint (*bag1*, *bag2*)), *a*)))
= itimes-list (eval$ ('list, *bag1*, *a*))

THEOREM: itimes-list-eval$-factors
itimes-list (eval$ ('list, itimes-factors (x), a)) = fix-int (eval$ (**t**, x, a))

THEOREM: iplus-or-itimes-term-integerp-eval$
iplus-or-itimes-term $(x) \rightarrow$ integerp (eval$ (**t**, x, a))

THEOREM: eval$-list-bagint-0
$(\text{itimes-list}\,(\text{eval\$}\,(\text{'list}, \text{bagint}\,(x, y), a)) = 0)$
$\rightarrow$ $((\text{itimes-list}\,(\text{eval\$}\,(\text{'list}, x, a)) = 0)$
$\wedge$ $(\text{itimes-list}\,(\text{eval\$}\,(\text{'list}, y, a)) = 0))$

THEOREM: eval$-list-bagint-0-implies-equal
((itimes-list (eval$ ('list, bagint (itimes-factors (v), itimes-factors (w))), a))
= 0)
$\wedge$ integerp (eval$ (**t**, v, a))
$\wedge$ integerp (eval$ (**t**, w, a)))
$\rightarrow$ $((\text{eval\$}\,(\mathbf{t}, v, a) = \text{eval\$}\,(\mathbf{t}, w, a)) = \mathbf{t})$

THEOREM: correctness-of-cancel-itimes-factors
eval$ (**t**, x, a) = eval$ (**t**, cancel-itimes-factors-expanded (x), a)

```
;; OK -- now, the lessp case, finally.  Ugh!
```

DEFINITION:
cancel-itimes-ilessp-factors (x)
= **if** listp (x)

then if car (*x*) = 'ilessp
then if listp (bagint (itimes-factors (cadr (*x*)),
itimes-factors (caddr (*x*))))
then make-cancel-itimes-inequality (itimes-factors (cadr (*x*)),
itimes-factors (caddr (*x*)),
bagint (itimes-factors (cadr (*x*)),
itimes-factors (caddr (*x*))))
else *x* **endif**
else *x* **endif**
else *x* **endif**

THEOREM: bagint-singleton
bagint (*x*, list (*y*))
= **if** *y* ∈ *x* **then** list (*y*)
else nil endif

THEOREM: izerop-ilessp-0-relationship
(fix-int (*x*) = 0) = ((¬ ilessp (*x*, 0)) ∧ (¬ ilessp (0, *x*)))

THEOREM: ilessp-itimes-list-eval$-list-delete-helper-1
ilessp (0, *w*) → (ilessp (itimes (*x*, *w*), itimes (*w*, *u*)) = ilessp (*x*, *u*))

THEOREM: ilessp-itimes-list-eval$-list-delete-helper-2
ilessp (*w*, 0) → (ilessp (itimes (*w*, *u*), itimes (*x*, *w*)) = ilessp (*x*, *u*))

THEOREM: ilessp-itimes-list-eval$-list-delete
((*z* ∈ *y*) ∧ (fix-int (eval$ (**t**, *z*, *a*)) ≠ 0))
→ (ilessp (*x*, itimes-list (eval$ ('list, delete (*z*, *y*), *a*)))
= **if** ilessp (0, eval$ (**t**, *z*, *a*))
then ilessp (itimes (*x*, eval$ (**t**, *z*, *a*)),
itimes-list (eval$ ('list, *y*, *a*)))
elseif ilessp (eval$ (**t**, *z*, *a*), 0)
then ilessp (itimes-list (eval$ ('list, *y*, *a*)),
itimes (*x*, eval$ (**t**, *z*, *a*)))
else f endif)

THEOREM: ilessp-itimes-list-eval$-list-delete-prime-helper-1
ilessp (0, *w*) → (ilessp (itimes (*w*, *u*), itimes (*x*, *w*)) = ilessp (*u*, *x*))

THEOREM: ilessp-itimes-list-eval$-list-delete-prime-helper-2
ilessp (*w*, 0) → (ilessp (itimes (*x*, *w*), itimes (*w*, *u*)) = ilessp (*u*, *x*))

THEOREM: ilessp-itimes-list-eval$-list-delete-prime
((*z* ∈ *y*) ∧ (fix-int (eval$ (**t**, *z*, *a*)) ≠ 0))
→ (ilessp (itimes-list (eval$ ('list, delete (*z*, *y*), *a*)), *x*)

```
      =  if ilessp(0, eval$(t, z, a))
         then ilessp(itimes-list(eval$('list, y, a)),
                     itimes(x, eval$(t, z, a)))
         elseif ilessp(eval$(t, z, a), 0)
         then ilessp(itimes(x, eval$(t, z, a)),
                     itimes-list(eval$('list, y, a)))
         else f endif)

;; **** Do I have anything like the following two lemmas for the equality case?
;; Should I?

;;;***** I should also consider if I've dealt with things like 0 = a*x + b*x, and
;;; simlilarly for ilessp.
```

THEOREM: ilessp-0-itimes
ilessp(0, itimes(*x*, *y*))
= ((ilessp(0, *x*) ∧ ilessp(0, *y*)) ∨ (ilessp(*x*, 0) ∧ ilessp(*y*, 0)))

THEOREM: ilessp-itimes-0
ilessp(itimes(*x*, *y*), 0)
= ((ilessp(0, *x*) ∧ ilessp(*y*, 0)) ∨ (ilessp(*x*, 0) ∧ ilessp(0, *y*)))

THEOREM: ilessp-itimes-list-eval$-list-bagdiff
(subbagp(*in-both*, *bag1*)
∧ subbagp(*in-both*, *bag2*)
∧ (itimes-list(eval$('list, *in-both*, *a*)) ≠ 0))
→ (ilessp(itimes-list(eval$('list, bagdiff(*bag1*, *in-both*), *a*)),
itimes-list(eval$('list, bagdiff(*bag2*, *in-both*), *a*)))
= **if** ilessp(0, itimes-list(eval$('list, *in-both*, *a*)))
then ilessp(itimes-list(eval$('list, *bag1*, *a*)),
itimes-list(eval$('list, *bag2*, *a*)))
else ilessp(itimes-list(eval$('list, *bag2*, *a*)),
itimes-list(eval$('list, *bag1*, *a*))) **endif**)

THEOREM: zero-ilessp-implies-not-equal
ilessp(0, *x*) → (0 ≠ *x*)

THEOREM: ilessp-itimes-list-eval$-list-bagdiff-corollary-1
(subbagp(*in-both*, *bag1*)
∧ subbagp(*in-both*, *bag2*)
∧ ilessp(0, itimes-list(eval$('list, *in-both*, *a*))))
→ (ilessp(itimes-list(eval$('list, bagdiff(*bag1*, *in-both*), *a*)),
itimes-list(eval$('list, bagdiff(*bag2*, *in-both*), *a*)))
= ilessp(itimes-list(eval$('list, *bag1*, *a*)),
itimes-list(eval$('list, *bag2*, *a*))))

THEOREM: ilessp-zero-implies-not-equal
ilessp(*x*, 0) → (0 ≠ *x*)

THEOREM: ilessp-itimes-list-eval$-list-bagdiff-corollary-2
(subbagp(*in-both*, *bag1*)
 ∧ subbagp(*in-both*, *bag2*)
 ∧ ilessp(itimes-list(eval$('list, *in-both*, *a*)), 0))
→ (ilessp(itimes-list(eval$('list, bagdiff(*bag1*, *in-both*), *a*)),
          itimes-list(eval$('list, bagdiff(*bag2*, *in-both*), *a*)))
    = ilessp(itimes-list(eval$('list, *bag2*, *a*)),
             itimes-list(eval$('list, *bag1*, *a*))))

THEOREM: member-0-itimes-factors-yields-0
(eval$(**t**, *w*, *a*) ≠ 0) → (0 ∉ itimes-factors(*w*))

THEOREM: member-0-itimes-factors-yields-0-ilessp-consequence-1
ilessp(eval$(**t**, *w*, *a*), 0) → (0 ∉ itimes-factors(*w*))

THEOREM: member-0-itimes-factors-yields-0-ilessp-consequence-2
ilessp(0, eval$(**t**, *w*, *a*)) → (0 ∉ itimes-factors(*w*))

```
#|
(prove-lemma eval$-list-bagint-0 nil
  (implies (equal (itimes-list (eval$ 'list (bagint x y) a)) 0)
   (and (equal (itimes-list (eval$ 'list x a)) 0)
(equal (itimes-list (eval$ 'list y a)) 0)))
  ((use (subsetp-implies-itimes-list-eval$-equals-0
 (x (bagint x y))
 (y x))
(subsetp-implies-itimes-list-eval$-equals-0
 (x (bagint x y))
 (y y)))))
|#

#|
(prove-lemma eval$-list-bagint-0-implies-equal (rewrite)
  (implies (and (equal (itimes-list (eval$ 'list (bagint (itimes-factors v) (itimes-factors
       0)
(integerp (eval$ t v a))
(integerp (eval$ t w a)))
   (equal (equal (eval$ t v a) (eval$ t w a))
  t))
  ((use (eval$-list-bagint-0 (x (itimes-factors v))
     (y (itimes-factors w))))))
|#
```

```
;; At this point I'm going to switch the states of ilessp-trichotomy and
;; izerop-ilessp-0-relationship, for good (or till I change my mind again!).
```

EVENT: Enable ilessp-trichotomy.

EVENT: Disable izerop-ilessp-0-relationship.

THEOREM: eval$-list-bagint-0-for-ilessp
$$\begin{array}{ll}
((\neg\ \text{ilessp}\,(\text{itimes-list}\,(\text{eval\$}\,(\texttt{'list},\ \text{bagint}\,(x,\ y),\ a)),\ \texttt{0})) & \\
\ \wedge\quad (\neg\ \text{ilessp}\,(\texttt{0},\ \text{itimes-list}\,(\text{eval\$}\,(\texttt{'list},\ \text{bagint}\,(x,\ y),\ a))))) & \\
\rightarrow\quad ((\text{fix-int}\,(\text{itimes-list}\,(\text{eval\$}\,(\texttt{'list},\ x,\ a))) = \texttt{0}) & \\
\qquad \wedge\quad (\text{fix-int}\,(\text{itimes-list}\,(\text{eval\$}\,(\texttt{'list},\ y,\ a))) = \texttt{0}))
\end{array}$$

THEOREM: eval$-list-bagint-0-implies-equal-for-ilessp-lemma
$$\begin{array}{ll}
((\neg\ \text{ilessp}\,(\text{itimes-list}\,(\text{eval\$}\,(\texttt{'list}, & \\
\qquad\qquad \text{bagint}\,(\text{itimes-factors}\,(v),\ \text{itimes-factors}\,(w)), & \\
\qquad\qquad a)), & \\
\qquad \texttt{0})) & \\
\ \wedge\quad (\neg\ \text{ilessp}\,(\texttt{0}, & \\
\qquad\qquad \text{itimes-list}\,(\text{eval\$}\,(\texttt{'list}, & \\
\qquad\qquad\qquad\qquad \text{bagint}\,(\text{itimes-factors}\,(v), & \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{itimes-factors}\,(w)), & \\
\qquad\qquad\qquad\qquad a))))) & \\
\rightarrow\quad (\text{fix-int}\,(\text{eval\$}\,(\mathbf{t},\ v,\ a)) = \text{fix-int}\,(\text{eval\$}\,(\mathbf{t},\ w,\ a)))
\end{array}$$

THEOREM: equal-fix-int-to-ilessp
$$(\text{fix-int}\,(x) = \text{fix-int}\,(y)) \rightarrow (\neg\ \text{ilessp}\,(x,\ y))$$

THEOREM: eval$-list-bagint-0-implies-equal-for-ilessp
$$\begin{array}{ll}
((\neg\ \text{ilessp}\,(\text{itimes-list}\,(\text{eval\$}\,(\texttt{'list}, & \\
\qquad\qquad \text{bagint}\,(\text{itimes-factors}\,(v),\ \text{itimes-factors}\,(w)), & \\
\qquad\qquad a)), & \\
\qquad \texttt{0})) & \\
\ \wedge\quad (\neg\ \text{ilessp}\,(\texttt{0}, & \\
\qquad\qquad \text{itimes-list}\,(\text{eval\$}\,(\texttt{'list}, & \\
\qquad\qquad\qquad\qquad \text{bagint}\,(\text{itimes-factors}\,(v), & \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{itimes-factors}\,(w)), & \\
\qquad\qquad\qquad\qquad a))))) & \\
\rightarrow\quad ((\neg\ \text{ilessp}\,(\text{eval\$}\,(\mathbf{t},\ v,\ a),\ \text{eval\$}\,(\mathbf{t},\ w,\ a))) & \\
\qquad \wedge\quad (\neg\ \text{ilessp}\,(\text{eval\$}\,(\mathbf{t},\ w,\ a),\ \text{eval\$}\,(\mathbf{t},\ v,\ a))))
\end{array}$$

```
;; The rewrite rule ILESSP-TRICHOTOMY seemed to mess up the proof of the following,
;; so I'm just going to leave it disabled.
```

EVENT: Disable ilessp-trichotomy.

THEOREM: correctness-of-cancel-itimes-ilessp-factors
eval$ (**t**, x, a) = eval$ (**t**, cancel-itimes-ilessp-factors (x), a)

```
;; OK -- now, the zero cases.
```

EVENT: Enable lessp-count-listp-cdr.

DEFINITION:
disjoin-equalities-with-0 (*factors*)
= **if** listp (cdr (*factors*))
then list ('or,
list ('equal, list ('fix-int, car (*factors*)), ''0),
disjoin-equalities-with-0 (cdr (*factors*)))
else list ('equal, list ('fix-int, car (*factors*)), ''0) **endif**

EVENT: Disable lessp-count-listp-cdr.

DEFINITION:
cancel-factors-0 (x)
= **if** listp (x)
then if car (x) = 'equal
then if cadr (x) = ''0
then let *factors* **be** itimes-factors (caddr (x))
in
if listp (cdr (*factors*))
then disjoin-equalities-with-0 (*factors*)
else x **endif endlet**
elseif caddr (x) = ''0
then let *factors* **be** itimes-factors (cadr (x))
in
if listp (cdr (*factors*))
then disjoin-equalities-with-0 (*factors*)
else x **endif endlet**
else x **endif**
else x **endif**
else x **endif**

DEFINITION:
some-eval$s-to-0 ($x$, a)
= **if** listp (x)

then (fix-int (eval$ (**t**, car (*x*), *a*)) = `0`)
∨ some-eval$s-to-0 (cdr (*x*), *a*)
else f endif

THEOREM: eval$-disjoin-equalities-with-0
listp (*lst*)
→ (eval$ (**t**, disjoin-equalities-with-0 (*lst*), *a*)
= some-eval$s-to-0 (*lst*, *a*))

THEOREM: some-eval$s-to-0-append
some-eval$s-to-0 (append (*x*, *y*), *a*)
= (some-eval$s-to-0 (*x*, *a*) ∨ some-eval$s-to-0 (*y*, *a*))

THEOREM: some-eval$s-to-0-eliminator
some-eval$s-to-0 (*x*, *a*) = (itimes-list (eval$ (`'list`, *x*, *a*)) = `0`)

THEOREM: listp-cdr-factors-implies-integerp
listp (cdr (itimes-factors (*v*))) → integerp (eval$ (**t**, *v*, *a*))

THEOREM: correctness-of-cancel-factors-0
eval$ (**t**, *x*, *a*) = eval$ (**t**, cancel-factors-0 (*x*), *a*)

```
;; and now for inequalities...
```

EVENT: Enable lessp-count-listp-cdr.

DEFINITION:
conjoin-inequalities-with-0 (*factors*, *parity*)
= **if** listp (cdr (*factors*))
then if *parity*
then list (`'or`,
list (`'and`,
list (`'ilessp`, `''0`, car (*factors*)),
conjoin-inequalities-with-0 (cdr (*factors*), **t**)),
list (`'and`,
list (`'ilessp`, car (*factors*), `''0`),
conjoin-inequalities-with-0 (cdr (*factors*), **f**)))
else list (`'or`,
list (`'and`,
list (`'ilessp`, car (*factors*), `''0`),
conjoin-inequalities-with-0 (cdr (*factors*), **t**)),
list (`'and`,
list (`'ilessp`, `''0`, car (*factors*)),
conjoin-inequalities-with-0 (cdr (*factors*), **f**))) **endif**
elseif *parity* **then** list (`'ilessp`, `''0`, car (*factors*))
else list (`'ilessp`, car (*factors*), `''0`) **endif**

EVENT: Disable lessp-count-listp-cdr.

DEFINITION:
cancel-factors-ilessp-0 (x)
= **if** listp (x)
then if car (x) = `'ilessp`
then if cadr (x) = `''0`
then let *factors* **be** itimes-factors (caddr (x))
in
if listp (cdr (*factors*))
then conjoin-inequalities-with-0 (*factors*, **t**)
else x **endif endlet**
elseif caddr (x) = `''0`
then let *factors* **be** itimes-factors (cadr (x))
in
if listp (cdr (*factors*))
then conjoin-inequalities-with-0 (*factors*, **f**)
else x **endif endlet**
else x **endif**
else x **endif**

THEOREM: conjoin-inequalities-with-0-eliminator
listp (x)
→ (eval$ (**t**, conjoin-inequalities-with-0 (x, *parity*), a)
= **if** *parity* **then** ilessp (`0`, itimes-list (eval$ (`'list`, x, a)))
else ilessp (itimes-list (eval$ (`'list`, x, a)), `0`) **endif**)

THEOREM: correctness-of-cancel-factors-ilessp-0
eval$ (**t**, x, a) = eval$ (**t**, cancel-factors-ilessp-0 (x), a)

EVENT: Disable equal-itimes-list-eval$-list-delete-new-1.

EVENT: Disable equal-itimes-list-eval$-list-delete-new-2.

EVENT: Disable itimes-itimes-list-eval$-list-delete.

EVENT: Disable equal-itimes-list-eval$-list-bagdiff.

EVENT: Disable itimes-list-eval$-factors-lemma.

EVENT: Disable itimes-list-eval$-factors-lemma-prime.

EVENT: Disable itimes-list-eval$-factors.

EVENT: Disable iplus-or-itimes-term-integerp-eval$.

EVENT: Disable eval$-list-bagint-0.

EVENT: Disable eval$-list-bagint-0-implies-equal.

EVENT: Disable izerop-ilessp-0-relationship.

EVENT: Disable ilessp-itimes-list-eval$-list-delete-helper-1.

EVENT: Disable ilessp-itimes-list-eval$-list-delete-helper-2.

EVENT: Disable ilessp-itimes-list-eval$-list-delete.

EVENT: Disable ilessp-itimes-list-eval$-list-delete-prime-helper-1.

EVENT: Disable ilessp-itimes-list-eval$-list-delete-prime-helper-2.

EVENT: Disable ilessp-itimes-list-eval$-list-delete-prime.

EVENT: Disable ilessp-0-itimes.

EVENT: Disable ilessp-itimes-0.

EVENT: Disable listp-cdr-factors-implies-integerp.

```
;; We presumably have better meta-lemmas now, but if we want we
;; can disable those (i.e., correctness-of-cancel-itimes-factors,
;; correctness-of-cancel-itimes-ilessp-factors,
;; correctness-of-cancel-factors-0, and
;; correctness-of-cancel-factors-ilessp-0) and enable the two
;; mentioned below:
```

EVENT: Disable correctness-of-cancel-itimes.

EVENT: Disable correctness-of-cancel-itimes-ilessp.

```
;; I'll disable some rules now, finally, that I'd previously thought
;; would be OK but now fear because of potential nasty backchaining.
```

EVENT: Disable not-integerp-implies-not-equal-iplus.

EVENT: Disable not-integerp-implies-not-equal-itimes.

EVENT: Disable subbagp-subsetp.

EVENT: Disable eval$-list-bagint-0-implies-equal-for-ilessp.

```
; ---------- Cancel ineg terms from equalities and inequalities ----------
```

DEFINITION:
split-out-ineg-terms (*x*)
= **if** listp (*x*)
then let *pair* **be** split-out-ineg-terms (cdr (*x*)),
a **be** car (*x*)
in
if listp (*a*)
then if car (*a*) = 'ineg
then cons (car (*pair*), cons (cadr (*a*), cdr (*pair*)))
elseif (car (*a*) = 'quote)
∧ negativep (cadr (*a*))
∧ (negative-guts (cadr (*a*)) ≠ 0)
then cons (car (*pair*),
cons (list ('quote, negative-guts (cadr (*a*))),
cdr (*pair*)))
else cons (cons (*a*, car (*pair*)), cdr (*pair*)) **endif**
else cons (cons (*a*, car (*pair*)), cdr (*pair*)) **endif endlet**
else cons (**nil**, **nil**) **endif**

DEFINITION:
remove-inegs (*x*, *y*)
= **let** *xpair* **be** split-out-ineg-terms (*x*),
ypair **be** split-out-ineg-terms (*y*)
in
if listp (cdr (*xpair*)) ∨ listp (cdr (*ypair*))

```
    then cons (append (cdr (ypair), car (xpair)),
                      append (cdr (xpair), car (ypair)))
    else f endif endlet
```

DEFINITION:

```
iplus-or-ineg-term (x)
=   (listp (x) ∧ ((car (x) = 'ineg) ∨ (car (x) = 'iplus)))
```

DEFINITION:

```
make-cancel-ineg-terms-equality (x)
=   let new-fringes be remove-inegs (iplus-fringe (cadr (x)),
                                     iplus-fringe (caddr (x)))
    in
    if new-fringes
    then if iplus-or-ineg-term (cadr (x))
         then if iplus-or-ineg-term (caddr (x))
              then list ('equal,
                         iplus-tree (car (new-fringes)),
                         iplus-tree (cdr (new-fringes)))
              else list ('if,
                         list ('integerp, caddr (x)),
                         list ('equal,
                               iplus-tree (car (new-fringes)),
                               iplus-tree (cdr (new-fringes))),
                         list ('quote, f)) endif
         elseif iplus-or-ineg-term (caddr (x))
         then list ('if,
                    list ('integerp, cadr (x)),
                    list ('equal,
                          iplus-tree (car (new-fringes)),
                          iplus-tree (cdr (new-fringes))),
                    list ('quote, f))
         else x endif
    else x endif endlet
```

DEFINITION:

```
cancel-ineg-terms-from-equality (x)
=   if listp (x) ∧ (car (x) = 'equal)
    then make-cancel-ineg-terms-equality (x)
    else x endif
```

```
;; The following was created from nqthm-macroexpand with arguments
;; and or make-cancel-ineg-terms-equality iplus-or-ineg-term
```

DEFINITION:

```
cancel-ineg-terms-from-equality-expanded (x)
=   if listp (x)
    then if car (x) = 'equal
         then if remove-inegs (iplus-fringe (cadr (x)),
                               iplus-fringe (caddr (x)))
              then if listp (cadr (x))
                   then case on car (car (cdr (x))):
                        case = ineg
                        then if listp (caddr (x))
                             then case on car (car (cdr (cdr (x)))):
                                  case = ineg
                                  then list ('equal,
                                             iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                                            iplus-fringe (caddr (x))))),
                                             iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                            iplus-fringe (caddr (x))))))
                                  case = iplus
                                   then list ('equal,
                                              iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                                             iplus-fringe (caddr (x))))),
                                              iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                             iplus-fringe (caddr (x))))))
                                  otherwise list ('if,
                                                  list ('integerp,
                                                        caddr (x)),
                                                  list ('equal,
                                                        iplus-tree (car (remove-inegs (iplus-fringe (cadr (x
                                                                                                    iplus-fringe (caddr (
                                                        iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x
                                                                                                    iplus-fringe (caddr (
                                                                    list ('quote, f)) endcase
                             else list ('if,
                                        list ('integerp, caddr (x)),
                                        list ('equal,
                                              iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                                             iplus-fringe (caddr (x))))),
                                              iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                             iplus-fringe (caddr (x))))),
                                        list ('quote, f)) endif
                        case = iplus
                         then if listp (caddr (x))
                              then case on car (car (cdr (cdr (x)))):
                                   case = ineg
                                   then list ('equal,
```

```
                    iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                   iplus-fringe (caddr (x))))),
                    iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                   iplus-fringe (caddr (x))))))
          case = iplus
           then list ('equal,
                      iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                     iplus-fringe (caddr (x))))),
                      iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                     iplus-fringe (caddr (x))))))
          otherwise list ('if,
                          list ('integerp,
                                caddr (x)),
                          list ('equal,
                                iplus-tree (car (remove-inegs (iplus-fringe (cadr (
                                                               iplus-fringe (caddr
                                iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (
                                                               iplus-fringe (caddr
                          list ('quote, f)) endcase
    else list ('if,
               list ('integerp, caddr (x)),
               list ('equal,
                     iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                    iplus-fringe (caddr (x))))),
                     iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                    iplus-fringe (caddr (x))))),
               list ('quote, f)) endif
otherwise if listp (caddr (x))
          then case on car (car (cdr (cdr (x)))):
               case = ineg
               then list ('if,
                          list ('integerp,
                                cadr (x)),
                          list ('equal,
                                iplus-tree (car (remove-inegs (iplus-fringe (cadr (x
                                                               iplus-fringe (caddr (
                                iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x
                                                               iplus-fringe (caddr (
                          list ('quote, f))
               case = iplus
                then list ('if,
                           list ('integerp,
                                 cadr (x)),
                           list ('equal,
```

```
                                                    iplus-tree (car (remove-inegs (iplus-fringe (cadr (
                                                                                   iplus-fringe (caddr
                                                    iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (
                                                                                   iplus-fringe (caddr
                                                list ('quote, f))
                                            otherwise x endcase
                                        else x endif endcase
                    elseif listp (caddr (x))
                    then case on car (car (cdr (cdr (x)))):
                         case = ineg
                         then list ('if,
                                    list ('integerp, cadr (x)),
                                    list ('equal,
                                         iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                                        iplus-fringe (caddr (x))))),
                                         iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                        iplus-fringe (caddr (x))))),
                                    list ('quote, f))
                         case = iplus
                          then list ('if,
                                     list ('integerp, cadr (x)),
                                     list ('equal,
                                          iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                                         iplus-fringe (caddr (x))))),
                                          iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                         iplus-fringe (caddr (x))))),
                                     list ('quote, f))
                         otherwise x endcase
                    else x endif
               else x endif
          else x endif
     else x endif
```

THEOREM: cancel-ineg-terms-from-equality-cancel-ineg-terms-from-equality-expanded
cancel-ineg-terms-from-equality-expanded (*x*)
= cancel-ineg-terms-from-equality (*x*)

EVENT: Disable cancel-ineg-terms-from-equality-expanded.

THEOREM: integerp-eval$-iplus-or-ineg-term
iplus-or-ineg-term (*x*) → integerp (eval$ (**t**, *x*, *a*))

EVENT: Disable iplus-or-ineg-term.

THEOREM: eval$-iplus-list-car-remove-inegs
remove-inegs $(x,\ y)$
$\rightarrow$ (iplus-list (eval$ ('list, car (remove-inegs $(x,\ y)$), a))
= iplus (iplus-list (eval$ ('list, car (split-out-ineg-terms (x)), a)),
iplus-list (eval$ ('list, cdr (split-out-ineg-terms (y)), a))))

THEOREM: eval$-iplus-list-cdr-remove-inegs
remove-inegs $(x,\ y)$
$\rightarrow$ (iplus-list (eval$ ('list, cdr (remove-inegs $(x,\ y)$), a))
= iplus (iplus-list (eval$ ('list, car (split-out-ineg-terms (y)), a)),
iplus-list (eval$ ('list, cdr (split-out-ineg-terms (x)), a))))

THEOREM: minus-ineg
$((x \in \mathbf{N}) \wedge (x \neq 0)) \rightarrow ((-\ x) = \text{ineg}\,(x))$

THEOREM: iplus-list-eval$-car-split-out-ineg-terms
iplus-list (eval$ ('list, car (split-out-ineg-terms (x)), a))
= iplus (iplus-list (eval$ ('list, x, a)),
iplus-list (eval$ ('list, cdr (split-out-ineg-terms (x)), a)))

EVENT: Disable remove-inegs.

THEOREM: correctness-of-cancel-ineg-terms-from-equality
eval$ (**t**, x, a) = eval$ (**t**, cancel-ineg-terms-from-equality-expanded (x), a)

DEFINITION:
make-cancel-ineg-terms-inequality (x)
= **let** *new-fringes* **be** remove-inegs (iplus-fringe (cadr (x)),
iplus-fringe (caddr (x)))
in
if *new-fringes*
then list ('ilessp,
iplus-tree (car (*new-fringes*)),
iplus-tree (cdr (*new-fringes*)))
else x **endif endlet**

DEFINITION:
cancel-ineg-terms-from-inequality (x)
= **if** listp $(x) \wedge$ (car (x) = 'ilessp)
then if iplus-or-ineg-term (cadr (x))
then make-cancel-ineg-terms-inequality (x)
elseif iplus-or-ineg-term (caddr (x))
then make-cancel-ineg-terms-inequality (x)
else x **endif**
else x **endif**

```
;; The following was created from nqthm-macroexpand with arguments
;; and or make-cancel-ineg-terms-inequality iplus-or-ineg-term
```

DEFINITION:
cancel-ineg-terms-from-inequality-expanded (*x*)
= **if** listp (*x*)
then if car (*x*) = 'ilessp
then if listp (cadr (*x*))
then case on car (car (cdr (*x*))):
case = *ineg*
then if remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*)))
then list ('ilessp,
iplus-tree (car (remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*))))),
iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*))))))
else *x* **endif**
case = *iplus*
then if remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*)))
then list ('ilessp,
iplus-tree (car (remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*))))),
iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*))))))
else *x* **endif**
otherwise if listp (caddr (*x*))
then case on car (car (cdr (cdr (*x*)))):
case = *ineg*
then if remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*)))
then list ('ilessp,
iplus-tree (car (remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*))))),
iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*))))))
else *x* **endif**
case = *iplus*
then if remove-inegs (iplus-fringe (cadr (*x*)),
iplus-fringe (caddr (*x*)))
then list ('ilessp,
iplus-tree (car (remove-inegs (iplus-fringe (cadr (*x*)),

```
                                                                    iplus-fringe (caddr (x)))
                                                  iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                                 iplus-fringe (caddr (x))
                                        else x endif
                                  otherwise x endcase
                            else x endif endcase
                  elseif listp (caddr (x))
                  then case on car (car (cdr (cdr (x)))):
                        case = ineg
                        then if remove-inegs (iplus-fringe (cadr (x)),
                                              iplus-fringe (caddr (x)))
                                then list ('ilessp,
                                           iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                                          iplus-fringe (caddr (x))))),
                                           iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                          iplus-fringe (caddr (x))))))
                                else x endif
                        case = iplus
                         then if remove-inegs (iplus-fringe (cadr (x)),
                                               iplus-fringe (caddr (x)))
                                 then list ('ilessp,
                                            iplus-tree (car (remove-inegs (iplus-fringe (cadr (x)),
                                                                           iplus-fringe (caddr (x))))),
                                            iplus-tree (cdr (remove-inegs (iplus-fringe (cadr (x)),
                                                                           iplus-fringe (caddr (x))))))
                                 else x endif
                        otherwise x endcase
                  else x endif
            else x endif
      else x endif
```

THEOREM: cancel-ineg-terms-from-inequality-cancel-ineg-terms-from-inequality-expanded
cancel-ineg-terms-from-inequality-expanded (*x*)
= cancel-ineg-terms-from-inequality (*x*)

EVENT: Disable cancel-ineg-terms-from-inequality-expanded.

THEOREM: correctness-of-cancel-ineg-terms-from-inequality
eval$ (**t**, *x*, *a*) = eval$ (**t**, cancel-ineg-terms-from-inequality-expanded (*x*), *a*)

EVENT: Disable minus-ineg.

EVENT: Disable integerp-eval$-iplus-or-ineg-term.

```
; ---------- Eliminating constants ----------

;; We want to combine in terms like (iplus 3 (iplus x 7)).  Also, when
;; two iplus terms are equated or in-equated, there should only be a
;; natural number summand on at most one side.  Finally, if one adds 1
;; to the right side of a strict inequality, a stronger inequality (in
;; a certain sense) is obtained by removing the 1 and making a non-strict
;; inequality in the other direction.
```

THEOREM: plus-iplus
$((i \in \mathbf{N}) \wedge (j \in \mathbf{N})) \rightarrow ((i + j) = \text{iplus}\,(i, j))$

THEOREM: iplus-constants
$\text{iplus}\,(1 + i, \text{iplus}\,(1 + j, x)) = \text{iplus}\,((1 + i) + (1 + j), x)$

THEOREM: numberp-is-integerp
$(w \in \mathbf{N}) \rightarrow \text{integerp}\,(w)$

THEOREM: difference-idifference
$((x \in \mathbf{N}) \wedge (y \in \mathbf{N}) \wedge (x \leq y)) \rightarrow ((y - x) = \text{idifference}\,(y, x))$

THEOREM: cancel-constants-equal-lemma
$((m \in \mathbf{N}) \wedge (n \in \mathbf{N}))$
$\rightarrow \quad ((\text{iplus}\,(m, x) = \text{iplus}\,(n, y))$
$= \quad$ **if** $m < n$ **then** $\text{fix-int}\,(x) = \text{iplus}\,(n - m, y)$
else $\text{iplus}\,(m - n, x) = \text{fix-int}\,(y)$ **endif**)

THEOREM: cancel-constants-equal
$(\text{iplus}\,(1 + i, x) = \text{iplus}\,(1 + j, y))$
$=$ **if** $i < j$ **then** $\text{fix-int}\,(x) = \text{iplus}\,(j - i, y)$
else $\text{iplus}\,(i - j, x) = \text{fix-int}\,(y)$ **endif**

THEOREM: ilessp-add1
$(y \in \mathbf{N}) \rightarrow (\text{ilessp}\,(x, 1 + y) = (\neg\ \text{ilessp}\,(y, x)))$

THEOREM: ilessp-add1-iplus
$(y \in \mathbf{N}) \rightarrow (\text{ilessp}\,(x, \text{iplus}\,(1 + y, z)) = (\neg\ \text{ilessp}\,(\text{iplus}\,(y, z), x)))$

THEOREM: cancel-constants-ilessp-lemma-1
$((m \in \mathbf{N}) \wedge (n \in \mathbf{N}))$
$\rightarrow \quad (\text{ilessp}\,(\text{iplus}\,(m, x), \text{iplus}\,(n, y))$
$= \quad$ **if** $m < n$ **then** $\text{ilessp}\,(x, \text{iplus}\,(n - m, y))$
else $\text{ilessp}\,(\text{iplus}\,(m - n, x), y)$ **endif**)

THEOREM: cancel-constants-ilessp-lemma-2
$((m \in \mathbf{N}) \wedge (n \in \mathbf{N}))$
$\rightarrow$ (ilessp (iplus (m, x), iplus (n, y))
= **if** $m < n$ **then** $\neg$ ilessp (iplus $((n - m) - 1, y), x)$
else ilessp (iplus $(m - n, x), y)$ **endif**)

THEOREM: cancel-constants-ilessp
ilessp (iplus $(1 + i, x)$, iplus $(1 + j, y)$)
= **if** $i < j$ **then** $\neg$ ilessp (iplus $((j - i) - 1, y), x)$
else ilessp (iplus $(i - j, x), y)$ **endif**

EVENT: Disable plus-iplus.

EVENT: Disable numberp-is-integerp.

EVENT: Disable difference-idifference.

```
; ---------- Final DEFTHEORY event ----------

;; I'll go ahead and include iplus-list and itimes-list and lemmas
;; about them that were developed.

;; I've left out ILESSP-TRICHOTOMY because I'm scared it will slow
;; things down too much.  But it certainly represents useful
;; information.
```

EVENT: Let us define the theory *integers* to consist of the following events: ileq, idifference, integerp-fix-int, integerp-iplus, integerp-idifference, integerp-ineg, integerp-iabs, integerp-itimes, fix-int-remover, fix-int-fix-int, fix-int-iplus, fix-int-idifference, fix-int-ineg, fix-int-iabs, fix-int-itimes, ineg-iplus, ineg-ineg, ineg-fix-int, ineg-of-non-integerp, ineg-0, iplus-left-id, iplus-right-id, iplus-0-left, iplus-0-right, commutativity2-of-iplus, commutativity-of-iplus, associativity-of-iplus, iplus-cancellation-1, iplus-cancellation-2, iplus-ineg1, iplus-ineg2, iplus-fix-int1, iplus-fix-int2, idifference-fix-int1, idifference-fix-int2, iplus-list, eval$-list-append, iplus-list-append, iplus-ineg3, iplus-ineg4, correctness-of-cancel-iplus, ilessp-fix-int-1, ilessp-fix-int-2, iplus-cancellation-1-for-ilessp, iplus-cancellation-2-for-ilessp, correctness-of-cancel-iplus-ilessp, itimes-0-left, itimes-0-right, itimes-fix-int1, itimes-fix-int2, commutativity-of-itimes, itimes-distributes-over-iplus-proof, itimes-distributes-over-iplus, commutativity2-of-itimes, associativity-of-itimes, equal-itimes-0, equal-itimes-1, equal-itimes-minus-1, itimes-1-arg1, quotient-remainder-uniqueness, division-theorem, itimes-ineg-1, itimes-ineg-2, itimes-cancellation-1, itimes-cancellation-2, itimes-cancellation-3, integerp-iquotient, integerp-iremainder,

integerp-idiv, integerp-imod, integerp-iquo, integerp-irem, iquotient-fix-int1, iquotient-fix-int2, iremainder-fix-int1, iremainder-fix-int2, idiv-fix-int1, idiv-fix-int2, imod-fix-int1, imod-fix-int2, iquo-fix-int1, iquo-fix-int2, irem-fix-int1, irem-fix-int2, fix-int-iquotient, fix-int-iremainder, fix-int-idiv, fix-int-imod, fix-int-iquo, fix-int-irem, itimes-list, itimes-list-append, member-append, equal-fix-int, subsetp, correctness-of-cancel-itimes, correctness-of-cancel-itimes-ilessp, ilessp-strict, eval$-list-cons, eval$-list-nlistp, eval$-litatom, eval$-quote, eval$-other, iplus-x-y-ineg-x, correctness-of-cancel-ineg, integerp-iplus-list, eval$-iplus-list-delete, eval$-iplus-list-bagdiff, itimes-tree-ineg, itimes-factors, itimes–1, equal-ineg-ineg, ilessp-ineg-ineg, fix-int-eval$-itimes-tree-rec, eval$-itimes-tree-ineg, ineg-eval$-itimes-tree-ineg, iplus-eval$-itimes-tree-ineg, itimes-eval$-itimes-tree-ineg, iplus-or-itimes-term, cancel-itimes-factors, cancel-itimes-factors-expanded, cancel-itimes-factors-expanded-cancel-itimes-factors, membership-of-0-implies-itimes-list-is-0, member-0-eval$-list, correctness-of-cancel-itimes-factors, cancel-itimes-ilessp-factors, bagint-singleton, ilessp-itimes-list-eval$-list-bagdiff, ilessp-itimes-list-eval$-list-bagdiff-corollary-1, member-0-itimes-factors-yields-0, member-0-itimes-factors-yields-0-ilessp-consequence-1, member-0-itimes-factors-yields-0-ilessp-consequence-2, ilessp-itimes-list-eval$-list-bagdiff-corollary-2, correctness-of-cancel-itimes-ilessp-factors, disjoin-equalities-with-0, cancel-factors-0, some-eval$s-to-0, eval$-disjoin-equalities-with-0, some-eval$s-to-0-append, some-eval$s-to-0-eliminator, correctness-of-cancel-factors-0, conjoin-inequalities-with-0, cancel-factors-ilessp-0, split-out-ineg-terms, correctness-of-cancel-ineg-terms-from-equality, correctness-of-cancel-ineg-terms-from-inequality, iplus-constants, cancel-constants-equal, ilessp-add1, ilessp-add1-iplus, cancel-constants-ilessp.

```
; ------------------------------------------------------------
; was lists.events
; ------------------------------------------------------------


; ------------------------------------------------------------------------------------
; Basic Functions on Lists
; ------------------------------------------------------------------------------------

; ---------------------------------- DEFINITIONS ----------------------------------

; Besides these functions, the theory of lists assumes APPEND is defined.
```

DEFINITION:
firstn $(n,\ l)$
$=$ **if** $\neg$ listp (l) **then nil**
elseif $n \simeq$ `0` **then nil**
else cons (car (l), firstn $(n - 1,$ cdr $(l))$) **endif**

DEFINITION:
init (*value*, *length*)
= **if** *length* $\simeq$ 0 **then nil**
else cons (*value*, init (*value*, *length* − 1)) **endif**

DEFINITION:
lastcdr (*l*)
= **if** ¬ listp (*l*) **then** *l*
else lastcdr (cdr (*l*)) **endif**

DEFINITION:
length (*l*)
= **if** ¬ listp (*l*) **then** 0
else 1 + length (cdr (*l*)) **endif**

DEFINITION:
plist (*l*)
= **if** ¬ listp (*l*) **then nil**
else cons (car (*l*), plist (cdr (*l*))) **endif**

DEFINITION:
plistp (*l*)
= **if** ¬ listp (*l*) **then** *l* = **nil**
else plistp (cdr (*l*)) **endif**

DEFINITION:
restn (*n*, *l*)
= **if** ¬ listp (*l*) **then** *l*
elseif *n* $\simeq$ 0 **then** *l*
else restn (*n* − 1, cdr (*l*)) **endif**

DEFINITION:
reverse (*l*)
= **if** ¬ listp (*l*) **then nil**
else append (reverse (cdr (*l*)), list (car (*l*))) **endif**

; ---------- THEOREMS ----------

; ---------- LISTP ----------

THEOREM: listp-append
listp (append (*a*, *b*)) = (listp (*a*) ∨ listp (*b*))

THEOREM: listp-firstn
listp (firstn (*n*, *l*)) = (listp (*l*) ∧ (*n* $\not\simeq$ 0))

THEOREM: listp-init
$\text{listp}\,(\text{init}\,(val,\ n)) = (n \not\simeq \mathtt{0})$

THEOREM: listp-lastcdr
$\text{listp}\,(\text{lastcdr}\,(l)) = \mathbf{f}$

THEOREM: listp-plist
$\text{listp}\,(\text{plist}\,(l)) = \text{listp}\,(l)$

THEOREM: listp-restn
$\text{listp}\,(\text{restn}\,(n,\ l)) = (n < \text{length}\,(l))$

THEOREM: listp-reverse
$\text{listp}\,(\text{reverse}\,(l)) = \text{listp}\,(l)$

```
; ---------- plistp ----------
```

THEOREM: plistp-nlistp
$(l \simeq \mathbf{nil}) \rightarrow (\text{plistp}\,(l) = (l = \mathbf{nil}))$

THEOREM: plistp-cons
$\text{plistp}\,(\text{cons}\,(a,\ l)) = \text{plistp}\,(l)$

THEOREM: plistp-append
$\text{plistp}\,(\text{append}\,(a,\ b)) = \text{plistp}\,(b)$

THEOREM: plistp-firstn
$\text{plistp}\,(\text{firstn}\,(n,\ l))$

THEOREM: plistp-init
$\text{plistp}\,(\text{init}\,(val,\ n))$

THEOREM: plistp-lastcdr
$\text{plistp}\,(\text{lastcdr}\,(l)) = \text{plistp}\,(l)$

THEOREM: plistp-plist
$\text{plistp}\,(\text{plist}\,(l))$

THEOREM: plistp-restn
$\text{plistp}\,(\text{restn}\,(n,\ l)) = \text{plistp}\,(l)$

THEOREM: plistp-reverse
$\text{plistp}\,(\text{reverse}\,(a))$

```
; ---------- LENGTH ----------
```

THEOREM: equal-length-0
$(\text{length}\,(l) = \texttt{0}) = (\neg\ \text{listp}\,(l))$

THEOREM: length-nlistp
$(x \simeq \textbf{nil}) \rightarrow (\text{length}\,(x) = \texttt{0})$

THEOREM: length-cons
$\text{length}\,(\text{cons}\,(a,\ x)) = (1 + \text{length}\,(x))$

THEOREM: length-append
$\text{length}\,(\text{append}\,(a,\ b)) = (\text{length}\,(a) + \text{length}\,(b))$

THEOREM: length-firstn
$\text{length}\,(\text{firstn}\,(n,\ l))$
$=$ **if** $n \leq \text{length}\,(l)$ **then** $\text{fix}\,(n)$
else $\text{length}\,(l)$ **endif**

THEOREM: length-init
$\text{length}\,(\text{init}\,(bit,\ n)) = \text{fix}\,(n)$

THEOREM: length-lastcdr
$\text{length}\,(\text{lastcdr}\,(l)) = \texttt{0}$

THEOREM: length-plist
$\text{length}\,(\text{plist}\,(l)) = \text{length}\,(l)$

THEOREM: length-restn
$\text{length}\,(\text{restn}\,(n,\ l)) = (\text{length}\,(l) - n)$

THEOREM: length-reverse
$\text{length}\,(\text{reverse}\,(l)) = \text{length}\,(l)$

```
; ---------- APPEND ----------
```

THEOREM: append-left-id
$(\neg\ \text{listp}\,(a)) \rightarrow (\text{append}\,(a,\ b) = b)$

THEOREM: append-nil
$\text{append}\,(a,\ \textbf{nil}) = \text{plist}\,(a)$

THEOREM: associativity-of-append
$\text{append}\,(\text{append}\,(a,\ b),\ c) = \text{append}\,(a,\ \text{append}\,(b,\ c))$

THEOREM: associativity-of-append-inverse
$\text{append}\,(a,\ \text{append}\,(b,\ c)) = \text{append}\,(\text{append}\,(a,\ b),\ c)$

EVENT: Disable associativity-of-append-inverse.

THEOREM: append-firstn-restn
append (firstn (*i*, *l*), restn (*i*, *l*)) = *l*

THEOREM: append-init-list
append (init (*v*, *n*), list (*v*)) = init (*v*, 1 + *n*)

THEOREM: append-init-init
append (init (*v*, *i*), init (*v*, *j*)) = init (*v*, *i* + *j*)

THEOREM: append-lastcdr-arg1
append (lastcdr (*a*), *b*) = *b*

THEOREM: append-lastcdr-arg2
append (*l*, lastcdr (*l*)) = *l*

THEOREM: append-plist
append (plist (*a*), *b*) = append (*a*, *b*)

THEOREM: append-plist-lastcdr
append (plist (*l*), lastcdr (*l*)) = *l*

```
; ---------- FIRSTN ----------
```

THEOREM: firstn-with-large-index
(length (*l*) ≤ *n*) → (firstn (*n*, *l*) = plist (*l*))

THEOREM: firstn-with-non-number-index
(*n* ∉ **N**) → (firstn (*n*, *l*) = firstn (0, *l*))

THEOREM: firstn-nlistp
(¬ listp (*l*)) → (firstn (*n*, *l*) = **nil**)

THEOREM: firstn-0
firstn (0, *l*) = **nil**

THEOREM: firstn-cons
firstn (*n*, cons (*a*, *b*))
= **if** *n* ≃ 0 **then nil**
else cons (*a*, firstn (*n* − 1, *b*)) **endif**

THEOREM: firstn-append
firstn (*n*, append (*a*, *b*))
= **if** *n* ≤ length (*a*) **then** firstn (*n*, *a*)
else append (*a*, firstn (*n* − length (*a*), *b*)) **endif**

THEOREM: firstn-firstn
firstn (*i*, firstn (*n*, *l*))
= **if** *i* < *n* **then** firstn (*i*, *l*)
else firstn (*n*, *l*) **endif**

THEOREM: firstn-init
firstn (*n*, init (*v*, *i*))
= **if** *n* < *i* **then** init (*v*, *n*)
else init (*v*, *i*) **endif**

THEOREM: firstn-lastcdr
firstn (*n*, lastcdr (*l*)) = **nil**

THEOREM: firstn-plist
firstn (*n*, plist (*l*)) = firstn (*n*, *l*)

```
; FIRSTN-RESTN has no obvious reduction.


; ---------- INIT ----------
```

THEOREM: car-init
(*n* ≯ 0) → (car (init (*v*, *n*)) = *v*)

THEOREM: init-with-non-number-index
(*n* ∉ **N**) → (init (*v*, *n*) = init (*v*, 0))

THEOREM: init-add1
init (*x*, 1 + *n*) = cons (*x*, init (*x*, *n*))

THEOREM: init-0
init (*v*, 0) = **nil**

```
; ---------- LASTCDR ----------
```

THEOREM: lastcdr-nlistp
(¬ listp (*l*)) → (lastcdr (*l*) = *l*)

THEOREM: lastcdr-cons
lastcdr (cons (*a*, *l*)) = lastcdr (*l*)

THEOREM: lastcdr-append
lastcdr (append (*a*, *b*)) = lastcdr (*b*)

THEOREM: lastcdr-firstn
lastcdr (firstn (*n*, *l*)) = **nil**

THEOREM: lastcdr-init
lastcdr (init (*v*, *i*)) = **nil**

THEOREM: lastcdr-lastcdr
lastcdr (lastcdr (*l*)) = lastcdr (*l*)

THEOREM: lastcdr-plist
lastcdr (plist (*l*)) = **nil**

THEOREM: lastcdr-restn
lastcdr (restn (*n*, *l*)) = lastcdr (*l*)

THEOREM: lastcdr-reverse
lastcdr (reverse (*l*)) = **nil**

```
; ---------- PLIST ----------
```

THEOREM: equal-plist
plistp (*l*) → (plist (*l*) = *l*)

THEOREM: plist-nlistp
(¬ listp (*x*)) → (plist (*x*) = **nil**)

THEOREM: plist-cons
plist (cons (*a*, *l*)) = cons (*a*, plist (*l*))

THEOREM: plist-append
plist (append (*a*, *b*)) = append (*a*, plist (*b*))

THEOREM: plist-firstn
plist (firstn (*n*, *l*)) = firstn (*n*, *l*)

THEOREM: plist-init
plist (init (*v*, *i*)) = init (*v*, *i*)

THEOREM: plist-lastcdr
plist (lastcdr (*l*)) = **nil**

THEOREM: plist-plist
plist (plist (*l*)) = plist (*l*)

THEOREM: plist-restn
plist (restn (*n*, *l*)) = restn (*n*, plist (*l*))

THEOREM: plist-reverse
plist (reverse (*l*)) = reverse (*l*)

```
; ---------- RESTN ----------
```

THEOREM: restn-with-non-number-index
($n \notin \mathbf{N}$) → (restn (*n*, *l*) = restn (0, *l*))

THEOREM: restn-with-large-index
(length (*l*) ≤ *n*) → (restn (*n*, *l*) = lastcdr (*l*))

THEOREM: restn-nlistp
(¬ listp (*l*)) → (restn (*n*, *l*) = *l*)

THEOREM: restn-0
restn (0, *l*) = *l*

THEOREM: restn-cons
restn (*n*, cons (*a*, *b*))
= **if** *n* ≃ 0 **then** cons (*a*, *b*)
else restn (*n* − 1, *b*) **endif**

THEOREM: restn-append
restn (*n*, append (*a*, *b*))
= **if** *n* ≤ length (*a*) **then** append (restn (*n*, *a*), *b*)
else restn (*n* − length (*a*), *b*) **endif**

THEOREM: restn-firstn
restn (*n*, firstn (*i*, *l*))
= **if** *i* ≤ *n* **then nil**
else firstn (*i* − *n*, restn (*n*, *l*)) **endif**

THEOREM: restn-init
restn (*n*, init (*v*, *i*)) = init (*v*, *i* − *n*)

THEOREM: restn-lastcdr
restn (*n*, lastcdr (*l*)) = lastcdr (*l*)

THEOREM: restn-plist
restn (*n*, plist (*l*)) = plist (restn (*n*, *l*))

EVENT: Disable restn-plist.

THEOREM: restn-restn
restn (*i*, restn (*j*, *l*)) = restn (*j* + *i*, *l*)

; ---------- REVERSE ----------

THEOREM: reverse-nlistp
$(\neg\ \text{listp}\,(l)) \rightarrow (\text{reverse}\,(l) = \textbf{nil})$

THEOREM: reverse-cons
$\text{reverse}\,(\text{cons}\,(a,\ l)) = \text{append}\,(\text{reverse}\,(l),\ \text{list}\,(a))$

THEOREM: reverse-append
$\text{reverse}\,(\text{append}\,(a,\ b)) = \text{append}\,(\text{reverse}\,(b),\ \text{reverse}\,(a))$

THEOREM: reverse-init
$\text{reverse}\,(\text{init}\,(v,\ n)) = \text{init}\,(v,\ n)$

THEOREM: reverse-lastcdr
$\text{reverse}\,(\text{lastcdr}\,(l)) = \textbf{nil}$

THEOREM: reverse-plist
$\text{reverse}\,(\text{plist}\,(l)) = \text{reverse}\,(l)$

THEOREM: reverse-reverse
$\text{plistp}\,(l) \rightarrow (\text{reverse}\,(\text{reverse}\,(l)) = l)$

; --

EVENT: Let us define the theory *lists* to consist of the following events: append-firstn-restn, append-init-init, append-init-list, append-lastcdr-arg1, append-lastcdr-arg2, append-left-id, append-nil, append-plist, append-plist-lastcdr, associativity-of-append, car-init, equal-length-0, equal-plist, firstn-0, firstn-append, firstn-cons, firstn-firstn, firstn-init, firstn-lastcdr, firstn-nlistp, firstn-plist, firstn-with-large-index, firstn-with-non-number-index, init-0, init-add1, init-with-non-number-index, lastcdr-append, lastcdr-cons, lastcdr-firstn, lastcdr-init, lastcdr-lastcdr, lastcdr-nlistp, lastcdr-plist, lastcdr-restn, lastcdr-reverse, length-append, length-cons, length-firstn, length-init, length-lastcdr, length-nlistp, length-plist, length-restn, length-reverse, listp-append, listp-firstn, listp-init, listp-lastcdr, listp-plist, listp-restn, listp-reverse, plist-append, plist-cons, plist-firstn, plist-init, plist-lastcdr, plist-nlistp, plist-plist, plist-restn, plist-reverse, plistp-append, plistp-cons, plistp-firstn, plistp-init, plistp-lastcdr, plistp-nlistp, plistp-plist, plistp-restn, plistp-reverse, restn-0, restn-append, restn-cons, restn-firstn, restn-init, restn-lastcdr, restn-nlistp, restn-restn, restn-with-large-index, restn-with-non-number-index, reverse-append, reverse-cons, reverse-init, reverse-lastcdr, reverse-nlistp, reverse-plist, reverse-reverse.

```
; ------------------------------------------------------------
; was piton-basis.events
; ------------------------------------------------------------


;; The following stuff is from Piton, defs.events
```

EVENT: Add the shell *p-state*, with recognizer function symbol *p-statep* and 9 accessors: *p-pc*, with type restriction (none-of) and default value zero; *p-ctrl-stk*, with type restriction (none-of) and default value zero; *p-temp-stk*, with type restriction (none-of) and default value zero; *p-prog-segment*, with type restriction (none-of) and default value zero; *p-data-segment*, with type restriction (none-of) and default value zero; *p-max-ctrl-stk-size*, with type restriction (none-of) and default value zero; *p-max-temp-stk-size*, with type restriction (none-of) and default value zero; *p-word-size*, with type restriction (none-of) and default value zero; *p-psw*, with type restriction (none-of) and default value zero.

```
; Each element of the program segment is a program.  A program is a list
; of the form:

; (name (formal1 formal2 ... formaln)
;       ((temp1 init1)
;        ...
;        (tempk initk))
;       instr1
;       instr2
;       ...
;       instrm)

; The name and each formal and temp is a symbol.  The initial values
; of the temps are tagged values.  The instrs are either LISTP objects
; which may be optionalled labelled by litatoms.  To label an
; instruction ins with the label lab, write (dl lab comment ins).
; Comment may be any object.  It is completely ignored.

; Roughly speaking, a CALL of name binds the formals to the top n
; elements of the temp-stk (removing them from that stack and building
; a ctrl-stk frame), binds the temps to the corresponding tagged
; values (also in the ctrl-stk frame), and executes each instruction.

; When a symbol is used as a variable name in an instruction we will
; look for its value as though it were stored in an alist whose keys
; were listed in the following order:
```

```
;    formal1 ... formaln temp1 ... tempk.

; That is, a variable reference is first to the formals, then to the
; temps.  In the case of duplications we use the first occurrence in
; the list.

; The following function sets the psw to the one given.  If that is
; anything besides 'RUN the Piton machine halts.
```

DEFINITION:
p-halt (p, psw)
= p-state (p-pc (p),
p-ctrl-stk (p),
p-temp-stk (p),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
psw)

```
; We characterize the errorneous psws as all those besides 'RUN and
; 'HALT:
```

DEFINITION:
errorp (psw) = (($psw \neq$ 'run) $\wedge$ ($psw \neq$ 'halt))

```
; Association lists.
```

DEFINITION:
put-assoc (val, $name$, $alist$)
= **if** $alist \simeq$ **nil then** $alist$
elseif $name$ = caar ($alist$) **then** cons (cons ($name$, val), cdr ($alist$))
else cons (car ($alist$), put-assoc (val, $name$, cdr ($alist$))) **endif**

```
; The following function determines whether name is bound in alist.
```

DEFINITION:
definedp ($name$, $alist$)
= **if** $alist \simeq$ **nil then f**
elseif $name$ = caar ($alist$) **then t**
else definedp ($name$, cdr ($alist$)) **endif**

```
; This function is used to obtain the definition of a Piton variable
; or subroutine name.
```

DEFINITION: definition (*name*, *alist*) = assoc (*name*, *alist*)

```
; These two functions are used for manipulating Piton variable values.
```

DEFINITION: value (*name*, *alist*) = cdr (definition (*name*, *alist*))

DEFINITION:
put-value (*val*, *name*, *alist*) = put-assoc (*val*, *name*, *alist*)

```
; This function is used to strip the values out of an alist.
```

DEFINITION:
strip-cdrs (*alist*)
= **if** *alist* ≃ **nil then nil**
else cons (cdr (car (*alist*)), strip-cdrs (cdr (*alist*))) **endif**

DEFINITION:
strip-cadrs (*alist*)
= **if** *alist* ≃ **nil then nil**
else cons (cadr (car (*alist*)), strip-cadrs (cdr (*alist*))) **endif**

```
; Piton programs.

; When I fetch the definitions of programs from the prog-segment I
; want to decompose them into the following pieces:
```

DEFINITION: name (*d*) = car (*d*)

DEFINITION: formal-vars (*d*) = cadr (*d*)

DEFINITION: temp-var-dcls (*d*) = caddr (*d*)

DEFINITION: program-body (*d*) = cdddr (*d*)

```
; The ``locals'' of a program is the union of the formal-vars
; and the temporaries.
```

DEFINITION:
local-vars (*d*) = append (formal-vars (*d*), strip-cars (temp-var-dcls (*d*)))

```
; Addresses

; Our addresses are tagged ``address pairs''.  The pair specifies
; an area name in some memory segment and an offset from the
; base address of that area.  We call such a pair an ``adp''
; (pronounced ``a d p'').
```

DEFINITION: adp-name $(adp) = \text{car}\,(adp)$

DEFINITION: adp-offset $(adp) = \text{cdr}\,(adp)$

```
; The tag indicates which segment of memory the address points into.
; For example, at the Piton level, the tag PC means into the prog
; segment and the tag ADDR means into the data segment.  At lower levels
; there are other tags and other segments.  An object is an address
; pair if it and its associated segment satisfy the following relation
```

DEFINITION:
adpp $(x,\ segment)$
$=$ (listp (x)
$\wedge$ (adp-offset $(x) \in \mathbf{N}$)
$\wedge$ definedp (adp-name (x), $segment$)
$\wedge$ (adp-offset $(x) <$ length (value (adp-name (x), $segment$))))

```
; Program counters are slightly different from addresses because
; the areas in the program segment are not (name . value) but
; (name formal-vars temps . program-body).
```

DEFINITION:
pcpp $(x,\ segment)$
$=$ (listp (x)
$\wedge$ (adp-offset $(x) \in \mathbf{N}$)
$\wedge$ definedp (adp-name (x), $segment$)
$\wedge$ (adp-offset (x)
$<$ length (program-body (definition (adp-name (x), $segment$)))))

```
; We nevertheless often manipulate them with the same functions we
; do adps.  For example, add1-adp, below, increments the offset of an
; adp by 1.  We will write (add1-adp pc) even though pc is a pcpp.
```

DEFINITION:
add-adp $(adp,\ n) = \text{cons}\,(\text{adp-name}\,(adp), \text{adp-offset}\,(adp) + n)$

DEFINITION: add1-adp (*adp*) = add-adp (*adp*, 1)

DEFINITION:
sub-adp (*adp*, *n*) = cons (adp-name (*adp*), adp-offset (*adp*) − *n*)

DEFINITION: sub1-adp (*adp*) = sub-adp (*adp*, 1)

DEFINITION:
get (*n*, *lst*)
= **if** *n* ≃ 0 **then** car (*lst*)
else get (*n* − 1, cdr (*lst*)) **endif**

DEFINITION:
put (*val*, *n*, *lst*)
= **if** *n* ≃ 0
then if listp (*lst*) **then** cons (*val*, cdr (*lst*))
else list (*val*) **endif**
else cons (car (*lst*), put (*val*, *n* − 1, cdr (*lst*))) **endif**

DEFINITION:
fetch-adp (*adp*, *segment*)
= get (adp-offset (*adp*), value (adp-name (*adp*), *segment*))

DEFINITION:
deposit-adp (*val*, *adp*, *segment*)
= put-value (put (*val*, adp-offset (*adp*), value (adp-name (*adp*), *segment*)),
adp-name (*adp*),
segment)

```
; While get enumerates the elements of lst from 0 starting at
; the left, we also need the function that enumerates the elements
; from 0 starting at the right.  This is used in the specification
; of the fetch-temp-stk instruction.  The name rget stands for "right get"
; or perhaps "reversed get":
```

DEFINITION: rget (*n*, *lst*) = get ((length (*lst*) − *n*) − 1, *lst*)

```
; The corresponding put function:
```

DEFINITION:
rput (*val*, *n*, *lst*) = put (*val*, (length (*lst*) − *n*) − 1, *lst*)

```
; Tagged Objects

; We keep all Piton objects tagged.  The form we use is
; (tag obj) where tag is one of a fixed set of LITATOMs
; enumerated by the function p-objectp below.  The following
; three functions let us create a tagged object, access the
; tag field and access the contents field.
```

DEFINITION: tag (*type*, *obj*) = list (*type*, *obj*)

DEFINITION: type (*const*) = car (*const*)

DEFINITION: untag (*const*) = cadr (*const*)

```
; Tagged Addresses

; We have a variety of objects in the system that are tagged address pairs.
; We call these things ``addresses'' and define primitives for recognizing
; and manipulating them.  At this level we do not care what the tags are,
; we just strip them off or duplicate them as necessary.  That is, to
; increment (tag (DELTA1 . 25)) we build (tag (DELTA1 . 26)) without
; inspecting tag.
```

DEFINITION: addressp (*x*, *segment*) = adpp (untag (*x*), *segment*)

DEFINITION: area-name (*x*) = adp-name (untag (*x*))

DEFINITION: offset (*x*) = adp-offset (untag (*x*))

DEFINITION:
add-addr (*addr*, *n*) = tag (type (*addr*), add-adp (untag (*addr*), *n*))

DEFINITION: add1-addr (*addr*) = add-addr (*addr*, 1)

DEFINITION:
sub-addr (*addr*, *n*) = tag (type (*addr*), sub-adp (untag (*addr*), *n*))

DEFINITION: sub1-addr (*addr*) = sub-addr (*addr*, 1)

DEFINITION:
fetch (*addr*, *segment*) = fetch-adp (untag (*addr*), *segment*)

DEFINITION:
deposit (*val*, *addr*, *segment*) = deposit-adp (*val*, untag (*addr*), *segment*)

```
; Note:  Addresses are just one of many data types.  We could define
; tagged versions of all of our data manipulation functions.  E.g.,
; we could define (nat-plus x y) = (tag 'nat (plus (untag x) (untag y))).
; However we do not.  Instead we use tag and untag freely below.
; The reason we go to the trouble of defining tagged address functions
; is because we use those constructs so often -- not just to implement
; the semantics of the ADDR data type operations.  For example, fetching
; and depositing into memory, getting instructions, etc., all use
; tagged address manipulation.
```

```
; Booleans

; Unlike the logic, the PITON machine uses the LITATOM 'T for true and
; the LITATOM 'F for false.  This means we can't use the primitive
; AND, OR, and NOT functions of the logic to do AND-BOOL, OR-BOOL, and
; NOT-BOOL.  The reason I adopted this curious convention is so that T
; and F can be written inside of quoted list constants -- i.e., in
; PITON programs.  That is, if in the logic you write '(PUSH-CONSTANT
; (BOOL T)) that T is the litatom 'T, not (TRUE).
```

DEFINITION: booleanp$(x) = ((x =$ `'t`$) \vee (x =$ `'f`$))$

DEFINITION:
bool(x)
= tag(`'bool`,
if x **then** `'t`
else `'f` **endif**)

```
; Because we know we only apply the boolean operators to booleanps we
; can skimp on the tests.
```

DEFINITION:
or-bool(x, y)
= **if** $x =$ `'f` **then** y
else `'t` **endif**

DEFINITION:
and-bool(x, y)
= **if** $x =$ `'f` **then** `'f`
else y **endif**

DEFINITION:
not-bool(x)
= **if** $x =$ `'f` **then** `'t`
else `'f` **endif**

```
; *** 14-12-87/jsm:  The following defn was added during the proofs
; for i->m.
```

DEFINITION:
xor-bool(x, y)
= **if** $x =$ `'f` **then** y
elseif $y =$ `'f` **then** `'t`
else `'f` **endif**

```
; Naturals
```

DEFINITION:
small-naturalp $(i,$ *word-size*$) = ((i \in \mathbf{N}) \wedge (i < \exp(2,$ *word-size*$)))$

DEFINITION:
bool-to-nat (*flg*)
= **if** *flg* = 'f **then** 0
else 1 **endif**

```
; *** 16-12-87/jsm:  "fix-small" was renamed "fix-small-natural" so that
; a parallel "fix-small-integer" could be defined.
```

DEFINITION:
fix-small-natural $(n,$ *word-size*$) = (n$ **mod** $\exp(2,$ *word-size*$))$

```
; Integers

; The functions below are defined so that if given integerps they
; return integerps.  In particular, they never create (MINUS 0) though
; they might pass one through.

;; INTEGERP and ILESSP are defined in integers.events
;(defn integerp (i)
;  (or (numberp i)
;      (and (negativep i)
;    (not (equal (negative-guts i) 0)))))

;(defn ilessp (i j)
;  (if (negativep i)
;      (if (negativep j)
;   (lessp (negative-guts j)
;  (negative-guts i))
;   t)
;      (if (negativep j)
;   f
;   (lessp i j))))
```

DEFINITION:
small-integerp $(i,$ *word-size*$)$
= $(\text{integerp}(i)$
$\wedge \quad (\neg\ \text{ilessp}(i, -\exp(2,$ *word-size* $- 1)))$
$\wedge \quad \text{ilessp}(i, \exp(2,$ *word-size* $- 1)))$

```
;; IPLUS is defined in integers.events
;(defn iplus (i j)
;  (if (negativep i)
;      (if (negativep j)
;   (minus
;     (plus (negative-guts i)
;   (negative-guts j)))
;   (if (lessp j (negative-guts i))
;       (minus (difference (negative-guts i) j))
;       (difference j (negative-guts i))))
;      (if (negativep j)
;   (if (lessp i (negative-guts j))
;       (minus (difference (negative-guts j) i))
;       (difference i (negative-guts j)))
;   (plus i j))))

;; INEG is the same as INEGATE and is defined integer.events
;(defn inegate (i)
;  (if (negativep i)
;      (negative-guts i)
;      (if (zerop i)
;   0
;   (minus i))))
```

DEFINITION: inegate (i) = ineg (i)

```
;; IDIFFERENCE is defined in integers.events
;(defn idifference (i j)
;  (iplus i (inegate j)))
```

DEFINITION:
fix-small-integer $(i,$ *word-size*$)$
= **if** small-integerp $(i,$ *word-size*$)$ **then** i
elseif negativep (i) **then** iplus $(i,$ exp $(2,$ *word-size*$))$
else iplus $(i, -$ exp $(2,$ *word-size*$))$ **endif**

```
; Bit vectors
```

DEFINITION: bitp $(x) = ((x = 0) \vee (x = 1))$

DEFINITION:
bit-vectorp $(x,\ n)$
= **if** $x \simeq$ **nil then** $(x =$ **nil**$) \wedge (n \simeq 0)$
else $(n \not\simeq 0) \wedge$ bitp $($car $(x)) \wedge$ bit-vectorp $($cdr $(x),\ n - 1)$ **endif**

DEFINITION:
or-bit (*bit1*, *bit2*)
= **if** *bit1* = `0`
then if *bit2* = `0` **then** `0`
else `1` **endif**
else `1` **endif**

DEFINITION:
not-bit (*bit*)
= **if** *bit* = `0` **then** `1`
else `0` **endif**

DEFINITION:
and-bit (*bit1*, *bit2*)
= **if** *bit1* = `0` **then** `0`
elseif *bit2* = `0` **then** `0`
else `1` **endif**

DEFINITION:
xor-bit (*bit1*, *bit2*)
= **if** *bit1* = `0`
then if *bit2* = `0` **then** `0`
else `1` **endif**
elseif *bit2* = `0` **then** `1`
else `0` **endif**

DEFINITION:
or-bitv (*a*, *b*)
= **if** *a* ≃ **nil** **then nil**
else cons (or-bit (car (*a*), car (*b*)), or-bitv (cdr (*a*), cdr (*b*))) **endif**

DEFINITION:
not-bitv (*a*)
= **if** *a* ≃ **nil** **then nil**
else cons (not-bit (car (*a*)), not-bitv (cdr (*a*))) **endif**

DEFINITION:
and-bitv (*a*, *b*)
= **if** *a* ≃ **nil** **then nil**
else cons (and-bit (car (*a*), car (*b*)), and-bitv (cdr (*a*), cdr (*b*))) **endif**

DEFINITION:
xor-bitv (*a*, *b*)
= **if** *a* ≃ **nil** **then nil**
else cons (xor-bit (car (*a*), car (*b*)), xor-bitv (cdr (*a*), cdr (*b*))) **endif**

DEFINITION:
all-but-last (a)
= **if** $a \simeq$ **nil then nil**
elseif cdr $(a) \simeq$ **nil then nil**
else cons (car (a), all-but-last (cdr (a))) **endif**

DEFINITION: rsh-bitv (a) = cons (0, all-but-last (a))

DEFINITION: lsh-bitv (a) = append (cdr (a), list (0))

DEFINITION:
all-zero-bitvp (a)
= **if** listp (a) **then** (car $(a) = 0$) $\wedge$ all-zero-bitvp (cdr (a))
else t endif

```
; Stacks
```

DEFINITION: push $(x,\ stk)$ = cons $(x,\ stk)$

DEFINITION: top (stk) = car (stk)

DEFINITION: pop (stk) = cdr (stk)

DEFINITION:
popn $(n,\ x)$
= **if** $n \simeq 0$ **then** x
else popn $(n - 1,$ cdr $(x))$ **endif**

DEFINITION: top1 (stk) = top (pop (stk))

DEFINITION: top2 (stk) = top (pop (pop (stk)))

```
; Labels

; In an earlier version of this I used the Lisp PROG convention for
; denoting labels:  a label is an atom in a list that otherwise
; contains non-atomic instructions.  The trouble with that convention
; is that the notions of the ``ith element'' and the ``length'' of
; such instruction lists have to be redefined to skip labels.  I
; anticipate much work going into the accessing of items at given
; addresses and wanted to use the same kind of fetch for program items
; as for data items.  Thus I have decided that each item in a list is
; an instruction and some are labelled.

; Here is a function which labels ins with lab:
```

DEFINITION: dl (*lab*, *comment*, *ins*) = list (`'dl`, *lab*, *comment*, *ins*)

```
; The following function recognizes labelled items:
```

DEFINITION: labelledp (*x*) = (car (*x*) = `'dl`)

```
; This function extracts the instruction from the labelling.
```

DEFINITION:
unlabel (*x*)
= **if** labelledp (*x*) **then** cadddr (*x*)
else *x* **endif**

```
; I suspect we will never want to know whether something was labelled
; or not but rather deal with everything as though it weren't.  So I
; will disable the unlabel function.
```

EVENT: Disable unlabel.

```
; The following function determines whether x is defined as a label
; in lst.
```

DEFINITION:
find-labelp (*x*, *lst*)
= **if** *lst* ≃ **nil** **then f**
elseif labelledp (car (*lst*)) ∧ (*x* = cadr (car (*lst*))) **then t**
else find-labelp (*x*, cdr (*lst*)) **endif**

```
; This function counts the number of items in lst before the first
; one labelled x.
```

DEFINITION:
find-label (*x*, *lst*)
= **if** *lst* ≃ **nil** **then** `0`
elseif labelledp (car (*lst*)) ∧ (*x* = cadr (car (*lst*))) **then** `0`
else 1 + find-label (*x*, cdr (*lst*)) **endif**

```
; Generally we convert labels into tagged PCs.  We use the following
; function:
```

DEFINITION:
pc (*lab*, *program*)
= tag ('pc, cons (name (*program*), find-label (*lab*, program-body (*program*))))

```
; The following function is used by the JUMP-CASE instruction to
; check that a list of labels has been provided.
```

DEFINITION:
all-find-labelp (*lab-lst*, *lst*)
= **if** *lab-lst* ≃ **nil then t**
else find-labelp (car (*lab-lst*), *lst*)
∧ all-find-labelp (cdr (*lab-lst*), *lst*) **endif**

```
;  P-Level Primitives
```

DEFINITION:
p-objectp (*x*, *p*)
= (listp (*x*)
∧ (cddr (*x*) = **nil**)
∧ **case on** type (*x*):
case = *nat*
then small-naturalp (untag (*x*), p-word-size (*p*))
case = *int*
then small-integerp (untag (*x*), p-word-size (*p*))
case = *bitv*
then bit-vectorp (untag (*x*), p-word-size (*p*))
case = *bool*
then booleanp (untag (*x*))
case = *addr*
then adpp (untag (*x*), p-data-segment (*p*))
case = *pc*
then pcpp (untag (*x*), p-prog-segment (*p*))
case = *subr*
then definedp (untag (*x*), p-prog-segment (*p*))
otherwise f endcase)

DEFINITION:
p-objectp-type (*type*, *x*, *p*) = ((type (*x*) = *type*) ∧ p-objectp (*x*, *p*))

DEFINITION:
all-p-objectps (*lst*, *p*)
= **if** *lst* ≃ **nil then** *lst* = **nil**
else p-objectp (car (*lst*), *p*) ∧ all-p-objectps (cdr (*lst*), *p*) **endif**

```
; The following function checks that it is ok to increment
; the pc by 1.
```

DEFINITION: add1-p-pc (*p*) = add1-addr (p-pc (*p*))

DEFINITION:
add1-p-pcp (*p*) = pcpp (untag (add1-p-pc (*p*)), p-prog-segment (*p*))

DEFINITION:
p-current-program (*p*) = definition (area-name (p-pc (*p*)), p-prog-segment (*p*))

DEFINITION:
p-current-instruction (*p*)
= unlabel (get (offset (p-pc (*p*)), program-body (p-current-program (*p*))))

DEFINITION: p-frame (*bindings*, *ret-pc*) = list (*bindings*, *ret-pc*)

DEFINITION: bindings (*frame*) = car (*frame*)

DEFINITION: ret-pc (*frame*) = cadr (*frame*)

DEFINITION:
p-frame-size (*frame*) = (2 + length (bindings (*frame*)))

DEFINITION:
p-ctrl-stk-size (*ctrl-stk*)
= **if** *ctrl-stk* ≃ **nil** **then** 0
else p-frame-size (top (*ctrl-stk*))
\+ p-ctrl-stk-size (cdr (*ctrl-stk*)) **endif**

DEFINITION:
local-varp (*var*, *ctrl-stk*) = definedp (*var*, bindings (top (*ctrl-stk*)))

DEFINITION:
local-var-value (*var*, *ctrl-stk*) = value (*var*, bindings (top (*ctrl-stk*)))

DEFINITION:
set-local-var-value (*val*, *var*, *ctrl-stk*)
= push (p-frame (put-value (*val*, *var*, bindings (top (*ctrl-stk*))),
ret-pc (top (*ctrl-stk*))),
pop (*ctrl-stk*))

```
;; REVERSE is defined in lists.events
;(defn reverse (x)
;  (if (nlistp x)
;      nil
;      (append (reverse (cdr x)) (list (car x)))))
```

DEFINITION:
first-n (n, x)
= **if** $n \simeq$ `0` **then nil**
else cons (car (x), first-n ($n - 1$, cdr (x))) **endif**

DEFINITION:
pair-formal-vars-with-actuals (*formal-vars*, *temp-stk*)
= pairlist (*formal-vars*, reverse (first-n (length (*formal-vars*), *temp-stk*)))

```
; Note:  I considered defining this function with:
;   (reverse (pairlist (reverse formal-vars) temp-stk))
; I am not sure which defn is the easiest to work with.
```

DEFINITION:
pair-temps-with-initial-values (*temp-var-dcls*)
= **if** *temp-var-dcls* $\simeq$ **nil then nil**
else cons (cons (caar (*temp-var-dcls*), cadar (*temp-var-dcls*)),
pair-temps-with-initial-values (cdr (*temp-var-dcls*))) **endif**

DEFINITION:
make-p-call-frame (*formal-vars*, *temp-stk*, *temp-var-dcls*, *ret-pc*)
= p-frame (append (pair-formal-vars-with-actuals (*formal-vars*, *temp-stk*),
pair-temps-with-initial-values (*temp-var-dcls*)),
ret-pc)

```
; If (FOO (X Y Z) ((A a) (B b) (C c)) ...) is a Piton program then
; the frame constructed when FOO is CALLed with temp-stk
; '(2 1 0 ...) looks like this:

; (((X . 0) (Y . 1) (Z . 2) (A . a) (B . b) (C . c))
;  ret-pc)
```

```
;----------------------------------------------------------------------------
;  Individual Instructions.

;  I now begin defining the instruction set.  Suppose ins is an
;  instruction opcode.  Then I will define 3 functions, each with the
;  word ins in its name.  The three functions are:

;  p-ins-okp:  T or F according to whether the given instruction
;              is well-formed and legal to execute in a given state.

;  p-ins-step: the new state produced by the execution of a legal
;              well-formed instruction

;  icode-ins: the icode for this instruction.
```

```
;  (CALL subr) Push a new frame onto the ctrl-stk binding the
; formals and temps of the subroutine subr to their
; actual values and saving the current return pc.
; Transfer control to the beginning of subr.
```

DEFINITION:
p-call-okp (*ins*, *p*)
= ((p-max-ctrl-stk-size (*p*)
$\not<$ p-ctrl-stk-size (push (make-p-call-frame (formal-vars (definition (cadr (*ins*),
p-prog-segment (*p*))),
p-temp-stk (*p*),
temp-var-dcls (definition (cadr (*ins*),
p-prog-segment (*p*))),
add1-addr (p-pc (*p*))),
p-ctrl-stk (*p*))))
$\wedge$ (length (p-temp-stk (*p*))
$\not<$ length (formal-vars (definition (cadr (*ins*),
p-prog-segment (*p*))))))

DEFINITION:
p-call-step (*ins*, *p*)
= p-state (tag ('pc, cons (cadr (*ins*), 0)),
push (make-p-call-frame (formal-vars (definition (cadr (*ins*),
p-prog-segment (*p*))),
p-temp-stk (*p*),
temp-var-dcls (definition (cadr (*ins*),
p-prog-segment (*p*))),
add1-addr (p-pc (*p*))),
p-ctrl-stk (*p*)),
popn (length (formal-vars (definition (cadr (*ins*),
p-prog-segment (*p*)))),
p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

```
; We now discuss how we compile a call instruction.  This means
; explaining a lot about the lower level machines and the structure of
; the compiler.
```

```
; Associated with every opcode, xxx, is a function named
; icode-xxx which generates the symbolic machine code
; for the opcode.  Code generated is called ‘‘icode’’ because it is
; the code executed by the I machine.

; Every icode-xxx function gets three arguments:

; ins          the Piton instruction being compiled
; pcn          the offset of the pc pointing to that Piton instruction
; program      the Piton program containing the instruction

; The pcn may be used as a unique seed for icode labels generated by the
; compiler.  It is guaranteed to be a NUMBERP.

; Icode contains labels.  What are these labels?  That is, what objects
; do we use for icode labels?  The answer is Piton PC-type objects.
; Icode labels are tagged so we can distinguish them from other things.
; The tag we use is 'PC!  At the icode level the initials PC should be
; thought of as standing for LABEL.

; Thus, in icode, (SUBR . 25) is a label!  It is tagged PC and thus
; a reference to it in the instruction stream would be denoted
; (PC (SUBR . 25)).  Remember that in icode PC stands for LABEL.

; To distinguish a reference from the defining occurrence, the defining
; occurrence is tagged exactly as in Piton code with DL.

; Thus, the defining occurrence of (SUBR . 25) would be in an instruction
; of the form
;     (DL (SUBR . 25) comment icode-ins)

; For each valid pc in a Piton program the icompiler defines that pc
; as a label at the beginning of the basic block of icode generated
; for the instruction at that pc.  Thus, the PC objects of Piton
; become the labels of icode programs and are given meaning.  In
; mapping the Piton PCs down to icode labels it is not necessary to do
; anything.  That is the beauty of our using the tag PC at the icode
; level.  Once upon a time I marked icode labels with the tag 'ILAB.
; But that means that when mapping the stacks or data memory down to
; the i level I had to go through and change all the PCs to ILABs.
; Now I don't.  The stacks at the r level are identical to the stacks
; at the i level!  (In the toy version, the r level stacks had pcs on
; them and they were transformed into i-level pcs.  That is no longer
; done.  The objects on the i-level stack become labels without any
```

```
; work.)

; User defined Piton labels, e.g., LOOP, are systematically eliminated
; by the icompiler into these pc type labels.  That is, when the
; defining occurrence of LOOP occurs at pcn 25 in a Piton program, it
; simply dropped in favor of the automatically generated pc-style
; label (DL (name . 25) & &).  When LOOP is used in a JUMP
; statement the corresponding pc-style label is computed and used
; instead.

; So far, we have established that all valid PC type objects in Piton
; are defined icode labels.  Furthermore, the label objects used in
; Piton are systematically eliminated in favor of pc-style labels.
; However, there are icode labels that are NOT valid Piton PC type
; objects!

; The problem arises because icode needs internal labels within basic
; blocks or to mark the prelude and postlude.  To generate distinct
; system-internal labels I exploit the fact that all valid Piton pc
; labels are of the form (name . number) where number is less than
; the length of the program body.   I generate labels of the
; form (name anything) and I also generate a single label of the form
; (name . length) where length is the length of the program body.
; I use labels of the first form to mark the PRELUDE for subroutine
; entry.  The label is (name PRELUDE).  I use the second label to
; mark the postlude.   Because these are icode labels they
; are tagged in the icode with either the PC tag, indicating a
; reference, e.g., (PC (name PRELUDE)), or the DL tag,
; indicating the defining occurrence (DL (name POSTLUDE) & &).
; Despite the fact that they are sometimes tagged PC they are not
; Piton PC.  Remember:  in icode PC stands for LABEL!

; Once upon a time I marked the postlude with (name POSTLUDE).  But
; that required that every program body end with a (RET) because
; many instructions assumed that the label marking the beginning of
; the block of instructions icompiled for the "next" Piton instruction
; was (name . n+1) where n is the position of the current instruction.
; (If the last instr is a (RET) then every other instr has a
; next one that is so labelled.)  I don't like enforcing the
; "last instr is a (RET)" convention, and it also makes the
; resulting implementation less efficient because often programs
; hit a (RET) which merely jumps to the instruction following the (RET)
; rather than just falling off the end.
```

```
; It is important that every icode label be structured enough to
; permit me to determine in which program it is defined.  All my
; labels have the name of the parent program as the car and then
; arbitrary stuff after.  I need to be able to find definitions since
; icode sometimes jumps to foreign labels (upon subroutine exit) and
; it must be possible to figure out the new pc from the supplied
; label.


; Historical Note and Commentary on Icode Labels:

; The presence in the icode of labels is rather unexpected to me.
; Originally labels were eliminated by the icompiler.  It is also odd
; that the high level language provides the notion of PC and the low
; level language uses labels!  Its a reversal of conceptual depth.
; The reason is that our pcs can be non-local.  That is, it is
; permitted in Piton for subroutine FOO to manipulate the non-local pc
; (BAR . 25) and to pass it to BAR and jumped to.

; If I were to map PCs in Piton down to pcs in the icode it would be
; necessary for the icompiler to compute icode pcs from Piton pcs.
; But that can't be done without an overall view of the system -- it
; cannot be done on a program by program basis.  So instead of using
; icode pcs below I use symbolic labels.  The fact that the labels I
; use are not atomic is irrelevant -- imagine icode labels were
; symbols if you wish.  The natural time to replace these symbols is
; at link time when we have absolute addresses.  An address like (BAR
; . 25) can be thought of as simply an alternative entry point into
; BAR and thus is like a subroutine in itself.

; Of course, I could make the icompiler have a second pass in which
; labels are eliminated in favor of icode-level pcs.  (That pass was
; called REMOVE-LABELS in HRIL.)  But the proof of the correctness of
; the icompiler is in the R->I transition and that is already the
; hardest of the three transitions to prove.  So in a way it is nice
; to eliminate a complexity from there.  Introducing the complexity in
; I->L is ok too both because that transition is the simplest and
; because it is already dealing with the idea that jumping to a
; symbolic subroutine address is the same as jumping to an absolute
; address where that subroutine is located.  So I don't think pushing
; label removal to the final transition is necessarily going to
; complicate the proof down there very much.
```

```
; Now, on with my icompiler!
```

DEFINITION:
icode-call (*ins*, *pcn*, *program*)
= list (’(cpush_*),
tag (’pc, cons (name (*program*), 1 + *pcn*)),
’(jump_*),
tag (’pc, cons (cadr (*ins*), ’(prelude))))

```
; Notes:  See the "Guide to Decrypting I-level Opcodes" near the
; defn of link-instruction-alist for an explanation of the meaning
; of the i-level opcodes used in the icode- fns.

; Four observations come to mind about this piece of icode.
; First, while the tag function has heretofore been used only to
; produce Piton objects, I am here using it to produce icode objects,
; namely PCs.  Second, the return address pushed onto csp is an icode
; label not a pc.  Third, the return address label, (name . pcn+1) is
; not defined here.  That is because it is defined when I process the
; pcn+1st instruction.  In fact, icompile-program-body lays down the
; DLs implicit in the use of that label.  Fourth, the (name
; prelude) label is not defined here.  In fact, generate-prelude takes
; care of its defn.

; There is more to the compilation of a CALL instruction than merely
; the sequence of instructions generated for the CALL itself:  one
; must also consider the prelude that is part of every subroutine
; entry.  The prelude must be undone by the postlude.  I therefore
; include below the CALL code, the prelude, and the postlude.
```

DEFINITION:
generate-prelude1 (*temp-var-dcls*)
= **if** *temp-var-dcls* $\simeq$ **nil then nil**
else cons (’(cpush_*),
cons (cadr (car (*temp-var-dcls*)),
generate-prelude1 (cdr (*temp-var-dcls*)))) **endif**

DEFINITION:
generate-prelude2 (*formal-vars*)
= **if** *formal-vars* $\simeq$ **nil then nil**
else cons (’(cpush_<tsp>+),
generate-prelude2 (cdr (*formal-vars*))) **endif**

DEFINITION:
generate-prelude (*program*)
= append (list (dl (cons (name (*program*), '(prelude)),
'(prelude),
'(cpush_cfp)),
'(move_cfp_csp)),
append (generate-prelude1 (reverse (temp-var-dcls (*program*))),
generate-prelude2 (formal-vars (*program*))))

```
;  Observe how the formals and temps are arranged on the stack.
;  Suppose that the header for the subroutine FOO is (FOO (X Y Z) ((A
;  a) (B b) (C c)) ...) where a, b, and c are constants.  At the
;  beginning of the execution of (CALL FOO) the temp-stk and ctrl-stk
;  look like this:

;     ...              ...
;      e1               c1  <- csp
;       x
;       y
;       z  <- tsp

;  Our stacks on the lower level grow downward.  For example, a
;  typical value for tsp might be '((TEMP-STK) . 25), which denotes
;  the 25th word from the base of the area named '(TEMP-STK).  The
;  contents of that word is the item on the top of the stack.  To push
;  something, tsp would become '((TEMP-STK) . 24).  To pop something,
;  tsp would become '((TEMP-STK) . 26).

;  Note that the base of the area named '(TEMP-STK) is not the
;  base of the stack!

;  After execution of (CALL FOO) the stacks are:

;     ...              ...
;      e1  <- tsp       c1
;                    ret-pc
;                    old-fp  <- cfp
;                         c           +5
;                         b           +4
;                         a           +3
;                         z           +2
;                         y           +1
;                         x  <- csp   +0
```

DEFINITION:
find-position-of-var (*var*, *lst*)
= **if** *lst* $\simeq$ **nil** **then** 0
elseif *var* = car (*lst*) **then** 0
else 1 + find-position-of-var (*var*, cdr (*lst*)) **endif**

DEFINITION:
offset-from-csp (*var*, *program*)
= find-position-of-var (*var*, local-vars (*program*))

```
;  We now define
```

DEFINITION:
generate-postlude (*program*)
= list (dl (cons (name (*program*), length (program-body (*program*))),
'(postlude),
'(move_csp_cfp)),
'(cpop_cfp),
'(cpop_pc))

```
;  (RET)              Return from the current subroutine, leaving temp-stk
;                     untouched but restoring ctrl-stk.  We make a special
;                     case out of top-level RETs:  halt the machine with the
;                     distinguished, non-erroneous psw HALT.
```

DEFINITION: p-ret-okp (*ins*, *p*) = **t**

DEFINITION:
p-ret-step (*ins*, *p*)
= **if** listp (pop (p-ctrl-stk (*p*)))
then p-state (ret-pc (top (p-ctrl-stk (*p*))),
pop (p-ctrl-stk (*p*)),
p-temp-stk (*p*),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)
else p-halt (*p*, 'halt) **endif**

DEFINITION:
icode-ret (*ins*, *pcn*, *program*)
= list ('(jump_*),
tag ('pc, cons (name (*program*), length (program-body (*program*)))))

```
;  (LOCN var)           The argument var must be a local variable.  Suppose
;                       its value is i.  Then push onto temp-stk the value
;                       of the ith variable bound in the current frame.
;                       i must be less than the number of variables bound.
;                       We enumerate the variables from left to right.
;                       The enumeration is 0 based.  Thus, if a subroutine
;                       has formals X, Y, and Z, and declares temps
;                       A, B, and C, in that order, X is the 0th local,
;                       Y the 1st, Z the 2nd, A the 3rd, etc.
```

DEFINITION:
p-locn-okp (*ins*, *p*)
= (p-objectp-type ('nat, local-var-value (cadr (*ins*), p-ctrl-stk (*p*)), *p*)
∧ (untag (local-var-value (cadr (*ins*), p-ctrl-stk (*p*)))
< length (bindings (top (p-ctrl-stk (*p*)))))
∧ (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*)))

```
; Observe that LOCN enumerates the variables in the same order that
; they are offset from csp.  This is also the same order in which
; they are bound in the p-frame built for the subroutine.

; *** 14-12-87/jsm:  The definitions of keyn and var-name were once
; found here but have been deleted because they are no longer used.

; Here then is the step function for LOCN:
```

DEFINITION:
p-locn-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (cdr (get (untag (local-var-value (cadr (*ins*), p-ctrl-stk (*p*))),
bindings (top (p-ctrl-stk (*p*))))),
p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-locn (*ins*, *pcn*, *program*)
= list ('(move_x_*),

```
        tag('nat, offset-from-csp(cadr(ins), program)),
        '(add_x{n}_csp),
        '(move_x_<x{s}>),
        '(add_x{n}_csp),
        '(tpush_<x{s}>))

;push value of vari
```

```
;  (PUSH-CONSTANT c)    Push c onto temp-stk.  c is normally a tagged
;                       constant but we permit two abbreviations.
;                       If c is the atom 'pc, it denotes the pc
;                       of the instruction following this one.
;                       If c is a non-LISTP that is defined in
;                       the current program with a DL, it
;                       denotes the pc of that label.
```

DEFINITION:
p-push-constant-okp (*ins*, *p*)
= (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))

```
;  First I define the ``unabbreviate'' function.  Observe that the
;  abbreviation depends upon the current state, p, since it permits the
;  abbreviation of the current p-pc.
```

DEFINITION:
unabbreviate-constant (*c*, *p*)
= **if** *c* = 'pc **then** add1-p-pc (*p*)
elseif *c* ≃ **nil** **then** pc (*c*, p-current-program (*p*))
else *c* **endif**

DEFINITION:
p-push-constant-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (unabbreviate-constant (cadr (*ins*), *p*), p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-push-constant (*ins*, *pcn*, *program*)
= list ('(tpush_*),
if cadr (*ins*) = 'pc **then** tag ('pc, cons (name (*program*), 1 + *pcn*))
elseif cadr (*ins*) ≃ **nil** **then** pc (cadr (*ins*), *program*)
else cadr (*ins*) **endif**)

```
;  (PUSH-LOCAL var)     Push the value of the local variable var onto
;                       the temp-stk.
```

DEFINITION:
p-push-local-okp (*ins*, *p*)
= (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))

DEFINITION:
p-push-local-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (local-var-value (cadr (*ins*), p-ctrl-stk (*p*)), p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-push-local (*ins*, *pcn*, *program*)
= list ('(move_x_*),
tag ('nat, offset-from-csp (cadr (*ins*), *program*)),
'(add_x{n}_csp),
'(tpush_<x{s}>))

;push value of var onto tsp

```
;  (PUSH-GLOBAL var)    Push the value of the global variable var onto
;                       temp-stk.
```

DEFINITION:
p-push-global-okp (*ins*, *p*)
= (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))

DEFINITION:
p-push-global-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (fetch (tag (`'addr`, cons (cadr (*ins*), `0`)), p-data-segment (*p*)),
p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
icode-push-global (*ins*, *pcn*, *program*)
= list (`'(move_x_*)`,
tag (`'addr`, cons (cadr (*ins*), `0`)),
`'(tpush_<x{a}>)`)

```
;  (PUSH-CTRL-STK-FREE-SIZE)
;                       Push onto temp-stk the size of the
;                       remaining free space in the control
;                       stack.  This is the number of pushes
;                       that can be done on the ctrl-stack
;                       without causing an error.
```

DEFINITION:
p-push-ctrl-stk-free-size-okp (*ins*, *p*)
= (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))

DEFINITION:
p-push-ctrl-stk-free-size-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),

push (tag ('nat,
p-max-ctrl-stk-size (p)
$-$ p-ctrl-stk-size (p-ctrl-stk (p))),
p-temp-stk (p)),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run)

DEFINITION:
icode-push-ctrl-stk-free-size (ins, pcn, $program$)
= '((move_x_*)
(sys-addr (full-ctrl-stk-addr . 0))
(move_x_<x{s}>)
(tpush_csp)
(sub_<tsp>{s}_x{s}))

```
;  (PUSH-TEMP-STK-FREE-SIZE)
;                       Push onto temp-stk the size of the
;                       remaining free space in the temp
;                       stack.  This is the number of pushes
;                       that can be done on the temp-stack
;                       without causing an error -- assuming
;                       that the number pushed here has been
;                       popped off.  For example, if there
;                       is one slot left on the temp stack
;                       when this instruction is executed,
;                       (NAT 1) is pushed and the stack is
;                       then full.  When it is popped, we
;                       know 1 push can be done.
```

DEFINITION:
p-push-temp-stk-free-size-okp (ins, p)
= (length (p-temp-stk (p)) < p-max-temp-stk-size (p))

DEFINITION:
p-push-temp-stk-free-size-step (ins, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag ('nat,
p-max-temp-stk-size (p) $-$ length (p-temp-stk (p))),
p-temp-stk (p)),

p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-push-temp-stk-free-size (*ins*, *pcn*, *program*)
= '((move_x_*)
(sys-addr (full-temp-stk-addr . 0))
(move_x_<x{s}>)
(tpush_tsp)
(sub_<tsp>{s}_x{s}))

```
;  (PUSH-TEMP-STK-INDEX n)

;                       Push onto temp-stk the temp stk index of the
;                       slot n below the current top.  n must be a
;                       natural number less than the length of the
;                       stack.  The object pushed is a NAT.
;           Below is a picture of a temp-stk with
;                       of length 6, A being at the top:

;                       temp-stk:   (A B C D E F)
;                       index:       5 4 3 2 1 0
;                       n:           0 1 2 3 4 5

;                       The index returned is suitable for use by
;                       FETCH-TEMP-STK and DEPOSIT-TEMP-STK.
```

DEFINITION:
p-push-temp-stk-index-okp (*ins*, *p*)
= ((length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))
∧ (cadr (*ins*) < length (p-temp-stk (*p*))))

DEFINITION:
p-push-temp-stk-index-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag ('nat, (length (p-temp-stk (*p*)) − cadr (*ins*)) − 1),
p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),

p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

```
; Note:  The assembly code below uses sub_<z>_x{s}_y{s} but is not actually
; sensitive to the z-flg.  I use that instruction simply because it
; was in the i machine already.
```

DEFINITION:
icode-push-temp-stk-index (*ins*, *pcn*, *program*)
= list ('(move_y_tsp),
'(move_x_*),
'(sys-addr (empty-temp-stk-addr . 0)),
'(move_x_<x{s}>),
'(sub_<z>_x{s}_y{s}),
'(tpush_x),
'(move_x_*),
tag ('nat, 1 + cadr (*ins*)),
'(sub_<tsp>{n}_x{n}))

```
;  (JUMP-IF-TEMP-STK-FULL lab)
;                       If the temp-stk is full, jump to lab.
```

DEFINITION: p-jump-if-temp-stk-full-okp (*ins*, *p*) = **t**

DEFINITION:
p-jump-if-temp-stk-full-step (*ins*, *p*)
= p-state (**if** length (p-temp-stk (*p*)) = p-max-temp-stk-size (*p*)
then pc (cadr (*ins*), p-current-program (*p*))
else add1-p-pc (*p*) **endif**,
p-ctrl-stk (*p*),
p-temp-stk (*p*),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-jump-if-temp-stk-full (*ins*, *pcn*, *program*)
= list ('(move_x_tsp),

```
        '(move_y_*),
        '(sys-addr (full-temp-stk-addr . 0)),
        '(move_y_<y{s}>),
        '(sub_<z>_x{s}_y{s}),
        '(move_x_*),
        pc (cadr (ins), program),
        '(jump-z_x))

;  (JUMP-IF-TEMP-STK-EMPTY lab)
;                       If the temp-stk is empty, jump to lab.
```

DEFINITION: p-jump-if-temp-stk-empty-okp (*ins*, *p*) = **t**

DEFINITION:
p-jump-if-temp-stk-empty-step (*ins*, *p*)
= p-state (**if** length (p-temp-stk (*p*)) ≃ 0
then pc (cadr (*ins*), p-current-program (*p*))
else add1-p-pc (*p*) **endif**,
p-ctrl-stk (*p*),
p-temp-stk (*p*),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-jump-if-temp-stk-empty (*ins*, *pcn*, *program*)
= list ('(move_y_tsp),
'(move_x_*),
'(sys-addr (empty-temp-stk-addr . 0)),
'(move_x_<x{s}>),
'(sub_<z>_x{s}_y{s}),
'(move_x_*),
pc (cadr (*ins*), *program*),
'(jump-z_x))

```
;  (POP)                Pop top of temp-stk and discard
```

DEFINITION: p-pop-okp (*ins*, *p*) = listp (p-temp-stk (*p*))

DEFINITION:

p-pop-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION: icode-pop (*ins*, *pcn*, *program*) = '((tpop_x))

```
;  (POP* n)                 Pop temp-stk n times.
```

DEFINITION:
p-pop*-okp (*ins*, *p*) = (length (p-temp-stk (*p*)) $\not<$ cadr (*ins*))

DEFINITION:
p-pop*-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
popn (cadr (*ins*), p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-pop* (*ins*, *pcn*, *program*)
= list ('(add_tsp_*{n}), tag ('nat, cadr (*ins*)))

```
;  (POPN)                Pop temp-stk once to obtain n,
;                        a NAT.  Then pop and discard n
;                        things from temp-stk.  n must be
;                        less than or equal to the length
;                        of the stack (after it is popped
;                        off).
```

DEFINITION:
p-popn-okp (*ins*, *p*)

```
=   (listp (p-temp-stk (p))
    ∧   p-objectp-type ('nat, top (p-temp-stk (p)), p)
    ∧   (length (p-temp-stk (p)) ≮ (1 + untag (top (p-temp-stk (p))))))
```

DEFINITION:
p-popn-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
popn (untag (top (p-temp-stk (*p*))), pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-popn (*ins*, *pcn*, *program*) = '((tpop_x) (add_tsp_x{n}))

```
;  (POP-LOCAL var)      Pop top of temp-stk and put the value into the local
;                       variable var.
```

DEFINITION: p-pop-local-okp (*ins*, *p*) = listp (p-temp-stk (*p*))

DEFINITION:
p-pop-local-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
set-local-var-value (top (p-temp-stk (*p*)),
cadr (*ins*),
p-ctrl-stk (*p*)),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-pop-local (*ins*, *pcn*, *program*)
= list ('(move_x_*),
tag ('nat, offset-from-csp (cadr (*ins*), *program*)),
'(add_x{n}_csp),
'(tpop_<x{s}>))

```
;  (POP-GLOBAL var)      Pop top of temp-stk and put the value into the global
;                        variable var.
```

DEFINITION: p-pop-global-okp (*ins*, *p*) = listp (p-temp-stk (*p*))

DEFINITION:
p-pop-global-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
deposit (top (p-temp-stk (*p*)),
tag (`'addr`, cons (cadr (*ins*), `0`)),
p-data-segment (*p*)),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
icode-pop-global (*ins*, *pcn*, *program*)
= list (`'(move_x_*)`,
tag (`'addr`, cons (cadr (*ins*), `0`)),
`'(tpop_<x{a}>)`)

```
;  (POP-LOCN var)        Pop top of temp-stk and put the value into the ith
;                        bound var in the current frame, where i is the value
;                        of the bound variable var.
```

DEFINITION:
p-pop-locn-okp (*ins*, *p*)
= (p-objectp-type (`'nat`, local-var-value (cadr (*ins*), p-ctrl-stk (*p*)), *p*)
∧ (untag (local-var-value (cadr (*ins*), p-ctrl-stk (*p*)))
< length (bindings (top (p-ctrl-stk (*p*)))))
∧ listp (p-temp-stk (*p*)))

```
; *** 14-12-87/mk:  In the original definition we set the value
; by going through the name of the ith variable.  Duplicate
; names caused problems.  We now set the value via position.
; Two new functions are defined.
```

DEFINITION:
put-value-indirect (*val*, *n*, *lst*)
= **if** listp (*lst*)
then if *n* ≃ 0 **then** cons (cons (caar (*lst*), *val*), cdr (*lst*))
else cons (car (*lst*), put-value-indirect (*val*, *n* − 1, cdr (*lst*))) **endif**
else *lst* **endif**

DEFINITION:
set-local-var-indirect (*val*, *index*, *ctrl-stk*)
= push (p-frame (put-value-indirect (*val*, *index*, bindings (top (*ctrl-stk*))),
ret-pc (top (*ctrl-stk*))),
pop (*ctrl-stk*))

DEFINITION:
p-pop-locn-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
set-local-var-indirect (top (p-temp-stk (*p*)),
untag (local-var-value (cadr (*ins*),
p-ctrl-stk (*p*))),
p-ctrl-stk (*p*)),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-pop-locn (*ins*, *pcn*, *program*)
= list ('(move_x_*),
tag ('nat, offset-from-csp (cadr (*ins*), *program*)),
'(add_x{n}_csp),
'(move_x_<x{s}>),
'(add_x{n}_csp),
'(tpop_<x{s}>))

```
;  (POP-CALL)           Pop temp-stk to obtain a subroutine name and
;                       call that subroutine.
```

DEFINITION:
p-pop-call-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type ('subr, top (p-temp-stk (*p*)), *p*)

```
    ∧   p-call-okp (list ('call, untag (top (p-temp-stk (p)))),
                    p-state (p-pc (p),
                             p-ctrl-stk (p),
                             pop (p-temp-stk (p)),
                             p-prog-segment (p),
                             p-data-segment (p),
                             p-max-ctrl-stk-size (p),
                             p-max-temp-stk-size (p),
                             p-word-size (p),
                             'run)))
```

DEFINITION:

```
p-pop-call-step (ins, p)
=   p-call-step (list ('call, untag (top (p-temp-stk (p)))),
                 p-state (p-pc (p),
                          p-ctrl-stk (p),
                          pop (p-temp-stk (p)),
                          p-prog-segment (p),
                          p-data-segment (p),
                          p-max-ctrl-stk-size (p),
                          p-max-temp-stk-size (p),
                          p-word-size (p),
                          'run))
```

DEFINITION:

```
icode-pop-call (ins, pcn, program)
=   list ('(tpop_x),
          '(cpush_*),
          tag ('pc, cons (name (program), 1 + pcn)),
          '(jump_x{subr}))

;   (FETCH-TEMP-STK)    Pop top of temp-stk.  The result must
;                       be an index into the temp-stk, that is,
;                       a NAT less than the length of temp-stk.
;                       Push onto the temp-stk the contents of
;                       the indexed cell of temp-stk.
```

DEFINITION:

```
p-fetch-temp-stk-okp (ins, p)
=   (listp (p-temp-stk (p))
    ∧   p-objectp-type ('nat, top (p-temp-stk (p)), p)
    ∧   (untag (top (p-temp-stk (p))) < length (p-temp-stk (p))))
```

DEFINITION:

```
p-fetch-temp-stk-step (ins, p)
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (rget (untag (top (p-temp-stk (p))), p-temp-stk (p)),
                   pop (p-temp-stk (p))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run)
```

DEFINITION:

```
icode-fetch-temp-stk (ins, pcn, program)
=   '((tpop_y)
      (incr_y_y{n})
      (move_x_*)
      (sys-addr (empty-temp-stk-addr . 0))
      (move_x_<x{s}>)
      (sub_x{s}_y{n})
      (tpush_<x{s}>))

;   (DEPOSIT-TEMP-STK)  Pop top and top1 off of temp-stk.  top must be
;                       an index into temp-stk.  Deposit top1 into the indexed
;                       cell of temp-stk.
```

DEFINITION:

```
p-deposit-temp-stk-okp (ins, p)
=   (listp (p-temp-stk (p))
    ∧   listp (pop (p-temp-stk (p)))
    ∧   p-objectp-type ('nat, top (p-temp-stk (p)), p)
    ∧   (untag (top (p-temp-stk (p)))
          <   length (pop (pop (p-temp-stk (p))))))
```

DEFINITION:

```
p-deposit-temp-stk-step (ins, p)
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             rput (top1 (p-temp-stk (p)),
                   untag (top (p-temp-stk (p))),
                   pop (pop (p-temp-stk (p)))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
```

p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-deposit-temp-stk (*ins*, *pcn*, *program*)
= '((tpop_y)
(incr_y_y{n})
(move_x_*)
(sys-addr (empty-temp-stk-addr . 0))
(move_x_<x{s}>)
(sub_x{s}_y{n})
(tpop_<x{s}>))

```
;  (JUMP lab)           Jump to the location named by lab in the current
;                       program.
```

DEFINITION: p-jump-okp (*ins*, *p*) = **t**

DEFINITION:
p-jump-step (*ins*, *p*)
= p-state (pc (cadr (*ins*)), p-current-program (*p*)),
p-ctrl-stk (*p*),
p-temp-stk (*p*),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-jump (*ins*, *pcn*, *program*) = list ('(jump_*), pc (cadr (*ins*), *program*))

```
;   (JUMP-CASE lab0 lab1 ... labn)
;                       Pop top of temp-stk.  It must be a some nat, i.
;                       Jump to labi.
```

DEFINITION:
p-jump-case-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type ('nat, top (p-temp-stk (*p*)), *p*)
∧ (untag (top (p-temp-stk (*p*))) < length (cdr (*ins*))))

DEFINITION:
p-jump-case-step (*ins*, *p*)
= p-state (pc (get (untag (top (p-temp-stk (*p*))), cdr (*ins*)),
p-current-program (*p*)),
p-ctrl-stk (*p*),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
jump_*-lst (*lst*, *program*)
= **if** *lst* $\simeq$ **nil then nil**
else cons ('(jump_*),
cons (pc (car (*lst*), *program*),
jump_*-lst (cdr (*lst*), *program*))) **endif**

DEFINITION:
icode-jump-case (*ins*, *pcn*, *program*)
= append ('((tpop_x) (add_x_x{n}) (add_pc_x{n})),
jump_*-lst (cdr (*ins*), *program*))

```
;  (PUSHJ lab)          Push onto temp-stk the pc of the next instruction
;                       and then jump to the location named by lab in the
;                       current program.
```

DEFINITION:
p-pushj-okp (*ins*, *p*) = (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*))

DEFINITION:
p-pushj-step (*ins*, *p*)
= p-state (pc (cadr (*ins*), p-current-program (*p*)),
p-ctrl-stk (*p*),
push (add1-p-pc (*p*), p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-pushj (*ins*, *pcn*, *program*)
= list (`'(tpush_*)`,
tag (`'pc`, cons (name (*program*), 1 + *pcn*)),
`'(jump_*)`,
pc (cadr (*ins*), *program*))

```
;  (POPJ)               Pop the temp-stk to obtain a pc and jump to that
;                       location.  The location must be within the current
;                       program!
```

DEFINITION:
p-popj-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'pc`, top (p-temp-stk (*p*)), *p*)
∧ (area-name (top (p-temp-stk (*p*))) = area-name (p-pc (*p*))))

DEFINITION:
p-popj-step (*ins*, *p*)
= p-state (top (p-temp-stk (*p*)),
p-ctrl-stk (*p*),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION: icode-popj (*ins*, *pcn*, *program*) = `'((tpop_pc))`

```
;  (SET-LOCAL var)      Set the local variable var to the top of the temp-stk
;                       but do not pop the temp-stk.
```

DEFINITION: p-set-local-okp (*ins*, *p*) = listp (p-temp-stk (*p*))

DEFINITION:
p-set-local-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
set-local-var-value (top (p-temp-stk (*p*)),
cadr (*ins*),
p-ctrl-stk (*p*)),
p-temp-stk (*p*),

p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-set-local (*ins*, *pcn*, *program*)
= list ('(move_x_*),
tag ('nat, offset-from-csp (cadr (*ins*), *program*)),
'(add_x{n}_csp),
'(move_<x{s}>_<tsp>))

```
;  (SET-GLOBAL var)     Set the global variable var to the top of the temp-stk
;                       but do not pop the temp-stk.
```

DEFINITION: p-set-global-okp (*ins*, *p*) = listp (p-temp-stk (*p*))

DEFINITION:
p-set-global-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
p-temp-stk (*p*),
p-prog-segment (*p*),
deposit (top (p-temp-stk (*p*)),
tag ('addr, cons (cadr (*ins*), 0)),
p-data-segment (*p*)),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-set-global (*ins*, *pcn*, *program*)
= list ('(move_x_*),
tag ('addr, cons (cadr (*ins*), 0)),
'(move_<x{a}>_<tsp>))

```
; (TEST-type-AND-JUMP flg lab)
;                       Each instruction in this family pops one item, x, off
;                       temp-stk.  The item must be of the indicated type.
;                       The item is tested as indicated by the flg.  If the
```

```
;                       test is satisfied, we jump to the indicated label, lab.
;                       Otherwise, the next instruction is executed.  The
;                       flags and tests available depend upon the type.
;                       We enumerate them in the documentation for each
;                       test-and-jump instruction.

;  For each member of the family I define p-test-xxx-and-jump-okp,
;  which approves the execution of the instruction.  Then I define
;  p-test-xxx-and-jump-step and
;  icode-test-xxx-and-jump.  Both the -okp and the
;  -step function are defined in terms of more general purpose functions.
```

DEFINITION:
p-test-and-jump-okp (*ins*, *type*, *test*, *p*)
= (listp (p-temp-stk (*p*)) ∧ p-objectp-type (*type*, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-test-and-jump-step (*test*, *lab*, *p*)
= **if** *test*
then p-state (pc (*lab*, p-current-program (*p*)),
p-ctrl-stk (*p*),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)
else p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
pop (p-temp-stk (*p*)),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run) **endif**

```
; (TEST-NAT-AND-JUMP flg lab)

;                           flg                test

;                           ZERO           jump if x is 0
;                       NOT-ZERO           jump if x is not 0
```

DEFINITION:
p-test-natp (*flg*, *x*)
= **case on** *flg*:
  case = *zero*
  then *x* = 0
  otherwise *x* ≠ 0 **endcase**

DEFINITION:
p-test-nat-and-jump-okp (*ins*, *p*)
= p-test-and-jump-okp (*ins*,
                       'nat,
                       p-test-natp (cadr (*ins*), untag (top (p-temp-stk (*p*)))),
                       p)

DEFINITION:
p-test-nat-and-jump-step (*ins*, *p*)
= p-test-and-jump-step (p-test-natp (cadr (*ins*), untag (top (p-temp-stk (*p*)))),
                        caddr (*ins*),
                        p)

DEFINITION:
icode-test-nat-and-jump (*ins*, *pcn*, *program*)
= **case on** cadr (*ins*):
  case = *zero*
  then list ('(tpop{n}_<z>_y),
                 '(move_x_*),
                 pc (caddr (*ins*), *program*),
                 '(jump-z_x))
  otherwise list ('(tpop{n}_<z>_y),
                      '(move_x_*),
                      pc (caddr (*ins*), *program*),
                      '(jump-nz_x)) **endcase**

```
; (TEST-INT-AND-JUMP flg lab)
;                          flg                test
;                          ZERO          jump if x is 0
;                       NOT-ZERO         jump if x is not 0
;                          NEG           jump if x is negative
;                       NOT-NEG          jump if x is not negative
;                          POS           jump if x is positive
;                       NOT-POS          jump if x is not positive
```

DEFINITION:
p-test-intp (*flg*, x)
= **case on** *flg*:
case = *zero*
then $x = \texttt{0}$
case = *not-zero*
then $x \neq \texttt{0}$
case = *neg*
then negativep (x)
case = *not-neg*
then $\neg$ negativep (x)
case = *pos*
then $(x \in \mathbf{N}) \wedge (x \neq \texttt{0})$
otherwise $(x = \texttt{0}) \vee$ negativep (x) **endcase**

DEFINITION:
p-test-int-and-jump-okp (*ins*, p)
= p-test-and-jump-okp (*ins*,
'int,
p-test-intp (cadr (*ins*), untag (top (p-temp-stk (p)))),
p)

DEFINITION:
p-test-int-and-jump-step (*ins*, p)
= p-test-and-jump-step (p-test-intp (cadr (*ins*), untag (top (p-temp-stk (p)))),
caddr (*ins*),
p)

DEFINITION:
icode-test-int-and-jump (*ins*, *pcn*, *program*)
= **case on** cadr (*ins*):
case = *zero*
then list ('(tpop{i}_<zn>_y),
'(move_x_*),
pc (caddr (*ins*), *program*),
'(jump-z_x))
case = *not-zero*
then list ('(tpop{i}_<zn>_y),
'(move_x_*),
pc (caddr (*ins*), *program*),
'(jump-nz_x))
case = *neg*
then list ('(tpop{i}_<zn>_y),
'(move_x_*),
pc (caddr (*ins*), *program*),

```
            '(jump-n_x))
    case = not-neg
     then list('(tpop{i}_<zn>_y),
            '(move_x_*),
            pc(caddr(ins), program),
            '(jump-nn_x))
    case = pos
     then list('(tpop{i}_<zn>_y),
            '(move_x_*),
            tag('pc, cons(name(program), 1 + pcn)),
            '(jump-n_x),
            '(jump-z_x),
            '(jump_*),
            pc(caddr(ins), program))
    otherwise list('(tpop{i}_<zn>_y),
                '(move_x_*),
                pc(caddr(ins), program),
                '(jump-n_x),
                '(jump-z_x)) endcase

; (TEST-BOOL-AND-JUMP flg lab)
;                          flg               test
;                          T            jump if x is T
;                          F            jump if x is F
```

DEFINITION:
p-test-boolp(*flg*, *x*)
= **case on** *flg*:
case = t
then *x* = 't
otherwise *x* = 'f **endcase**

DEFINITION:
p-test-bool-and-jump-okp(*ins*, *p*)
= p-test-and-jump-okp(*ins*,
'bool,
p-test-boolp(cadr(*ins*), untag(top(p-temp-stk(*p*)))),
p)

DEFINITION:
p-test-bool-and-jump-step(*ins*, *p*)
= p-test-and-jump-step(p-test-boolp(cadr(*ins*), untag(top(p-temp-stk(*p*)))),
caddr(*ins*),
p)

DEFINITION:
icode-test-bool-and-jump (*ins*, *pcn*, *program*)
= **case on** cadr (*ins*):
case = t
then list (`'(tpop{b}_<z>_y)`,
`'(move_x_*)`,
pc (caddr (*ins*), *program*),
`'(jump-nz_x)`)
otherwise list (`'(tpop{b}_<z>_y)`,
`'(move_x_*)`,
pc (caddr (*ins*), *program*),
`'(jump-z_x)`) **endcase**

```
; (TEST-BITV-AND-JUMP flg lab)
;                          flg                test
;                          ALL-ZERO      jump if x contains all 0's
;                          NOT-ALL-ZERO  jump if x contains a 1
```

DEFINITION:
p-test-bitvp (*flg*, *x*)
= **case on** *flg*:
case = *all-zero*
then all-zero-bitvp (*x*)
otherwise ¬ all-zero-bitvp (*x*) **endcase**

DEFINITION:
p-test-bitv-and-jump-okp (*ins*, *p*)
= p-test-and-jump-okp (*ins*,
`'bitv`,
p-test-bitvp (cadr (*ins*), untag (top (p-temp-stk (*p*)))),
p)

DEFINITION:
p-test-bitv-and-jump-step (*ins*, *p*)
= p-test-and-jump-step (p-test-bitvp (cadr (*ins*), untag (top (p-temp-stk (*p*)))),
caddr (*ins*),
p)

DEFINITION:
icode-test-bitv-and-jump (*ins*, *pcn*, *program*)
= **case on** cadr (*ins*):
case = *all-zero*
then list (`'(tpop{v}_<z>_y)`,
`'(move_x_*)`,

pc (caddr (*ins*), *program*),
'(jump-z_x))
otherwise list ('(tpop{v}_<z>_y),
'(move_x_*),
pc (caddr (*ins*), *program*),
'(jump-nz_x)) **endcase**

```
;   (NO-OP)             Do nothing.  Advance the pc to the next instruction.
```

DEFINITION: p-no-op-okp (*ins*, *p*) = **t**

DEFINITION:
p-no-op-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
p-temp-stk (*p*),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-no-op (*ins*, *pcn*, *program*) = list ('(move_x_x))

```
; Note:  One might ask why no-ops generate any instructions at all.
; The reason is that it was the easiest way to guarantee that I kept
; proper track of high level pcs.  I have a guarantee that every pc
; at the high level corresponds to a label at the i level.  I have
; adopted the (DL pc & ins) hack to associate labels with instructions.
; I can't attach multiple labels to the same point.  So I see no alternative
; to generating a place holder instruction for the no-op and attaching
; a label to it.  Of course, I could change the DL conventions so that
; if no instruction was present we migrated the label down to the next
; DL and used (DL (pc1 pc2 ...) ins).  But even that loses
; if a no-op occurs as the last instruction.  This seems simplest.
```

```
;   (ADD-ADDR) Pop top and top1 from temp-stk.  Top1 must
;                        be an addr, top must be a nat.  Increment
;                        the addr by nat and if that is a legal address
;                        push it onto temp-stk.
```

DEFINITION:
p-add-addr-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ p-objectp-type (`'nat`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'addr`, top1 (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'addr`,
add-addr (top1 (p-temp-stk (*p*)),
untag (top (p-temp-stk (*p*)))),
p))

DEFINITION:
p-add-addr-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (add-addr (top1 (p-temp-stk (*p*)), untag (top (p-temp-stk (*p*)))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
icode-add-addr (*ins*, *pcn*, *program*)
= `'((tpop_x) (add_<tsp>{a}_x{n}))`

```
;   (SUB-ADDR)          Pop top and top1 off of temp-stk.
;                       Top must be a nat and top1 must be
;                       an addr.  Decrement the offset of the
;                       addr by nat and push the result.
```

DEFINITION:
p-sub-addr-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))

$\wedge$ p-objectp-type (`'nat`, top (p-temp-stk (p)), p)
$\wedge$ p-objectp-type (`'addr`, top1 (p-temp-stk (p)), p)
$\wedge$ (offset (top1 (p-temp-stk (p))) $\not<$ untag (top (p-temp-stk (p)))))

DEFINITION:
p-sub-addr-step (*ins*, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (sub-addr (top1 (p-temp-stk (p)), untag (top (p-temp-stk (p)))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
icode-sub-addr (*ins*, *pcn*, *program*)
= `'((tpop_x) (sub_<tsp>{a}_x{n}))`

```
;   (EQ)                Pop top and top1 off of temp-stk.  If they
;                       are both of the same type, push T or F according
;                       to whether they are equal.
```

DEFINITION:
p-eq-okp (*ins*, p)
= (listp (p-temp-stk (p))
$\wedge$ listp (pop (p-temp-stk (p)))
$\wedge$ (type (top (p-temp-stk (p))) = type (top1 (p-temp-stk (p)))))

DEFINITION:
p-eq-step (*ins*, p)
= p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (bool (untag (top1 (p-temp-stk (p)))
= untag (top (p-temp-stk (p)))),
pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
`'run`)

DEFINITION:
icode-eq (*ins*, *pcn*, *program*)
= '((tpop_x)
(xor_<z>_<tsp>_x)
(xor_<tsp>_<tsp>)
(move-z_<tsp>_*)
(bool t))

```
;   (LT-ADDR)           Pop top and top1 off of temp-stk,
;                       and if both are addresses into the same area,
;                       push T if top1 < top and push F otherwise.
```

DEFINITION:
p-lt-addr-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ p-objectp-type ('addr, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type ('addr, top1 (p-temp-stk (*p*)), *p*)
∧ (area-name (top (p-temp-stk (*p*)))
= area-name (top1 (p-temp-stk (*p*)))))

DEFINITION:
p-lt-addr-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (bool (offset (top1 (p-temp-stk (*p*)))
< offset (top (p-temp-stk (*p*)))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-lt-addr (*ins*, *pcn*, *program*)
= '((tpop_x)
(sub_<c>_<tsp>{a}_x{a})
(xor_<tsp>_<tsp>)
(move-c_<tsp>_*)
(bool t))

```
;   (FETCH)             Pop top of temp-stk.  The result must
```

```
;                       be an address.  Push the contents of that
;                       address onto temp-stk.
```

DEFINITION:
p-fetch-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('addr, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-fetch-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
        p-ctrl-stk (*p*),
        push (fetch (top (p-temp-stk (*p*)), p-data-segment (*p*)),
            pop (p-temp-stk (*p*))),
        p-prog-segment (*p*),
        p-data-segment (*p*),
        p-max-ctrl-stk-size (*p*),
        p-max-temp-stk-size (*p*),
        p-word-size (*p*),
        'run)

DEFINITION:
icode-fetch (*ins*, *pcn*, *program*) = '((tpop_x) (tpush_<x{a}>))

```
;   (DEPOSIT)           Pop top and top1 off of temp-stk.  top must be
;                       an address.  Deposit top1 into the address.
```

DEFINITION:
p-deposit-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))
  ∧ p-objectp-type ('addr, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-deposit-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
        p-ctrl-stk (*p*),
        pop (pop (p-temp-stk (*p*))),
        p-prog-segment (*p*),
        deposit (top1 (p-temp-stk (*p*)),
            top (p-temp-stk (*p*)),
            p-data-segment (*p*)),
        p-max-ctrl-stk-size (*p*),

p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
icode-deposit (*ins*, *pcn*, *program*) = `'((tpop_x) (tpop_<x{a}>))`

```
;   (ADD-INT)           Pop top and top1 off of temp-stk.  Both must
;                       be INTs.  Add them together and push the result,
;                       provided it is representable as an INT.
```

DEFINITION:
p-add-int-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ p-objectp-type (`'int`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'int`, top1 (p-temp-stk (*p*)), *p*)
∧ small-integerp (iplus (untag (top1 (p-temp-stk (*p*))),
untag (top (p-temp-stk (*p*)))),
p-word-size (*p*)))

DEFINITION:
p-add-int-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'int`,
iplus (untag (top1 (p-temp-stk (*p*))),
untag (top (p-temp-stk (*p*))))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
icode-add-int (*ins*, *pcn*, *program*)
= `'((tpop_x) (add_<tsp>{i}_x{i}))`

```
;   (ADD-INT-WITH-CARRY)
;                       Pop three things off temp stack, top, top1 and
;                       top2.  Top and top1 must be INTs.  Top2 must
;                       be Boolean.  Add top+top1+top2 -- coercing T to 1
```

```
;                       and F to 0.  Push two results.  First, a boolean
;                       indicating whether top+top1+top2 is not small.
;                       On top of that, push the corrected sum.  The corrected
;                       sum is the sum, if that is a small-integerp; the
;                       sum+2**word-size, if the sum is not small and is
;                       negative; and the sum-2**word-size, if the sum is
;                       not small and not negative.
```

DEFINITION:
p-add-int-with-carry-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))
  ∧ listp (pop (pop (p-temp-stk (*p*))))
  ∧ p-objectp-type (`'int`, top (p-temp-stk (*p*)), *p*)
  ∧ p-objectp-type (`'int`, top1 (p-temp-stk (*p*)), *p*)
  ∧ p-objectp-type (`'bool`, top2 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-add-int-with-carry-step (*ins*, *p*)
= **let** *sum* **be** iplus (bool-to-nat (untag (top2 (p-temp-stk (*p*)))),
                    iplus (untag (top1 (p-temp-stk (*p*))),
                           untag (top (p-temp-stk (*p*)))))
  in
  p-state (add1-p-pc (*p*),
           p-ctrl-stk (*p*),
           push (tag (`'int`, fix-small-integer (*sum*, p-word-size (*p*))),
                 push (bool (¬ small-integerp (*sum*, p-word-size (*p*))),
                       pop (pop (pop (p-temp-stk (*p*)))))),
           p-prog-segment (*p*),
           p-data-segment (*p*),
           p-max-ctrl-stk-size (*p*),
           p-max-temp-stk-size (*p*),
           p-word-size (*p*),
           `'run`) **endlet**

DEFINITION:
icode-add-int-with-carry (*ins*, *pcn*, *program*)
= `'((tpop_x)`
  `(tpop_y)`
  `(asr_<c>_<tsp>_<tsp>{b})`
  `(addc_<v>_x{i}_y{i})`
  `(move-v_<tsp>_*)`
  `(bool t)`
  `(tpush_x))`

```
;   (ADD1-INT)          Pop top of temp-stk.  The result must be an INT.
;                       Increment it by one and push the result, provided
;                       it is representable as an INT.
```

DEFINITION:
p-add1-int-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('int, top (p-temp-stk (*p*)), *p*)
  ∧ small-integerp (iplus (1, untag (top (p-temp-stk (*p*)))),
                    p-word-size (*p*)))

DEFINITION:
p-add1-int-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
           p-ctrl-stk (*p*),
           push (tag ('int, iplus (1, untag (top (p-temp-stk (*p*))))),
                 pop (p-temp-stk (*p*))),
           p-prog-segment (*p*),
           p-data-segment (*p*),
           p-max-ctrl-stk-size (*p*),
           p-max-temp-stk-size (*p*),
           p-word-size (*p*),
           'run)

DEFINITION:
icode-add1-int (*ins*, *pcn*, *program*) = '((incr_<tsp>_<tsp>{i}))

```
;   (SUB-INT)           Pop top and top1 off of temp-stk.  Both must be
;                       INTs.  Subtract top from top1, i.e., form top1-top,
;                       and push the result provided it is representable
;                       as an INT.
```

DEFINITION:
p-sub-int-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))
  ∧ p-objectp-type ('int, top (p-temp-stk (*p*)), *p*)
  ∧ p-objectp-type ('int, top1 (p-temp-stk (*p*)), *p*)
  ∧ small-integerp (idifference (untag (top1 (p-temp-stk (*p*))),
                                 untag (top (p-temp-stk (*p*)))),
                    p-word-size (*p*)))

DEFINITION:

p-sub-int-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'int`,
idifference (untag (top1 (p-temp-stk (*p*))),
untag (top (p-temp-stk (*p*))))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

Definition:
icode-sub-int (*ins*, *pcn*, *program*)
= `'((tpop_x) (sub_<tsp>{i}_x{i}))`

```
;   (SUB-INT-WITH-CARRY)
;                       Pop three things off temp stack, top, top1 and
;                       top2.  Top and top1 must be INTs.  Top2 must
;                       be Boolean.  Form top1-(top+top2) -- coercing T to 1
;                       and F to 0.  Push two results.  First, a boolean
;                       indicating whether top1-(top+top2) is not small.
;                       On top of that, push the corrected diff. The corrected
;                       diff is the diff, if that is a small-integerp; the
;                       diff+2**word-size, if the diff is not small and is
;                       negative; and the diff-2**word-size, if the diff is
;                       not small and not negative.
```

Definition:
p-sub-int-with-carry-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ listp (pop (pop (p-temp-stk (*p*))))
∧ p-objectp-type (`'int`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'int`, top1 (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'bool`, top2 (p-temp-stk (*p*)), *p*))

Definition:
p-sub-int-with-carry-step (*ins*, *p*)
= **let** *diff* **be** idifference (untag (top1 (p-temp-stk (*p*))),
iplus (untag (top (p-temp-stk (*p*))),
bool-to-nat (untag (top2 (p-temp-stk (*p*)))))))

in
p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'int`, fix-small-integer (*diff*, p-word-size (*p*))),
push (bool (¬ small-integerp (*diff*, p-word-size (*p*))),
pop (pop (pop (p-temp-stk (*p*)))))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`) **endlet**

DEFINITION:
icode-sub-int-with-carry (*ins*, *pcn*, *program*)
= `'((tpop_y)`
`(tpop_x)`
`(asr_<c>_<tsp>_<tsp>{b})`
`(subb_<v>_x{i}_y{i})`
`(move-v_<tsp>_*)`
`(bool t)`
`(tpush_x))`

```
;   (SUB1-INT)          Pop top of temp-stk.  It must be an INT.
;                       Decrement it by one and push the result,
;                       provided it is representable as an INT.
```

DEFINITION:
p-sub1-int-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'int`, top (p-temp-stk (*p*)), *p*)
∧ small-integerp (idifference (untag (top (p-temp-stk (*p*))), `1`),
p-word-size (*p*)))

DEFINITION:
p-sub1-int-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'int`, idifference (untag (top (p-temp-stk (*p*))), `1`)),
pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),

p-word-size (*p*),
'run)

DEFINITION:
icode-sub1-int (*ins*, *pcn*, *program*) = '((decr_<tsp>_<tsp>{i}))

```
;   (NEG-INT)           Pop top of temp-stk.  It must be an INT.
;                       Negate it and push the result provided it is
;                       representable as an INT.
```

DEFINITION:
p-neg-int-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('int, top (p-temp-stk (*p*)), *p*)
  ∧ small-integerp (inegate (untag (top (p-temp-stk (*p*)))),
                    p-word-size (*p*)))

DEFINITION:
p-neg-int-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
           p-ctrl-stk (*p*),
           push (tag ('int, inegate (untag (top (p-temp-stk (*p*))))),
                 pop (p-temp-stk (*p*))),
           p-prog-segment (*p*),
           p-data-segment (*p*),
           p-max-ctrl-stk-size (*p*),
           p-max-temp-stk-size (*p*),
           p-word-size (*p*),
           'run)

DEFINITION:
icode-neg-int (*ins*, *pcn*, *program*) = '((neg_<tsp>_<tsp>{i}))

```
;   (LT-INT)            Pop top and top1 off of temp-stk, and push T or F
;                       according to whether top1 < top.
```

DEFINITION:
p-lt-int-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))
  ∧ p-objectp-type ('int, top (p-temp-stk (*p*)), *p*)
  ∧ p-objectp-type ('int, top1 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-lt-int-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (bool (ilessp (untag (top1 (p-temp-stk (*p*))),
untag (top (p-temp-stk (*p*))))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

```
; Let n-flg and v-flg be the values of the n-flg and v-flg after computing x-y.
; It is a theorem that x<y iff (xor n-flg v-flg).
```

DEFINITION:
icode-lt-int (*ins*, *pcn*, *program*)
= '((tpop_x)
(sub_<nv>_<tsp>{i}_x{i})
(move_<tsp>_*)
(bool f)
(move-v_<tsp>_*)
(bool t)
(move_x_*)
(bool f)
(move-n_x_*)
(bool t)
(xor_<tsp>{b}_x{b}))

```
; The code above requires 7 clock ticks.  I have written one requiring
; 5 or 6 ticks, depending on whether n-flg was T or F.  But that code
; required an interior label and a move-n_pc_<pc+1> instruction where the
; label at pc+1 is one less than where I need to go (because the post
; increment adds to it).  The interior label would destroy the current
; r-i proof structure, but could probably be patched around.
```

```
;   (INT-TO-NAT)        Pop top off of temp-stk.  Top should
;                       be of type INT and be nonnegative.
;                       Push the NAT corresponding to top.
```

DEFINITION:
p-int-to-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('int, top (p-temp-stk (*p*)), *p*)
  ∧ (¬ negativep (untag (top (p-temp-stk (*p*))))))

DEFINITION:
p-int-to-nat-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
           p-ctrl-stk (*p*),
           push (tag ('nat, untag (top (p-temp-stk (*p*)))), pop (p-temp-stk (*p*))),
           p-prog-segment (*p*),
           p-data-segment (*p*),
           p-max-ctrl-stk-size (*p*),
           p-max-temp-stk-size (*p*),
           p-word-size (*p*),
           'run)

DEFINITION:
icode-int-to-nat (*ins*, *pcn*, *program*) = '((int-to-nat))

```
;   (ADD-NAT)           Pop top and top1 off of temp-stk.
;                       Both top and top1 must be NATs.  Push
;                       their NAT sum.

; *** 15-12-87/jsm:  This is actually a new Piton instruction with
; the same name as an old one.  The old ADD-NAT has been renamed
; ADD-NAT-WITH-CARRY.
```

DEFINITION:
p-add-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))
  ∧ p-objectp-type ('nat, top (p-temp-stk (*p*)), *p*)
  ∧ p-objectp-type ('nat, top1 (p-temp-stk (*p*)), *p*)
  ∧ small-naturalp (untag (top1 (p-temp-stk (*p*)))
                    + untag (top (p-temp-stk (*p*))),
                    p-word-size (*p*)))

DEFINITION:
p-add-nat-step (*ins*, *p*)
= **let** *sum* **be** untag (top1 (p-temp-stk (*p*)))
+ untag (top (p-temp-stk (*p*)))
in
p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'nat`, *sum*), pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`) **endlet**

DEFINITION:
icode-add-nat (*ins*, *pcn*, *program*)
= `'((tpop_x) (add_<tsp>{n}_x{n}))`

```
;   (ADD-NAT-WITH-CARRY)
; Pop top, top1 and top2 off of temp-stk.
;                       Both top and top1 must be NATs.  Top2 must be BOOL.
;                       Add all three together -- coercing T to 1 and F to 0.
;                       Let the result be sum.  Push T or F according to
;                       whether sum is not small.  Then push
;                       sum mod 2**word-size.

; *** 15-12-87/jsm:  This used to be called ADD-NAT.  I have renamed it
; so that the name ADD-NAT can be used for the simpler instruction
; that adds two naturals without input or output carry.
```

DEFINITION:
p-add-nat-with-carry-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ listp (pop (pop (p-temp-stk (*p*))))
∧ p-objectp-type (`'nat`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'nat`, top1 (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'bool`, top2 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-add-nat-with-carry-step (*ins*, *p*)
= **let** *sum* **be** bool-to-nat (untag (top2 (p-temp-stk (*p*))))
+ untag (top1 (p-temp-stk (*p*)))

```
                + untag (top (p-temp-stk (p)))
    in
    p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (tag ('nat, fix-small-natural (sum, p-word-size (p))),
                   push (bool (¬ small-naturalp (sum, p-word-size (p))),
                         pop (pop (pop (p-temp-stk (p)))))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run) endlet
```

DEFINITION:

```
icode-add-nat-with-carry (ins, pcn, program)
=  '((tpop_x)
     (tpop_y)
     (asr_<c>_<tsp>_<tsp>{b})
     (addc_<c>_x{n}_y{n})
     (move-c_<tsp>_*)
     (bool t)
     (tpush_x))

;push sum.
```

```
;   (ADD1-NAT)          Pop top off of temp-stk, add 1 to it
;                       and push the result.  The result must
;                       be representable.

; *** 16-12-87/jsm:  New instruction.
```

DEFINITION:
p-add1-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('nat, top (p-temp-stk (*p*)), *p*)
  ∧ small-naturalp (1 + untag (top (p-temp-stk (*p*))), p-word-size (*p*)))

DEFINITION:
p-add1-nat-step (*ins*, *p*)
= **let** *sum* **be** 1 + untag (top (p-temp-stk (*p*)))
  in
  p-state (add1-p-pc (*p*),
           p-ctrl-stk (*p*),
           push (tag ('nat, *sum*), pop (p-temp-stk (*p*))),
           p-prog-segment (*p*),
           p-data-segment (*p*),
           p-max-ctrl-stk-size (*p*),
           p-max-temp-stk-size (*p*),
           p-word-size (*p*),
           'run) **endlet**

DEFINITION:
icode-add1-nat (*ins*, *pcn*, *program*) = '((incr_<tsp>_<tsp>{n}))

```
;   (SUB-NAT)           Pop top, top1 off of temp-stk.
;                       Both top and top1 must be NATs.
;                       If top1 >= top
;                          then push top1-top.
;                       Else, error.

; *** 15-12-87/jsm:  This is a new instruction with an old name.
; The instruction SUB-NAT-WITH-CARRY is now what used to be SUB-NAT.
```

DEFINITION:
p-sub-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))

$\wedge$ p-objectp-type ('nat, top (p-temp-stk (p)), p)
$\wedge$ p-objectp-type ('nat, top1 (p-temp-stk (p)), p)
$\wedge$ (untag (top1 (p-temp-stk (p))) $\not<$ untag (top (p-temp-stk (p)))))

DEFINITION:
p-sub-nat-step (ins, p)
= **let** y **be** untag (top (p-temp-stk (p))),
x **be** untag (top1 (p-temp-stk (p)))
in
p-state (add1-p-pc (p),
p-ctrl-stk (p),
push (tag ('nat, $x - y$), pop (pop (p-temp-stk (p)))),
p-prog-segment (p),
p-data-segment (p),
p-max-ctrl-stk-size (p),
p-max-temp-stk-size (p),
p-word-size (p),
'run) **endlet**

DEFINITION:
icode-sub-nat (ins, pcn, $program$)
= '((tpop_x) (sub_<tsp>{n}_x{n}))

```
;   (SUB-NAT-WITH-CARRY)
; Pop top, top1 and top2 off of temp-stk.
;                        Both top and top1 must be NATs.  Top2 must be BOOL.
;                        If top1 >= top+top2 -- coercing T to 1 and F to 0 --
;                           then push F and top1-(top+top2).
;                        Else, push T and 2**word-size-((top+top2)-top1).

; *** 15-12-87/jsm:  This instruction was renamed from SUB-NAT to free
; that name for the new, simpler instruction.
```

DEFINITION:
p-sub-nat-with-carry-okp (ins, p)
= (listp (p-temp-stk (p))
$\wedge$ listp (pop (p-temp-stk (p)))
$\wedge$ listp (pop (pop (p-temp-stk (p))))
$\wedge$ p-objectp-type ('nat, top (p-temp-stk (p)), p)
$\wedge$ p-objectp-type ('nat, top1 (p-temp-stk (p)), p)
$\wedge$ p-objectp-type ('bool, top2 (p-temp-stk (p)), p))

DEFINITION:
p-sub-nat-with-carry-step (ins, p)

```
=   let y be untag (top (p-temp-stk (p))),
        x be untag (top1 (p-temp-stk (p))),
        c be bool-to-nat (untag (top2 (p-temp-stk (p))))
    in
    p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (tag ('nat,
                        if x < (y + c)
                        then exp (2, p-word-size (p))
                               −   ((y + c) − x)
                        else x − (y + c) endif),
                   push (bool (x < (y + c)),
                         pop (pop (pop (p-temp-stk (p)))))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run) endlet
```

DEFINITION:
icode-sub-nat-with-carry (*ins*, *pcn*, *program*)

```
=   '((tpop_y)
      (tpop_x)
      (asr_<c>_<tsp>_<tsp>{b})
      (subb_<c>_x{n}_y{n})
      (move-c_<tsp>_*)
      (bool t)
      (tpush_x))

;   (SUB1-NAT)          Pop top off of temp-stk.
;                       Top must be a NAT other than
;                       0.  Push top-1.

; *** 16-12-87/jsm:  This is a new instruction.
```

DEFINITION:
p-sub1-nat-okp (*ins*, *p*)

```
=   (listp (p-temp-stk (p))
    ∧   p-objectp-type ('nat, top (p-temp-stk (p)), p)
    ∧   (untag (top (p-temp-stk (p))) ≠ 0))
```

DEFINITION:
p-sub1-nat-step (*ins*, *p*)

```
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (tag ('nat, untag (top (p-temp-stk (p))) − 1),
                   pop (p-temp-stk (p))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run)
```

DEFINITION:
icode-sub1-nat (*ins*, *pcn*, *program*) = '((decr_<tsp>_<tsp>{n}))

```
;   (LT-NAT)            Pop top and top1 off of temp-stk, and push T or F
;                       according to whether top1 < top.
```

DEFINITION:
p-lt-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))
  ∧ p-objectp-type ('nat, top (p-temp-stk (*p*)), *p*)
  ∧ p-objectp-type ('nat, top1 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-lt-nat-step (*ins*, *p*)

```
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (bool (untag (top1 (p-temp-stk (p)))
                           <   untag (top (p-temp-stk (p)))),
                   pop (pop (p-temp-stk (p)))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run)
```

DEFINITION:
icode-lt-nat (*ins*, *pcn*, *program*)

```
=   '((tpop_x)
      (sub_<c>_<tsp>{n}_x{n})
      (xor_<tsp>_<tsp>)
      (move-c_<tsp>_*)
      (bool t))
```

```
;   (MULT2-NAT)         Pop top of temp-stk, multiply by 2 and push
;                       the result, which must be a small-naturalp.
```

DEFINITION:
p-mult2-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('nat, top (p-temp-stk (*p*)), *p*)
  ∧ small-naturalp (2 ∗ untag (top (p-temp-stk (*p*))),
                    p-word-size (*p*)))

DEFINITION:
p-mult2-nat-step (*ins*, *p*)
= **let** *prod* **be** 2 ∗ untag (top (p-temp-stk (*p*)))
  in
  p-state (add1-p-pc (*p*),
           p-ctrl-stk (*p*),
           push (tag ('nat, *prod*), pop (p-temp-stk (*p*))),
           p-prog-segment (*p*),
           p-data-segment (*p*),
           p-max-ctrl-stk-size (*p*),
           p-max-temp-stk-size (*p*),
           p-word-size (*p*),
           'run) **endlet**

DEFINITION:
icode-mult2-nat (*ins*, *pcn*, *program*) = '((add_<tsp>_<tsp>{n}))

```
;   (MULT2-NAT-WITH-CARRY-OUT)
; Pop top of temp-stk, multiply by 2 and push
;                       two things:  whether the result is not smallp and
;                       the result mod 2**word-size.

; *** 15-12-87/jsm:  Renamed from MULT2.
```

DEFINITION:
p-mult2-nat-with-carry-out-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('nat, top (p-temp-stk (*p*)), *p*)
  ∧ (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*)))

DEFINITION:
p-mult2-nat-with-carry-out-step (*ins*, *p*)
= **let** *prod* **be** 2 ∗ untag (top (p-temp-stk (*p*)))

in
p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'nat`, fix-small-natural (*prod*, p-word-size (*p*))),
push (bool (¬ small-naturalp (*prod*, p-word-size (*p*))),
pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`) **endlet**

DEFINITION:
icode-mult2-nat-with-carry-out (*ins*, *pcn*, *program*)
= `'((tpop_x)`
`(add_<c>_x_x{n})`
`(tpush_*)`
`(bool f)`
`(move-c_<tsp>_*)`
`(bool t)`
`(tpush_x))`

```
;   (DIV2-NAT)              Pop top of temp-stk, divide by 2 and push
;                         two things:  the quotient mod 2 and then
;                         the remainder mod 2.
```

DEFINITION:
p-div2-nat-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'nat`, top (p-temp-stk (*p*)), *p*)
∧ (length (p-temp-stk (*p*)) < p-max-temp-stk-size (*p*)))

DEFINITION:
p-div2-nat-step (*ins*, *p*)
= **let** *quo* **be** untag (top (p-temp-stk (*p*))) ÷ 2,
rem **be** untag (top (p-temp-stk (*p*))) **mod** 2
in
p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'nat`, *rem*),
push (tag (`'nat`, *quo*), pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),

p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run) **endlet**

DEFINITION:
icode-div2-nat (*ins*, *pcn*, *program*)
= '((tpop_<c>_x)
(lsr_<c>_x_x{n})
(tpush_x)
(tpush_*)
(nat 0)
(move-c_<tsp>_*)
(nat 1))

```
;   (OR-BITV)           Pop top and top1 off of temp-stk, or them together
;                       and push the result.
```

DEFINITION:
p-or-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ p-objectp-type ('bitv, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type ('bitv, top1 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-or-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag ('bitv,
or-bitv (untag (top1 (p-temp-stk (*p*))),
untag (top (p-temp-stk (*p*))))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-or-bitv (*ins*, *pcn*, *program*) = '((tpop_x) (or_<tsp>{v}_x{v}))

```
;   (AND-BITV)          Pop top and top1 off of temp-stk, and them together
;                       and push the result.
```

DEFINITION:
p-and-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ listp (pop (p-temp-stk (*p*)))
∧ p-objectp-type (`'bitv`, top (p-temp-stk (*p*)), *p*)
∧ p-objectp-type (`'bitv`, top1 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-and-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'bitv`,
and-bitv (untag (top1 (p-temp-stk (*p*))),
untag (top (p-temp-stk (*p*))))),
pop (pop (p-temp-stk (*p*)))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
icode-and-bitv (*ins*, *pcn*, *program*)
= `'((tpop_x) (and_<tsp>{v}_x{v}))`

```
;   (NOT-BITV)          Pop top of temp-stk, not it, and push the result.
```

DEFINITION:
p-not-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'bitv`, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-not-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'bitv`, not-bitv (untag (top (p-temp-stk (*p*))))),
pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

DEFINITION:
icode-not-bitv (*ins*, *pcn*, *program*) = '((not_<tsp>_<tsp>{v}))

```
;   (XOR-BITV)          Pop top and top1 off of temp-stk, xor them,
;                       and push the result.
```

DEFINITION:
p-xor-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ listp (pop (p-temp-stk (*p*)))
  ∧ p-objectp-type ('bitv, top (p-temp-stk (*p*)), *p*)
  ∧ p-objectp-type ('bitv, top1 (p-temp-stk (*p*)), *p*))

DEFINITION:
p-xor-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
  p-ctrl-stk (*p*),
  push (tag ('bitv,
  xor-bitv (untag (top1 (p-temp-stk (*p*))),
  untag (top (p-temp-stk (*p*))))),
  pop (pop (p-temp-stk (*p*)))),
  p-prog-segment (*p*),
  p-data-segment (*p*),
  p-max-ctrl-stk-size (*p*),
  p-max-temp-stk-size (*p*),
  p-word-size (*p*),
  'run)

DEFINITION:
icode-xor-bitv (*ins*, *pcn*, *program*)
= '((tpop_x) (xor_<tsp>{v}_x{v}))

```
;   (RSH-BITV)          Pop top of temp-stk, shift it right 1, bringing
;                       a 0 in at the top, and push the result.
```

DEFINITION:
p-rsh-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
  ∧ p-objectp-type ('bitv, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-rsh-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),

p-ctrl-stk (*p*),
push (tag (`'bitv`, rsh-bitv (untag (top (p-temp-stk (*p*))))),
pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

Definition:
icode-rsh-bitv (*ins*, *pcn*, *program*) = `'((lsr_<tsp>_<tsp>{v}))`

```
;   (LSH-BITV)          Pop top of temp-stk, shift it left 1, bringing
;                       a 0 in at the btm, and push the result.
```

Definition:
p-lsh-bitv-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type (`'bitv`, top (p-temp-stk (*p*)), *p*))

Definition:
p-lsh-bitv-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag (`'bitv`, lsh-bitv (untag (top (p-temp-stk (*p*))))),
pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
`'run`)

Definition:
icode-lsh-bitv (*ins*, *pcn*, *program*) = `'((add_<tsp>_<tsp>{v}))`

```
;   (OR-BOOL)           Pop top and top1 off of temp-stk, or them together
;                       and push the result.
```

Definition:
p-or-bool-okp (*ins*, *p*)

```
=   (listp (p-temp-stk (p))
    ∧   listp (pop (p-temp-stk (p)))
    ∧   p-objectp-type ('bool, top (p-temp-stk (p)), p)
    ∧   p-objectp-type ('bool, top1 (p-temp-stk (p)), p))
```

DEFINITION:

```
p-or-bool-step (ins, p)
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (tag ('bool,
                        or-bool (untag (top1 (p-temp-stk (p))),
                                 untag (top (p-temp-stk (p))))),
                   pop (pop (p-temp-stk (p)))),
             p-prog-segment (p),
             p-data-segment (p),
             p-max-ctrl-stk-size (p),
             p-max-temp-stk-size (p),
             p-word-size (p),
             'run)
```

DEFINITION:

```
icode-or-bool (ins, pcn, program) = '((tpop_x) (or_<tsp>{b}_x{b}))

;   (AND-BOOL)          Pop top and top1 off of temp-stk, and them together
;                       and push the result.
;
```

DEFINITION:

```
p-and-bool-okp (ins, p)
=   (listp (p-temp-stk (p))
    ∧   listp (pop (p-temp-stk (p)))
    ∧   p-objectp-type ('bool, top (p-temp-stk (p)), p)
    ∧   p-objectp-type ('bool, top1 (p-temp-stk (p)), p))
```

DEFINITION:

```
p-and-bool-step (ins, p)
=   p-state (add1-p-pc (p),
             p-ctrl-stk (p),
             push (tag ('bool,
                        and-bool (untag (top1 (p-temp-stk (p))),
                                  untag (top (p-temp-stk (p))))),
                   pop (pop (p-temp-stk (p)))),
             p-prog-segment (p),
             p-data-segment (p),
```

p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-and-bool (*ins*, *pcn*, *program*)
= '((tpop_x) (and_<tsp>{b}_x{b}))

```
;   (NOT-BOOL)          Pop top of temp-stk, not it, and push the result.
```

DEFINITION:
p-not-bool-okp (*ins*, *p*)
= (listp (p-temp-stk (*p*))
∧ p-objectp-type ('bool, top (p-temp-stk (*p*)), *p*))

DEFINITION:
p-not-bool-step (*ins*, *p*)
= p-state (add1-p-pc (*p*),
p-ctrl-stk (*p*),
push (tag ('bool, not-bool (untag (top (p-temp-stk (*p*))))),
pop (p-temp-stk (*p*))),
p-prog-segment (*p*),
p-data-segment (*p*),
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
'run)

DEFINITION:
icode-not-bool (*ins*, *pcn*, *program*)
= '((xor_<tsp>{b}_*{b}) (bool t))

```
;-----------------------------------------------------------------------------
;  The P Machine

; The complete list of PITON opcodes is given below.  To add a new
; instruction, xxx, to the PITON machine it is necessary to

; (a) define the function (p-xxx-okp ins p), which returns T or F
; according to whether it is legal to execute ins in state p, where
; ins is an instance of the new instruction xxx.

; (b) define the function (p-xxx-step ins p), which returns the state
```

```
; produced by stepping forward from state p by the instruction ins.
; Ins is known to be p-xxx-okp.

; (c) add xxx to the list below.

; The prescription above does NOT address how you implement the new
; instruction via the icompiler, etc.
```

DEFINITION:
PITON-OPCODES
= `'(call ret locn push-constant push-local push-global`
`push-ctrl-stk-free-size push-temp-stk-free-size`
`push-temp-stk-index jump-if-temp-stk-full`
`jump-if-temp-stk-empty pop pop* popn pop-local`
`pop-global pop-locn pop-call fetch-temp-stk`
`deposit-temp-stk jump jump-case pushj popj set-local`
`set-global test-nat-and-jump test-int-and-jump`
`test-bool-and-jump test-bitv-and-jump no-op add-addr`
`sub-addr eq lt-addr fetch deposit add-int`
`add-int-with-carry add1-int sub-int sub-int-with-carry`
`sub1-int neg-int lt-int int-to-nat add-nat`
`add-nat-with-carry add1-nat sub-nat sub-nat-with-carry`
`sub1-nat lt-nat mult2-nat mult2-nat-with-carry-out`
`div2-nat or-bitv and-bitv not-bitv xor-bitv rsh-bitv`
`lsh-bitv or-bool and-bool not-bool)`

```
; The following function generates the name of the error
; message produced when an illegal instruction is hit.  The
; first argument is the ``level'' name, e.g., 'P or 'R.
; the (cdr (unpack 'g-instruction)) is used instead of
; (unpack '-instruction) only because '-instruction is an
; illegal evg in the logic, since it does not start with an
; alphabetic character.
```

DEFINITION:
x-y-error-msg $(x,\ y)$
= pack (append (unpack (`'illegal-`),
append (unpack (y), cdr (unpack (`'g-instruction`)))))

```
; The function checks that it is legal to execute ins in state
; p:
```

DEFINITION:
p-ins-okp (*ins*, *p*)
= **case on** car (*ins*):
case = *call*
then p-call-okp (*ins*, *p*)
case = *ret*
then p-ret-okp (*ins*, *p*)
case = *locn*
then p-locn-okp (*ins*, *p*)
case = *push-constant*
then p-push-constant-okp (*ins*, *p*)
case = *push-local*
then p-push-local-okp (*ins*, *p*)
case = *push-global*
then p-push-global-okp (*ins*, *p*)
case = *push-ctrl-stk-free-size*
then p-push-ctrl-stk-free-size-okp (*ins*, *p*)
case = *push-temp-stk-free-size*
then p-push-temp-stk-free-size-okp (*ins*, *p*)
case = *push-temp-stk-index*
then p-push-temp-stk-index-okp (*ins*, *p*)
case = *jump-if-temp-stk-full*
then p-jump-if-temp-stk-full-okp (*ins*, *p*)
case = *jump-if-temp-stk-empty*
then p-jump-if-temp-stk-empty-okp (*ins*, *p*)
case = *pop*
then p-pop-okp (*ins*, *p*)
case = *pop**
then p-pop*-okp (*ins*, *p*)
case = *popn*
then p-popn-okp (*ins*, *p*)
case = *pop-local*
then p-pop-local-okp (*ins*, *p*)
case = *pop-global*
then p-pop-global-okp (*ins*, *p*)
case = *pop-locn*
then p-pop-locn-okp (*ins*, *p*)
case = *pop-call*
then p-pop-call-okp (*ins*, *p*)
case = *fetch-temp-stk*
then p-fetch-temp-stk-okp (*ins*, *p*)
case = *deposit-temp-stk*
then p-deposit-temp-stk-okp (*ins*, *p*)
case = *jump*

then p-jump-okp (*ins*, *p*)
case = *jump-case*
then p-jump-case-okp (*ins*, *p*)
case = *pushj*
then p-pushj-okp (*ins*, *p*)
case = *popj*
then p-popj-okp (*ins*, *p*)
case = *set-local*
then p-set-local-okp (*ins*, *p*)
case = *set-global*
then p-set-global-okp (*ins*, *p*)
case = *test-nat-and-jump*
then p-test-nat-and-jump-okp (*ins*, *p*)
case = *test-int-and-jump*
then p-test-int-and-jump-okp (*ins*, *p*)
case = *test-bool-and-jump*
then p-test-bool-and-jump-okp (*ins*, *p*)
case = *test-bitv-and-jump*
then p-test-bitv-and-jump-okp (*ins*, *p*)
case = *no-op*
then p-no-op-okp (*ins*, *p*)
case = *add-addr*
then p-add-addr-okp (*ins*, *p*)
case = *sub-addr*
then p-sub-addr-okp (*ins*, *p*)
case = *eq*
then p-eq-okp (*ins*, *p*)
case = *lt-addr*
then p-lt-addr-okp (*ins*, *p*)
case = *fetch*
then p-fetch-okp (*ins*, *p*)
case = *deposit*
then p-deposit-okp (*ins*, *p*)
case = *add-int*
then p-add-int-okp (*ins*, *p*)
case = *add-int-with-carry*
then p-add-int-with-carry-okp (*ins*, *p*)
case = *add1-int*
then p-add1-int-okp (*ins*, *p*)
case = *sub-int*
then p-sub-int-okp (*ins*, *p*)
case = *sub-int-with-carry*
then p-sub-int-with-carry-okp (*ins*, *p*)
case = *sub1-int*

then p-sub1-int-okp (*ins*, *p*)
case = *neg-int*
then p-neg-int-okp (*ins*, *p*)
case = *lt-int*
then p-lt-int-okp (*ins*, *p*)
case = *int-to-nat*
then p-int-to-nat-okp (*ins*, *p*)
case = *add-nat*
then p-add-nat-okp (*ins*, *p*)
case = *add-nat-with-carry*
then p-add-nat-with-carry-okp (*ins*, *p*)
case = *add1-nat*
then p-add1-nat-okp (*ins*, *p*)
case = *sub-nat*
then p-sub-nat-okp (*ins*, *p*)
case = *sub-nat-with-carry*
then p-sub-nat-with-carry-okp (*ins*, *p*)
case = *sub1-nat*
then p-sub1-nat-okp (*ins*, *p*)
case = *lt-nat*
then p-lt-nat-okp (*ins*, *p*)
case = *mult2-nat*
then p-mult2-nat-okp (*ins*, *p*)
case = *mult2-nat-with-carry-out*
then p-mult2-nat-with-carry-out-okp (*ins*, *p*)
case = *div2-nat*
then p-div2-nat-okp (*ins*, *p*)
case = *or-bitv*
then p-or-bitv-okp (*ins*, *p*)
case = *and-bitv*
then p-and-bitv-okp (*ins*, *p*)
case = *not-bitv*
then p-not-bitv-okp (*ins*, *p*)
case = *xor-bitv*
then p-xor-bitv-okp (*ins*, *p*)
case = *rsh-bitv*
then p-rsh-bitv-okp (*ins*, *p*)
case = *lsh-bitv*
then p-lsh-bitv-okp (*ins*, *p*)
case = *or-bool*
then p-or-bool-okp (*ins*, *p*)
case = *and-bool*
then p-and-bool-okp (*ins*, *p*)
case = *not-bool*

then p-not-bool-okp (*ins*, *p*)
otherwise f endcase

```
; Note: Initially I used an APPLY$ to define this function but
; that suffers because we don't track the dependency between
; p-ins-okp and p-not-bool-okp, say.  Thus, if the -okp
; function for some instruction is undone and changed,
; theorems about the top-level -okp remain in the data-base,
; inconsistently.  So I have abandoned the APPLY$ and define
; the function with a CASE statement.

; If ins is legal, the following function steps the state forward one.
```

DEFINITION:
p-ins-step (*ins*, *p*)
= **case on** car (*ins*):
case = *call*
then p-call-step (*ins*, *p*)
case = *ret*
then p-ret-step (*ins*, *p*)
case = *locn*
then p-locn-step (*ins*, *p*)
case = *push-constant*
then p-push-constant-step (*ins*, *p*)
case = *push-local*
then p-push-local-step (*ins*, *p*)
case = *push-global*
then p-push-global-step (*ins*, *p*)
case = *push-ctrl-stk-free-size*
then p-push-ctrl-stk-free-size-step (*ins*, *p*)
case = *push-temp-stk-free-size*
then p-push-temp-stk-free-size-step (*ins*, *p*)
case = *push-temp-stk-index*
then p-push-temp-stk-index-step (*ins*, *p*)
case = *jump-if-temp-stk-full*
then p-jump-if-temp-stk-full-step (*ins*, *p*)
case = *jump-if-temp-stk-empty*
then p-jump-if-temp-stk-empty-step (*ins*, *p*)
case = *pop*
then p-pop-step (*ins*, *p*)
case = *pop**
then p-pop*-step (*ins*, *p*)
case = *popn*

then p-popn-step (*ins*, *p*)
case = *pop-local*
then p-pop-local-step (*ins*, *p*)
case = *pop-global*
then p-pop-global-step (*ins*, *p*)
case = *pop-locn*
then p-pop-locn-step (*ins*, *p*)
case = *pop-call*
then p-pop-call-step (*ins*, *p*)
case = *fetch-temp-stk*
then p-fetch-temp-stk-step (*ins*, *p*)
case = *deposit-temp-stk*
then p-deposit-temp-stk-step (*ins*, *p*)
case = *jump*
then p-jump-step (*ins*, *p*)
case = *jump-case*
then p-jump-case-step (*ins*, *p*)
case = *pushj*
then p-pushj-step (*ins*, *p*)
case = *popj*
then p-popj-step (*ins*, *p*)
case = *set-local*
then p-set-local-step (*ins*, *p*)
case = *set-global*
then p-set-global-step (*ins*, *p*)
case = *test-nat-and-jump*
then p-test-nat-and-jump-step (*ins*, *p*)
case = *test-int-and-jump*
then p-test-int-and-jump-step (*ins*, *p*)
case = *test-bool-and-jump*
then p-test-bool-and-jump-step (*ins*, *p*)
case = *test-bitv-and-jump*
then p-test-bitv-and-jump-step (*ins*, *p*)
case = *no-op*
then p-no-op-step (*ins*, *p*)
case = *add-addr*
then p-add-addr-step (*ins*, *p*)
case = *sub-addr*
then p-sub-addr-step (*ins*, *p*)
case = *eq*
then p-eq-step (*ins*, *p*)
case = *lt-addr*
then p-lt-addr-step (*ins*, *p*)
case = *fetch*

then p-fetch-step (*ins*, *p*)
case = *deposit*
then p-deposit-step (*ins*, *p*)
case = *add-int*
then p-add-int-step (*ins*, *p*)
case = *add-int-with-carry*
then p-add-int-with-carry-step (*ins*, *p*)
case = *add1-int*
then p-add1-int-step (*ins*, *p*)
case = *sub-int*
then p-sub-int-step (*ins*, *p*)
case = *sub-int-with-carry*
then p-sub-int-with-carry-step (*ins*, *p*)
case = *sub1-int*
then p-sub1-int-step (*ins*, *p*)
case = *neg-int*
then p-neg-int-step (*ins*, *p*)
case = *lt-int*
then p-lt-int-step (*ins*, *p*)
case = *int-to-nat*
then p-int-to-nat-step (*ins*, *p*)
case = *add-nat*
then p-add-nat-step (*ins*, *p*)
case = *add-nat-with-carry*
then p-add-nat-with-carry-step (*ins*, *p*)
case = *add1-nat*
then p-add1-nat-step (*ins*, *p*)
case = *sub-nat*
then p-sub-nat-step (*ins*, *p*)
case = *sub-nat-with-carry*
then p-sub-nat-with-carry-step (*ins*, *p*)
case = *sub1-nat*
then p-sub1-nat-step (*ins*, *p*)
case = *lt-nat*
then p-lt-nat-step (*ins*, *p*)
case = *mult2-nat*
then p-mult2-nat-step (*ins*, *p*)
case = *mult2-nat-with-carry-out*
then p-mult2-nat-with-carry-out-step (*ins*, *p*)
case = *div2-nat*
then p-div2-nat-step (*ins*, *p*)
case = *or-bitv*
then p-or-bitv-step (*ins*, *p*)
case = *and-bitv*

then p-and-bitv-step (*ins*, *p*)
case = *not-bitv*
then p-not-bitv-step (*ins*, *p*)
case = *xor-bitv*
then p-xor-bitv-step (*ins*, *p*)
case = *rsh-bitv*
then p-rsh-bitv-step (*ins*, *p*)
case = *lsh-bitv*
then p-lsh-bitv-step (*ins*, *p*)
case = *or-bool*
then p-or-bool-step (*ins*, *p*)
case = *and-bool*
then p-and-bool-step (*ins*, *p*)
case = *not-bool*
then p-not-bool-step (*ins*, *p*)
otherwise p-halt (*p*, 'run) **endcase**

```
; We now package these up into a single function that takes an
; instruction and a state and returns the result of executing the
; given instruction in the state or causing an error.
```

DEFINITION:
p-step1 (*ins*, *p*)
= **if** p-ins-okp (*ins*, *p*) **then** p-ins-step (*ins*, *p*)
else p-halt (*p*, x-y-error-msg ('p, car (*ins*))) **endif**

```
; Of course, we are only interested in executing the current
; instruction of a state.  That is what we call stepping the state.
```

DEFINITION:
p-step (*p*)
= **if** p-psw (*p*) = 'run **then** p-step1 (p-current-instruction (*p*), *p*)
else *p* **endif**

```
; Note that p-step is a no-op if the psw is anything besides RUN.

; And given the ability to step a state, here is how you step it n
; times.
```

DEFINITION:
p (*p*, *n*)
= **if** *n* ≃ 0 **then** *p*
else p (p-step (*p*), *n* − 1) **endif**

```
; The function p, above, is the PITON machine.

; Note:  In the toy version of the system I set the final psw to
; TIMED-OUT if n exhausted before an error or HALT occurred.  I
; think the above defn is better because it gives us
; (p p (plus i j)) = (p (p p i) j).  Consider the psw of (p p n).
; If it is 'RUN then everything worked normally and n was exhausted.
; If it is 'HALT then a top-level return occurred.
; If it is anything else, an error halted the machine.
; Note that a final psw of RUN is equivalent in this world to a
; final psw of 'TIMED-OUT in the toy one.

; The initial Piton state.  We will in general be mapping down
; arbitrary Piton states.  But it is convenient to have a function
; that produces an initial state.  Here it is:
```

DEFINITION:
p0 (*temp-stk*, *prog-segment*, *data-segment*)
= p-state ('(pc (main . 0)),
push (make-p-call-frame (formal-vars (definition ('main, *prog-segment*)),
temp-stk,
temp-var-dcls (definition ('main, *prog-segment*)),
'(pc (main . 0))),
nil),
nil,
prog-segment,
data-segment,
20,
20,
32,
'run)

```
; The Icompiler

; Most of the icompiler has been defined already, in the
; icode-ins functions among the definitions of
; the P machine instruction steppers.

; With them we can generate the assembly code for a given instruction
; ins located at offset pcn in a given program.  Note that we here
; include a DL that labels the basic block of icode generated
```

```
; for the instruction.  The comment we ‘‘generate’’ is just the high
; level instruction being icompiled, complete with its label, if any.
; We completely ignore the Piton labels here, stripping them off with
; unlabel in every dealing with ins.  We get away with this because
; every reference to a Piton label, as in a JUMP statement, is converted
; to a reference to the PC instead and we generate a label for every
; PC.
```

DEFINITION:
icode1 (*ins*, *pcn*, *prog*)
= **case on** car (*ins*):
case = *call*
then icode-call (*ins*, *pcn*, *prog*)
case = *ret*
then icode-ret (*ins*, *pcn*, *prog*)
case = *locn*
then icode-locn (*ins*, *pcn*, *prog*)
case = *push-constant*
then icode-push-constant (*ins*, *pcn*, *prog*)
case = *push-local*
then icode-push-local (*ins*, *pcn*, *prog*)
case = *push-global*
then icode-push-global (*ins*, *pcn*, *prog*)
case = *push-ctrl-stk-free-size*
then icode-push-ctrl-stk-free-size (*ins*, *pcn*, *prog*)
case = *push-temp-stk-free-size*
then icode-push-temp-stk-free-size (*ins*, *pcn*, *prog*)
case = *push-temp-stk-index*
then icode-push-temp-stk-index (*ins*, *pcn*, *prog*)
case = *jump-if-temp-stk-full*
then icode-jump-if-temp-stk-full (*ins*, *pcn*, *prog*)
case = *jump-if-temp-stk-empty*
then icode-jump-if-temp-stk-empty (*ins*, *pcn*, *prog*)
case = *pop*
then icode-pop (*ins*, *pcn*, *prog*)
case = *pop**
then icode-pop* (*ins*, *pcn*, *prog*)
case = *popn*
then icode-popn (*ins*, *pcn*, *prog*)
case = *pop-local*
then icode-pop-local (*ins*, *pcn*, *prog*)
case = *pop-global*
then icode-pop-global (*ins*, *pcn*, *prog*)

case = *pop-locn*
 then icode-pop-locn (*ins*, *pcn*, *prog*)
case = *pop-call*
 then icode-pop-call (*ins*, *pcn*, *prog*)
case = *fetch-temp-stk*
 then icode-fetch-temp-stk (*ins*, *pcn*, *prog*)
case = *deposit-temp-stk*
 then icode-deposit-temp-stk (*ins*, *pcn*, *prog*)
case = *jump*
 then icode-jump (*ins*, *pcn*, *prog*)
case = *jump-case*
 then icode-jump-case (*ins*, *pcn*, *prog*)
case = *pushj*
 then icode-pushj (*ins*, *pcn*, *prog*)
case = *popj*
 then icode-popj (*ins*, *pcn*, *prog*)
case = *set-local*
 then icode-set-local (*ins*, *pcn*, *prog*)
case = *set-global*
 then icode-set-global (*ins*, *pcn*, *prog*)
case = *test-nat-and-jump*
 then icode-test-nat-and-jump (*ins*, *pcn*, *prog*)
case = *test-int-and-jump*
 then icode-test-int-and-jump (*ins*, *pcn*, *prog*)
case = *test-bool-and-jump*
 then icode-test-bool-and-jump (*ins*, *pcn*, *prog*)
case = *test-bitv-and-jump*
 then icode-test-bitv-and-jump (*ins*, *pcn*, *prog*)
case = *no-op*
 then icode-no-op (*ins*, *pcn*, *prog*)
case = *add-addr*
 then icode-add-addr (*ins*, *pcn*, *prog*)
case = *sub-addr*
 then icode-sub-addr (*ins*, *pcn*, *prog*)
case = *eq*
 then icode-eq (*ins*, *pcn*, *prog*)
case = *lt-addr*
 then icode-lt-addr (*ins*, *pcn*, *prog*)
case = *fetch*
 then icode-fetch (*ins*, *pcn*, *prog*)
case = *deposit*
 then icode-deposit (*ins*, *pcn*, *prog*)
case = *add-int*
 then icode-add-int (*ins*, *pcn*, *prog*)

case = *add-int-with-carry*
 then icode-add-int-with-carry (*ins*, *pcn*, *prog*)
case = *add1-int*
 then icode-add1-int (*ins*, *pcn*, *prog*)
case = *sub-int*
 then icode-sub-int (*ins*, *pcn*, *prog*)
case = *sub-int-with-carry*
 then icode-sub-int-with-carry (*ins*, *pcn*, *prog*)
case = *sub1-int*
 then icode-sub1-int (*ins*, *pcn*, *prog*)
case = *neg-int*
 then icode-neg-int (*ins*, *pcn*, *prog*)
case = *lt-int*
 then icode-lt-int (*ins*, *pcn*, *prog*)
case = *int-to-nat*
 then icode-int-to-nat (*ins*, *pcn*, *prog*)
case = *add-nat*
 then icode-add-nat (*ins*, *pcn*, *prog*)
case = *add-nat-with-carry*
 then icode-add-nat-with-carry (*ins*, *pcn*, *prog*)
case = *add1-nat*
 then icode-add1-nat (*ins*, *pcn*, *prog*)
case = *sub-nat*
 then icode-sub-nat (*ins*, *pcn*, *prog*)
case = *sub-nat-with-carry*
 then icode-sub-nat-with-carry (*ins*, *pcn*, *prog*)
case = *sub1-nat*
 then icode-sub1-nat (*ins*, *pcn*, *prog*)
case = *lt-nat*
 then icode-lt-nat (*ins*, *pcn*, *prog*)
case = *mult2-nat*
 then icode-mult2-nat (*ins*, *pcn*, *prog*)
case = *mult2-nat-with-carry-out*
 then icode-mult2-nat-with-carry-out (*ins*, *pcn*, *prog*)
case = *div2-nat*
 then icode-div2-nat (*ins*, *pcn*, *prog*)
case = *or-bitv*
 then icode-or-bitv (*ins*, *pcn*, *prog*)
case = *and-bitv*
 then icode-and-bitv (*ins*, *pcn*, *prog*)
case = *not-bitv*
 then icode-not-bitv (*ins*, *pcn*, *prog*)
case = *xor-bitv*
 then icode-xor-bitv (*ins*, *pcn*, *prog*)

case = *rsh-bitv*
then icode-rsh-bitv (*ins*, *pcn*, *prog*)
case = *lsh-bitv*
then icode-lsh-bitv (*ins*, *pcn*, *prog*)
case = *or-bool*
then icode-or-bool (*ins*, *pcn*, *prog*)
case = *and-bool*
then icode-and-bool (*ins*, *pcn*, *prog*)
case = *not-bool*
then icode-not-bool (*ins*, *pcn*, *prog*)
otherwise '((error)) **endcase**

```
; Note that if the opcode of ins is unrecognized we lay down an erroneous
; opcode, error.  The I-level machine halts with psw 'error on
; this instruction.
```

DEFINITION:
dl-block (*lab*, *comment*, *block*) = cons (dl (*lab*, *comment*, car (*block*)), cdr (*block*))

DEFINITION:
icode (*ins*, *pcn*, *program*)
= dl-block (cons (name (*program*), *pcn*), *ins*, icode1 (unlabel (*ins*), *pcn*, *program*))

```
; To icompile the body of a procedure we walk through the list of
; instructions in it, appending together the result of assembling
; each.  We increment the pcn on each step.
```

DEFINITION:
icompile-program-body (*lst*, *pcn*, *program*)
= **if** *lst* ≃ **nil then nil**
else append (icode (car (*lst*), *pcn*, *program*),
icompile-program-body (cdr (*lst*), 1 + *pcn*, *program*)) **endif**

```
; To icompile a program we sandwich the icompiled body between the prelude
; and postlude.
```

DEFINITION:
icompile-program (*program*)
= cons (name (*program*),
append (generate-prelude (*program*),
append (icompile-program-body (program-body (*program*),
0,
program),
generate-postlude (*program*))))

```
; The assembling of a system of programs is done by assembling each
; program in isolation.
```

DEFINITION:
icompile (*programs*)
= **if** *programs* ≃ **nil then nil**
else cons (icompile-program (car (*programs*)),
icompile (cdr (*programs*))) **endif**

```
; Proper P states

; I begin by defining what a proper piton program is, by defining
; what a proper piton instruction is, for each type of instruction.

; In the definitions below, ins is an instruction that occurs in
; the program named name in the p-state p.  Despite the fact that
; this notion of proper takes a p-state, I shall prove that it is
; invariant under execution.
```

DEFINITION:
proper-p-call-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2) ∧ definedp (cadr (*ins*), p-prog-segment (*p*)))

DEFINITION:
proper-p-ret-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-locn-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ (cadr (*ins*) ∈ local-vars (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-push-constant-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ (p-objectp (cadr (*ins*), *p*)
∨ (cadr (*ins*) = 'pc)
∨ find-labelp (cadr (*ins*),
program-body (definition (*name*,
p-prog-segment (*p*))))))

DEFINITION:
proper-p-push-local-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ (cadr (*ins*) ∈ local-vars (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-push-global-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2) ∧ definedp (cadr (*ins*), p-data-segment (*p*)))

DEFINITION:
proper-p-push-ctrl-stk-free-size-instructionp (*ins*, *name*, *p*)
= (length (*ins*) = 1)

DEFINITION:
proper-p-push-temp-stk-free-size-instructionp (*ins*, *name*, *p*)
= (length (*ins*) = 1)

DEFINITION:
proper-p-push-temp-stk-index-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2) ∧ small-naturalp (cadr (*ins*), p-word-size (*p*)))

DEFINITION:
proper-p-jump-if-temp-stk-full-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ find-labelp (cadr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-jump-if-temp-stk-empty-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ find-labelp (cadr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-pop-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-pop*-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2) ∧ small-naturalp (cadr (*ins*), p-word-size (*p*)))

DEFINITION:
proper-p-popn-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-pop-local-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ (cadr (*ins*) ∈ local-vars (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-pop-global-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2) ∧ definedp (cadr (*ins*), p-data-segment (*p*)))

DEFINITION:
proper-p-pop-locn-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ (cadr (*ins*) ∈ local-vars (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-pop-call-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-fetch-temp-stk-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-deposit-temp-stk-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-jump-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ find-labelp (cadr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-jump-case-instructionp (*ins*, *name*, *p*)
= (listp (cdr (*ins*))
∧ all-find-labelp (cdr (*ins*),
program-body (definition (*name*,
p-prog-segment (*p*)))))

DEFINITION:
proper-p-pushj-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ find-labelp (cadr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-popj-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-set-local-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2)
∧ (cadr (*ins*) ∈ local-vars (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-set-global-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 2) ∧ definedp (cadr (*ins*), p-data-segment (*p*)))

DEFINITION:
proper-p-test-nat-and-jump-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 3)
∧ find-labelp (caddr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-test-int-and-jump-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 3)
∧ find-labelp (caddr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-test-bool-and-jump-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 3)
∧ find-labelp (caddr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-test-bitv-and-jump-instructionp (*ins*, *name*, *p*)
= ((length (*ins*) = 3)
∧ find-labelp (caddr (*ins*),
program-body (definition (*name*, p-prog-segment (*p*)))))

DEFINITION:
proper-p-no-op-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-add-addr-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-sub-addr-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-eq-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-lt-addr-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-fetch-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-deposit-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-add-int-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-add-int-with-carry-instructionp (*ins*, *name*, *p*)
= (length (*ins*) = 1)

DEFINITION:
proper-p-add1-int-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-sub-int-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-sub-int-with-carry-instructionp (*ins*, *name*, *p*)
= (length (*ins*) = 1)

DEFINITION:
proper-p-sub1-int-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-neg-int-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-lt-int-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-int-to-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-add-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-add-nat-with-carry-instructionp (*ins*, *name*, *p*)
= (length (*ins*) = 1)

DEFINITION:
proper-p-add1-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-sub-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-sub-nat-with-carry-instructionp (*ins*, *name*, *p*)
= (length (*ins*) = 1)

DEFINITION:
proper-p-sub1-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-lt-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-mult2-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-mult2-nat-with-carry-out-instructionp (*ins*, *name*, *p*)
= (length (*ins*) = 1)

DEFINITION:
proper-p-div2-nat-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-or-bitv-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-and-bitv-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-not-bitv-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-xor-bitv-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-rsh-bitv-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-lsh-bitv-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-or-bool-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-and-bool-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-not-bool-instructionp (*ins*, *name*, *p*) = (length (*ins*) = 1)

DEFINITION:
proper-p-instructionp (*ins*, *name*, *p*)
= (plistp (*ins*)
  ∧ **case on** car (*ins*):
    case = *call*
    then proper-p-call-instructionp (*ins*, *name*, *p*)
    case = *ret*

then proper-p-ret-instructionp (*ins*, *name*, *p*)
case = *locn*
then proper-p-locn-instructionp (*ins*, *name*, *p*)
case = *push-constant*
then proper-p-push-constant-instructionp (*ins*, *name*, *p*)
case = *push-local*
then proper-p-push-local-instructionp (*ins*, *name*, *p*)
case = *push-global*
then proper-p-push-global-instructionp (*ins*, *name*, *p*)
case = *push-ctrl-stk-free-size*
then proper-p-push-ctrl-stk-free-size-instructionp (*ins*, *name*, *p*)
case = *push-temp-stk-free-size*
then proper-p-push-temp-stk-free-size-instructionp (*ins*, *name*, *p*)
case = *push-temp-stk-index*
then proper-p-push-temp-stk-index-instructionp (*ins*, *name*, *p*)
case = *jump-if-temp-stk-full*
then proper-p-jump-if-temp-stk-full-instructionp (*ins*, *name*, *p*)
case = *jump-if-temp-stk-empty*
then proper-p-jump-if-temp-stk-empty-instructionp (*ins*, *name*, *p*)
case = *pop*
then proper-p-pop-instructionp (*ins*, *name*, *p*)
case = *pop**
then proper-p-pop*-instructionp (*ins*, *name*, *p*)
case = *popn*
then proper-p-popn-instructionp (*ins*, *name*, *p*)
case = *pop-local*
then proper-p-pop-local-instructionp (*ins*, *name*, *p*)
case = *pop-global*
then proper-p-pop-global-instructionp (*ins*, *name*, *p*)
case = *pop-locn*
then proper-p-pop-locn-instructionp (*ins*, *name*, *p*)
case = *pop-call*
then proper-p-pop-call-instructionp (*ins*, *name*, *p*)
case = *fetch-temp-stk*
then proper-p-fetch-temp-stk-instructionp (*ins*, *name*, *p*)
case = *deposit-temp-stk*
then proper-p-deposit-temp-stk-instructionp (*ins*, *name*, *p*)
case = *jump*

then proper-p-jump-instructionp (*ins*, *name*, *p*)
case = *jump-case*
then proper-p-jump-case-instructionp (*ins*, *name*, *p*)
case = *pushj*
then proper-p-pushj-instructionp (*ins*, *name*, *p*)
case = *popj*
then proper-p-popj-instructionp (*ins*, *name*, *p*)
case = *set-local*
then proper-p-set-local-instructionp (*ins*, *name*, *p*)
case = *set-global*
then proper-p-set-global-instructionp (*ins*, *name*, *p*)
case = *test-nat-and-jump*
then proper-p-test-nat-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-int-and-jump*
then proper-p-test-int-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-bool-and-jump*
then proper-p-test-bool-and-jump-instructionp (*ins*, *name*, *p*)
case = *test-bitv-and-jump*
then proper-p-test-bitv-and-jump-instructionp (*ins*, *name*, *p*)
case = *no-op*
then proper-p-no-op-instructionp (*ins*, *name*, *p*)
case = *add-addr*
then proper-p-add-addr-instructionp (*ins*, *name*, *p*)
case = *sub-addr*
then proper-p-sub-addr-instructionp (*ins*, *name*, *p*)
case = *eq*
then proper-p-eq-instructionp (*ins*, *name*, *p*)
case = *lt-addr*
then proper-p-lt-addr-instructionp (*ins*, *name*, *p*)
case = *fetch*
then proper-p-fetch-instructionp (*ins*, *name*, *p*)
case = *deposit*
then proper-p-deposit-instructionp (*ins*, *name*, *p*)
case = *add-int*
then proper-p-add-int-instructionp (*ins*, *name*, *p*)
case = *add-int-with-carry*
then proper-p-add-int-with-carry-instructionp (*ins*, *name*, *p*)
case = *add1-int*
then proper-p-add1-int-instructionp (*ins*, *name*, *p*)
case = *sub-int*
then proper-p-sub-int-instructionp (*ins*, *name*, *p*)
case = *sub-int-with-carry*
then proper-p-sub-int-with-carry-instructionp (*ins*, *name*, *p*)
case = *sub1-int*

then proper-p-sub1-int-instructionp (*ins*, *name*, *p*)
case = *neg-int*
then proper-p-neg-int-instructionp (*ins*, *name*, *p*)
case = *lt-int*
then proper-p-lt-int-instructionp (*ins*, *name*, *p*)
case = *int-to-nat*
then proper-p-int-to-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat*
then proper-p-add-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat-with-carry*
then proper-p-add-nat-with-carry-instructionp (*ins*, *name*, *p*)
case = *add1-nat*
then proper-p-add1-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat*
then proper-p-sub-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat-with-carry*
then proper-p-sub-nat-with-carry-instructionp (*ins*, *name*, *p*)
case = *sub1-nat*
then proper-p-sub1-nat-instructionp (*ins*, *name*, *p*)
case = *lt-nat*
then proper-p-lt-nat-instructionp (*ins*, *name*, *p*)
case = *mult2-nat*
then proper-p-mult2-nat-instructionp (*ins*, *name*, *p*)
case = *mult2-nat-with-carry-out*
then proper-p-mult2-nat-with-carry-out-instructionp (*ins*,
name,
p)
case = *div2-nat*
then proper-p-div2-nat-instructionp (*ins*, *name*, *p*)
case = *or-bitv*
then proper-p-or-bitv-instructionp (*ins*, *name*, *p*)
case = *and-bitv*
then proper-p-and-bitv-instructionp (*ins*, *name*, *p*)
case = *not-bitv*
then proper-p-not-bitv-instructionp (*ins*, *name*, *p*)
case = *xor-bitv*
then proper-p-xor-bitv-instructionp (*ins*, *name*, *p*)
case = *rsh-bitv*
then proper-p-rsh-bitv-instructionp (*ins*, *name*, *p*)
case = *lsh-bitv*
then proper-p-lsh-bitv-instructionp (*ins*, *name*, *p*)
case = *or-bool*
then proper-p-or-bool-instructionp (*ins*, *name*, *p*)
case = *and-bool*

then proper-p-and-bool-instructionp (*ins*, *name*, *p*)
case = *not-bool*
then proper-p-not-bool-instructionp (*ins*, *name*, *p*)
otherwise f endcase)

DEFINITION:
legal-labelp (*ins*) = (labelledp (*ins*) → litatom (cadr (*ins*)))

```
; The following predicate recognizes lists of properly labeled
; proper instructions.
```

DEFINITION:
proper-labeled-p-instructionsp (*lst*, *name*, *p*)
= **if** *lst* ≃ **nil then** *lst* = **nil**
else legal-labelp (car (*lst*))
∧ proper-p-instructionp (unlabel (car (*lst*)), *name*, *p*)
∧ proper-labeled-p-instructionsp (cdr (*lst*), *name*, *p*) **endif**

```
; A list is fall-off proof if it is not possible for execution to
; fall off the end.  We shall prove that if a list is fall-off proof
; and you are at an instruction other than one of those listed below,
; then add1-p-pcp is ok.
```

DEFINITION:
fall-off-proofp (*lst*)
= (car (unlabel (get (length (*lst*) − 1, *lst*)))
∈ '(ret jump jump-case popj))

DEFINITION:
proper-p-program-bodyp (*lst*, *name*, *p*)
= (listp (*lst*)
∧ proper-labeled-p-instructionsp (*lst*, *name*, *p*)
∧ fall-off-proofp (*lst*))

DEFINITION:
all-litatoms (*lst*)
= **if** *lst* ≃ **nil then** *lst* = **nil**
else litatom (car (*lst*)) ∧ all-litatoms (cdr (*lst*)) **endif**

DEFINITION:
proper-p-temp-var-dclsp (*temp-var-dcls*, *p*)
= **if** *temp-var-dcls* ≃ **nil then t**
else litatom (car (car (*temp-var-dcls*)))
∧ p-objectp (cadr (car (*temp-var-dcls*)), *p*)
∧ proper-p-temp-var-dclsp (cdr (*temp-var-dcls*), *p*) **endif**

DEFINITION:
proper-p-programp (*prog*, *p*)
= (litatom (name (*prog*))
∧ all-litatoms (formal-vars (*prog*))
∧ proper-p-temp-var-dclsp (temp-var-dcls (*prog*), *p*)
∧ proper-p-program-bodyp (program-body (*prog*), name (*prog*), *p*))

DEFINITION:
proper-p-prog-segmentp (*segment*, *p*)
= **if** *segment* ≃ **nil** **then** *segment* = **nil**
else proper-p-programp (car (*segment*), *p*)
∧ proper-p-prog-segmentp (cdr (*segment*), *p*) **endif**

DEFINITION:
proper-p-alistp (*alist*, *p*)
= **if** *alist* ≃ **nil** **then** *alist* = **nil**
else listp (car (*alist*))
∧ litatom (caar (*alist*))
∧ p-objectp (cdr (car (*alist*)), *p*)
∧ proper-p-alistp (cdr (*alist*), *p*) **endif**

DEFINITION:
proper-p-framep (*frame*, *name*, *p*)
= (listp (*frame*)
∧ listp (cdr (*frame*))
∧ (cddr (*frame*) = **nil**)
∧ proper-p-alistp (bindings (*frame*), *p*)
∧ (strip-cars (bindings (*frame*))
= local-vars (definition (*name*, p-prog-segment (*p*))))
∧ p-objectp-type (`'pc`, ret-pc (*frame*), *p*))

DEFINITION:
proper-p-ctrl-stkp (*ctrl-stk*, *name*, *p*)
= **if** *ctrl-stk* ≃ **nil** **then** *ctrl-stk* = **nil**
else proper-p-framep (top (*ctrl-stk*), *name*, *p*)
∧ proper-p-ctrl-stkp (pop (*ctrl-stk*),
area-name (ret-pc (top (*ctrl-stk*))),
p) **endif**

DEFINITION:
proper-p-temp-stkp (*temp-stk*, *p*)
= **if** *temp-stk* ≃ **nil** **then** *temp-stk* = **nil**
else p-objectp (top (*temp-stk*), *p*)
∧ proper-p-temp-stkp (pop (*temp-stk*), *p*) **endif**

DEFINITION:
proper-p-area (*area*, *p*)
= (litatom (car (*area*))
∧ listp (cdr (*area*))
∧ all-p-objectps (cdr (*area*), *p*))

DEFINITION:
proper-p-data-segmentp (*data-segment*, *p*)
= **if** *data-segment* ≃ **nil** **then** *data-segment* = **nil**
else proper-p-area (car (*data-segment*), *p*)
∧ (¬ definedp (caar (*data-segment*), cdr (*data-segment*)))
∧ proper-p-data-segmentp (cdr (*data-segment*), *p*) **endif**

```
; Below is the definition of proper P state.  Intuitively,
; a proper p state has the following properties:

; (1) it is a p-state shell;
; (2) the pc is a legal address into the program segment;
; (3) the ctrl stack is non-empty;
; (4) the top frame of the ctrl stack is appropriate for
;     the current pc, which means the locals of that program
;     are bound to legal objects and the ret-pc is legal;
; (5) the rest of the ctrl stack is properly structured;
; (6) the ctrl stack isn't too large;
; (7) the temp stack contains legal objects;
; (8) the temp stack isn't too large;
; (9) the prog segment contains programs of the form
;     (name formal-vars temp-var-dcls . body) -- see proper-prog-segment;
;(10) the data segment contains non-empty areas consisting
;     of legal objects;
;(11) the resource limitations are all numeric;
;(12) the stack sizes are less than 2**word-size -- this insures that
;     the stack free sizes and indices are all representable
;(13) the word size is not 0.  This insures that lsh and other bitv
;     operations are well-defined (bitv's are non-empty).

; It is supposed to be the case that if p is proper and stepping p to
; p' does not produce an error then p' is proper.
```

DEFINITION:
proper-p-statep (*p*)
= (p-statep (*p*)
∧ p-objectp-type ('pc, p-pc (*p*), *p*)
∧ listp (p-ctrl-stk (*p*))

$\wedge$ proper-p-framep (top (p-ctrl-stk (p)), area-name (p-pc (p)), p)
$\wedge$ proper-p-ctrl-stkp (pop (p-ctrl-stk (p)),
area-name (ret-pc (top (p-ctrl-stk (p)))),
p)
$\wedge$ (p-max-ctrl-stk-size (p) $\not<$ p-ctrl-stk-size (p-ctrl-stk (p)))
$\wedge$ proper-p-temp-stkp (p-temp-stk (p), p)
$\wedge$ (p-max-temp-stk-size (p) $\not<$ length (p-temp-stk (p)))
$\wedge$ proper-p-prog-segmentp (p-prog-segment (p), p)
$\wedge$ proper-p-data-segmentp (p-data-segment (p), p)
$\wedge$ (p-max-ctrl-stk-size (p) $\in$ **N**)
$\wedge$ (p-max-temp-stk-size (p) $\in$ **N**)
$\wedge$ (p-word-size (p) $\in$ **N**)
$\wedge$ (p-max-ctrl-stk-size (p) $<$ exp (2, p-word-size (p)))
$\wedge$ (p-max-temp-stk-size (p) $<$ exp (2, p-word-size (p)))
$\wedge$ (0 $<$ p-word-size (p)))

```
; We also define the predicate that checks that a state is loadable on
; FM8502.  This is not technically part of the Piton definition but
; rather concerned with the implementation.
```

DEFINITION:
segment-length (*segment*)
= **if** *segment* $\simeq$ **nil then** 0
else length (cdr (car (*segment*)))
+ segment-length (cdr (*segment*)) **endif**

DEFINITION:
total-p-system-size (p)
= (segment-length (icompile (p-prog-segment (p)))
+ segment-length (p-data-segment (p))
+ (1 + p-max-ctrl-stk-size (p))
+ (1 + p-max-temp-stk-size (p))
+ 3)

DEFINITION:
p-loadablep (p) = (total-p-system-size (p) $<$ exp (2, p-word-size (p)))

```
; Examples of P and the Icompiler

; We are now ready to test the foregoing definitions.
```

DEFINITION:
display-p-state (p)

```
=  list('p-state,
        p-pc(p),
        list('nxt-inst, p-current-instruction(p)),
        list('bindings, bindings(top(p-ctrl-stk(p)))),
        list('temp-stk, p-temp-stk(p)),
        list('data-seg, p-data-segment(p)),
        list('psw, p-psw(p)),
        list('ret-pc, ret-pc(top(p-ctrl-stk(p)))),
        list('pop-ctrl-stk, pop(p-ctrl-stk(p))))
```

DEFINITION:

```
augment-displayed-p-state(p, p0)
=  p-state(assoc('pc, cdr(p)),
           push(p-frame(cadr(assoc('bindings, cdr(p))),
                        cadr(assoc('ret-pc, cdr(p)))),
                cadr(assoc('pop-ctrl-stk, cdr(p)))),
           cadr(assoc('temp-stk, cdr(p))),
           p-prog-segment(p0),
           cadr(assoc('data-seg, cdr(p))),
           p-max-ctrl-stk-size(p0),
           p-max-temp-stk-size(p0),
           p-word-size(p0),
           cadr(assoc('psw, cdr(p))))
```

DEFINITION:

```
ps(p, n, p0) = display-p-state(p(augment-displayed-p-state(p, p0), n))

; display-p-state abbreviates the given p state, showing just those
; components that change and showing them in the order I find convenient.
; augment-displayed-p-state takes a displayed state and ``the'' original
; one and constructs a new state whose display is the given state.
; Really it just takes the dynamic information in the displayed state
; and merges it with the static information in the original state.

; The effect of the foregoing defns is that we can use r-loop conveniently
; to step through p computations.  In particular, in r-loop

; (setq p0 (p-state ...))

; (setq p (display-p-state p0))

; then repeatedly type

; (setq p (ps p 1 p0))
```

```
; to single step from p0 and see each state in its displayed form.
; Here is a p state.  We call this p0 in our subsequent examples.

#|
(quote
(p-state '(pc (main . 0))
 '( ( nil (pc (main . 0))) )
 nil
 '((main nil nil (call test) (ret))
   (test nil nil
 (push-constant (addr (ar1 . 0)))
 (push-constant (nat 4))
 (call clear)
 (ret))
   (clear (array max-index) ((ptr (nat 0)))
  (push-local array)
  (push-local max-index)
  (add-addr)
  (pop-local ptr)

            (dl loop                        (this is the top level loop
                                             we deposit 0 into ptr
                                             test to see if ptr is array
                                             and if not decrement ptr and
                                             repeat)
              (push-constant (nat 0)))
  (push-local ptr)
  (deposit)

  (push-local ptr)
  (push-local array)
  (eq)
  (test-bool-and-jump t end)

  (push-local ptr)
  (push-constant (nat 1))
  (sub-addr)
  (pop-local ptr)
  (jump loop)
   (dl end                           (we get here when ptr is array)
                 (ret))))

 '((ar1 (nat 10)
(nat 11)
```

```
(nat 12)
(nat 13)
(nat 14)))

 12
 7
 8
 'run))

|#
```

DEFINITION: signature (*area*) = cons (car (*area*), length (*area*))

DEFINITION:
same-signature (*segment1*, *segment2*)
= **if** listp (*segment1*)
then listp (*segment2*)
∧ (signature (car (*segment1*)) = signature (car (*segment2*)))
∧ same-signature (cdr (*segment1*), cdr (*segment2*))
else *segment2* ≃ **nil endif**

```
;; ***** End stuff from defs.events *****

;; Now some other lemmas from Piton.  The ones that are easy to prove (i.e.
;; do not depend on other lemmas) we prove, the rest are added as axioms.

;; The following is from p-r.events

; I will disable type, tag and untag for sanity's sake,
; but prove the obvious relationships first.
```

THEOREM: type-tag
type (tag (*x*, *y*)) = *x*

THEOREM: untag-tag
untag (tag (*x*, *y*)) = *y*

THEOREM: equal-tags
(tag (*type*, *x*) = tag (*type*, *y*)) = (*x* = *y*)

THEOREM: cddr-tag
cddr (tag (*type*, *obj*)) = **nil**

EVENT: Disable type.

EVENT: Disable tag.

EVENT: Disable untag.

```
; Error messages are irrelevant as long as they are not 'run or 'halt.
```

THEOREM: not-equal-x-y-error-msg-run
x-y-error-msg (*x*, *y*) ≠ 'run

THEOREM: not-equal-x-y-error-msg-halt
x-y-error-msg (*x*, *y*) ≠ 'halt

EVENT: Disable x-y-error-msg.

THEOREM: p-objectp-opener
((type (*x*) = 'nat)
 → (p-objectp (*x*, *p*)
 = ((cddr (*x*) = **nil**)
 ∧ small-naturalp (untag (*x*), p-word-size (*p*)))))
∧ ((type (*x*) = 'int)
 → (p-objectp (*x*, *p*)
 = ((cddr (*x*) = **nil**)
 ∧ small-integerp (untag (*x*), p-word-size (*p*)))))
∧ ((type (*x*) = 'bitv)
 → (p-objectp (*x*, *p*)
 = ((cddr (*x*) = **nil**)
 ∧ bit-vectorp (untag (*x*), p-word-size (*p*)))))
∧ ((type (*x*) = 'bool)
 → (p-objectp (*x*, *p*)
 = ((cddr (*x*) = **nil**) ∧ booleanp (untag (*x*)))))
∧ ((type (*x*) = 'addr)
 → (p-objectp (*x*, *p*)
 = ((cddr (*x*) = **nil**)
 ∧ adpp (untag (*x*), p-data-segment (*p*)))))
∧ ((type (*x*) = 'pc)
 → (p-objectp (*x*, *p*)
 = ((cddr (*x*) = **nil**)
 ∧ pcpp (untag (*x*), p-prog-segment (*p*)))))
∧ ((type (*x*) = 'subr)

→ (p-objectp (*x*, *p*)
= ((cddr (*x*) = **nil**)
∧ definedp (untag (*x*), p-prog-segment (*p*)))))

EVENT: Disable p-objectp.

EVENT: Disable name.

EVENT: Disable formal-vars.

EVENT: Disable temp-var-dcls.

EVENT: Disable program-body.

THEOREM: bindings-p-frame
bindings (p-frame (*alist*, *pc*)) = *alist*

THEOREM: ret-pc-p-frame
ret-pc (p-frame (*alist*, *pc*)) = *pc*

EVENT: Disable bindings.

EVENT: Disable ret-pc.

EVENT: Disable p-frame.

EVENT: Disable proper-p-instructionp.

EVENT: Disable legal-labelp.

THEOREM: proper-labeled-p-instructionsp-implies-labelp-and-instructionp
(proper-labeled-p-instructionsp (*lst*, *name*, *p*) ∧ (*x* ∈ *lst*))
→ (legal-labelp (*x*) ∧ proper-p-instructionp (unlabel (*x*), *name*, *p*))

THEOREM: proper-p-prog-segmentp-implies-proper-p-programp
(proper-p-prog-segmentp (*segment*, *p*) ∧ (*prog* ∈ *segment*))
→ proper-p-programp (*prog*, *p*)

```
; Added disable to speed up proof, AF
```

THEOREM: member-assoc
definedp (*name*, *alist*) → (assoc (*name*, *alist*) ∈ *alist*)

THEOREM: name-assoc
definedp (*name*, *alist*) → (name (assoc (*name*, *alist*)) = *name*)

THEOREM: member-get
(*n* < length (*lst*)) → (get (*n*, *lst*) ∈ *lst*)

THEOREM: proper-p-statep-implies-proper-p-instructionp
(proper-p-prog-segmentp (p-prog-segment (*p*), *p*)
∧ definedp (car (untag (p-pc (*p*))), p-prog-segment (*p*))
∧ (cdr (untag (p-pc (*p*)))
< length (program-body (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*))))))
→ proper-p-instructionp (p-current-instruction (*p*), car (untag (p-pc (*p*))), *p*)

```
; Now I restructure proper-p-statep into two parts, those that
; have to do with arithmetic and those that do not.  The parts
; that do not are arithmetic, I confine to proper-p-statep1,
; which I keep closed and backchain into as needed.  The other
; parts I keep out in the open.  In addition, I throw in the
; proper-p-instructionp proved above, essentially forward
; chaining through the lemma.
```

EVENT: Disable p-loadablep.

DEFINITION:
proper-p-statep1 (*p*)
= (p-statep (*p*)
∧ (type (p-pc (*p*)) = 'pc)
∧ listp (p-pc (*p*))
∧ (cddr (p-pc (*p*)) = **nil**)
∧ listp (untag (p-pc (*p*)))
∧ definedp (car (untag (p-pc (*p*))), p-prog-segment (*p*))
∧ listp (p-ctrl-stk (*p*))
∧ listp (car (p-ctrl-stk (*p*)))
∧ listp (cdr (car (p-ctrl-stk (*p*))))
∧ (cddr (car (p-ctrl-stk (*p*))) = **nil**)
∧ proper-p-alistp (bindings (car (p-ctrl-stk (*p*))), *p*)
∧ (strip-cars (bindings (car (p-ctrl-stk (*p*))))
= append (formal-vars (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*))),

strip-cars (temp-var-dcls (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*))))))
∧ (type (ret-pc (car (p-ctrl-stk (*p*)))) = 'pc)
∧ listp (ret-pc (car (p-ctrl-stk (*p*))))
∧ (cddr (ret-pc (car (p-ctrl-stk (*p*)))) = **nil**)
∧ listp (untag (ret-pc (car (p-ctrl-stk (*p*)))))
∧ definedp (car (untag (ret-pc (car (p-ctrl-stk (*p*))))),
p-prog-segment (*p*))
∧ proper-p-ctrl-stkp (cdr (p-ctrl-stk (*p*)),
car (untag (ret-pc (car (p-ctrl-stk (*p*))))),
p)
∧ proper-p-temp-stkp (p-temp-stk (*p*), *p*)
∧ proper-p-prog-segmentp (p-prog-segment (*p*), *p*)
∧ proper-p-data-segmentp (p-data-segment (*p*), *p*))

THEOREM: proper-p-statep-restructuring
proper-p-statep (*p*)
= (proper-p-statep1 (*p*)
∧ (cdr (untag (p-pc (*p*))) ∈ **N**)
∧ (cdr (untag (p-pc (*p*)))
< length (program-body (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*)))))
∧ (cdr (untag (ret-pc (car (p-ctrl-stk (*p*))))) ∈ **N**)
∧ (cdr (untag (ret-pc (car (p-ctrl-stk (*p*))))))
< length (program-body (assoc (car (untag (ret-pc (car (p-ctrl-stk (*p*))))),
p-prog-segment (*p*)))))
∧ (p-max-ctrl-stk-size (*p*) ≮ p-ctrl-stk-size (p-ctrl-stk (*p*)))
∧ (p-max-temp-stk-size (*p*) ≮ length (p-temp-stk (*p*)))
∧ (p-max-ctrl-stk-size (*p*) ∈ **N**)
∧ (p-max-temp-stk-size (*p*) ∈ **N**)
∧ (p-word-size (*p*) ∈ **N**)
∧ (p-max-ctrl-stk-size (*p*) < exp (2, p-word-size (*p*)))
∧ (p-max-temp-stk-size (*p*) < exp (2, p-word-size (*p*)))
∧ (0 < p-word-size (*p*))
∧ proper-p-instructionp (p-current-instruction (*p*),
car (untag (p-pc (*p*))),
p))

THEOREM: proper-p-statep1-properties
proper-p-statep1 (*p*)
→ (p-statep (*p*)
∧ (type (p-pc (*p*)) = 'pc)
∧ listp (p-pc (*p*))
∧ (cddr (p-pc (*p*)) = **nil**)

∧ listp (untag (p-pc (*p*)))
∧ definedp (car (untag (p-pc (*p*))), p-prog-segment (*p*))
∧ listp (p-ctrl-stk (*p*))
∧ listp (car (p-ctrl-stk (*p*)))
∧ listp (cdr (car (p-ctrl-stk (*p*))))
∧ (cddr (car (p-ctrl-stk (*p*))) = **nil**)
∧ proper-p-alistp (bindings (car (p-ctrl-stk (*p*))), *p*)
∧ (strip-cars (bindings (car (p-ctrl-stk (*p*))))
= append (formal-vars (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*))),
strip-cars (temp-var-dcls (assoc (car (untag (p-pc (*p*))),
p-prog-segment (*p*))))))
∧ (type (ret-pc (car (p-ctrl-stk (*p*)))) = 'pc)
∧ listp (ret-pc (car (p-ctrl-stk (*p*))))
∧ (cddr (ret-pc (car (p-ctrl-stk (*p*)))) = **nil**)
∧ listp (untag (ret-pc (car (p-ctrl-stk (*p*)))))
∧ definedp (car (untag (ret-pc (car (p-ctrl-stk (*p*))))),
p-prog-segment (*p*))
∧ proper-p-ctrl-stkp (cdr (p-ctrl-stk (*p*)),
car (untag (ret-pc (car (p-ctrl-stk (*p*))))),
p)
∧ proper-p-temp-stkp (p-temp-stk (*p*), *p*)
∧ proper-p-prog-segmentp (p-prog-segment (*p*), *p*)
∧ proper-p-data-segmentp (p-data-segment (*p*), *p*))

EVENT: Disable proper-p-statep.

EVENT: Disable proper-p-statep1.

EVENT: Disable p-current-instruction.

```
; Now I provide a simple delayed opener for proper-p-instructionp:
```

THEOREM: proper-p-instructionp-opener
(car (*ins*) = pack (*xxx*))
→ (proper-p-instructionp (*ins*, *name*, *p*)
= (plistp (*ins*)
∧ **case on** car (*ins*):
case = *call*
then proper-p-call-instructionp (*ins*, *name*, *p*)
case = *ret*
then proper-p-ret-instructionp (*ins*, *name*, *p*)

case = *locn*
 then proper-p-locn-instructionp (*ins*, *name*, *p*)
case = *push-constant*
 then proper-p-push-constant-instructionp (*ins*, *name*, *p*)
case = *push-local*
 then proper-p-push-local-instructionp (*ins*, *name*, *p*)
case = *push-global*
 then proper-p-push-global-instructionp (*ins*, *name*, *p*)
case = *push-ctrl-stk-free-size*
 then proper-p-push-ctrl-stk-free-size-instructionp (*ins*,
name,
p)
case = *push-temp-stk-free-size*
 then proper-p-push-temp-stk-free-size-instructionp (*ins*,
name,
p)
case = *push-temp-stk-index*
 then proper-p-push-temp-stk-index-instructionp (*ins*,
name,
p)
case = *jump-if-temp-stk-full*
 then proper-p-jump-if-temp-stk-full-instructionp (*ins*,
name,
p)
case = *jump-if-temp-stk-empty*
 then proper-p-jump-if-temp-stk-empty-instructionp (*ins*,
name,
p)
case = *pop*
 then proper-p-pop-instructionp (*ins*, *name*, *p*)
case = *pop**
 then proper-p-pop*-instructionp (*ins*, *name*, *p*)
case = *popn*
 then proper-p-popn-instructionp (*ins*, *name*, *p*)
case = *pop-local*
 then proper-p-pop-local-instructionp (*ins*, *name*, *p*)
case = *pop-global*
 then proper-p-pop-global-instructionp (*ins*, *name*, *p*)
case = *pop-locn*
 then proper-p-pop-locn-instructionp (*ins*, *name*, *p*)
case = *pop-call*
 then proper-p-pop-call-instructionp (*ins*, *name*, *p*)
case = *fetch-temp-stk*
 then proper-p-fetch-temp-stk-instructionp (*ins*,

name,
p)
case = *deposit-temp-stk*
then proper-p-deposit-temp-stk-instructionp (*ins*,
name,
p)
case = *jump*
then proper-p-jump-instructionp (*ins*, *name*, *p*)
case = *jump-case*
then proper-p-jump-case-instructionp (*ins*, *name*, *p*)
case = *pushj*
then proper-p-pushj-instructionp (*ins*, *name*, *p*)
case = *popj*
then proper-p-popj-instructionp (*ins*, *name*, *p*)
case = *set-local*
then proper-p-set-local-instructionp (*ins*, *name*, *p*)
case = *set-global*
then proper-p-set-global-instructionp (*ins*, *name*, *p*)
case = *test-nat-and-jump*
then proper-p-test-nat-and-jump-instructionp (*ins*,
name,
p)
case = *test-int-and-jump*
then proper-p-test-int-and-jump-instructionp (*ins*,
name,
p)
case = *test-bool-and-jump*
then proper-p-test-bool-and-jump-instructionp (*ins*,
name,
p)
case = *test-bitv-and-jump*
then proper-p-test-bitv-and-jump-instructionp (*ins*,
name,
p)
case = *no-op*
then proper-p-no-op-instructionp (*ins*, *name*, *p*)
case = *add-addr*
then proper-p-add-addr-instructionp (*ins*, *name*, *p*)
case = *sub-addr*
then proper-p-sub-addr-instructionp (*ins*, *name*, *p*)
case = *eq*
then proper-p-eq-instructionp (*ins*, *name*, *p*)
case = *lt-addr*
then proper-p-lt-addr-instructionp (*ins*, *name*, *p*)

case = *fetch*
 then proper-p-fetch-instructionp (*ins*, *name*, *p*)
case = *deposit*
 then proper-p-deposit-instructionp (*ins*, *name*, *p*)
case = *add-int*
 then proper-p-add-int-instructionp (*ins*, *name*, *p*)
case = *add-int-with-carry*
 then proper-p-add-int-with-carry-instructionp (*ins*,
name,
p)
case = *add1-int*
 then proper-p-add1-int-instructionp (*ins*, *name*, *p*)
case = *sub-int*
 then proper-p-sub-int-instructionp (*ins*, *name*, *p*)
case = *sub-int-with-carry*
 then proper-p-sub-int-with-carry-instructionp (*ins*,
name,
p)
case = *sub1-int*
 then proper-p-sub1-int-instructionp (*ins*, *name*, *p*)
case = *neg-int*
 then proper-p-neg-int-instructionp (*ins*, *name*, *p*)
case = *lt-int*
 then proper-p-lt-int-instructionp (*ins*, *name*, *p*)
case = *int-to-nat*
 then proper-p-int-to-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat*
 then proper-p-add-nat-instructionp (*ins*, *name*, *p*)
case = *add-nat-with-carry*
 then proper-p-add-nat-with-carry-instructionp (*ins*,
name,
p)
case = *add1-nat*
 then proper-p-add1-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat*
 then proper-p-sub-nat-instructionp (*ins*, *name*, *p*)
case = *sub-nat-with-carry*
 then proper-p-sub-nat-with-carry-instructionp (*ins*,
name,
p)
case = *sub1-nat*
 then proper-p-sub1-nat-instructionp (*ins*, *name*, *p*)
case = *lt-nat*
 then proper-p-lt-nat-instructionp (*ins*, *name*, *p*)

case = *mult2-nat*
 then proper-p-mult2-nat-instructionp (*ins*, *name*, *p*)
case = *mult2-nat-with-carry-out*
 then proper-p-mult2-nat-with-carry-out-instructionp (*ins*, *name*, *p*)
case = *div2-nat*
 then proper-p-div2-nat-instructionp (*ins*, *name*, *p*)
case = *or-bitv*
 then proper-p-or-bitv-instructionp (*ins*, *name*, *p*)
case = *and-bitv*
 then proper-p-and-bitv-instructionp (*ins*, *name*, *p*)
case = *not-bitv*
 then proper-p-not-bitv-instructionp (*ins*, *name*, *p*)
case = *xor-bitv*
 then proper-p-xor-bitv-instructionp (*ins*, *name*, *p*)
case = *rsh-bitv*
 then proper-p-rsh-bitv-instructionp (*ins*, *name*, *p*)
case = *lsh-bitv*
 then proper-p-lsh-bitv-instructionp (*ins*, *name*, *p*)
case = *or-bool*
 then proper-p-or-bool-instructionp (*ins*, *name*, *p*)
case = *and-bool*
 then proper-p-and-bool-instructionp (*ins*, *name*, *p*)
case = *not-bool*
 then proper-p-not-bool-instructionp (*ins*, *name*, *p*)
otherwise f endcase))

THEOREM: length-put-assoc
length (put-assoc (*val*, *name*, *alist*)) = length (*alist*)

THEOREM: assoc-put-assoc-1
(*name1* ≠ *name2*)
→ (assoc (*name1*, put-assoc (*val*, *name2*, *alist*)) = assoc (*name1*, *alist*))

THEOREM: assoc-put-assoc-2
definedp (*name*, *alist*)
→ (assoc (*name*, put-assoc (*val*, *name*, *alist*)) = cons (*name*, *val*))

```
;; The original lemma SAME-SIGNATURE-PUT-ASSOC-1 had the DEFINEDP
;; term below in it.  It is not necessary so I commented it out.  -- AF
```

THEOREM: same-signature-put-assoc-1
(litatom (*name*) ∧ (length (*val*) = length (value (*name*, *segment1*))))
→ (same-signature (put-assoc (*val*, *name*, *segment1*), *segment2*)
 = same-signature (*segment1*, *segment2*))

```
;; The original lemma SAME-SIGNATURE-PUT-ASSOC-2 had the DEFINEDP
;; term below in it.  It is not necessary so I commented it out.  -- AF
```

THEOREM: same-signature-put-assoc-2
(litatom (*name*) ∧ (length (*val*) = length (value (*name*, *segment2*))))
→ (same-signature (*segment1*, put-assoc (*val*, *name*, *segment2*))
= same-signature (*segment1*, *segment2*))

EVENT: Disable same-signature.

```
; Now I want to disable get and put so that they are only manipulated by
; the lemmas we have proved.
```

EVENT: Disable get.

EVENT: Disable put.

THEOREM: length-first-n
length (first-n (*n*, *x*)) = fix (*n*)

THEOREM: same-signature-implies-equal-lengths
same-signature (*segment1*, *segment2*)
→ (length (cdr (assoc (*name*, *segment1*)))
= length (cdr (assoc (*name*, *segment2*))))

THEOREM: same-signature-implies-equal-definedp
same-signature (*segment1*, *segment2*)
→ (definedp (*name*, *segment1*) = definedp (*name*, *segment2*))

THEOREM: same-signature-reflexive-generalized
same-signature (*segment*, *segment*)

THEOREM: definedp-put-assoc
definedp (*name1*, put-assoc (*val*, *name2*, *alist*)) = definedp (*name1*, *alist*)

THEOREM: p-objectp-type-opener
(p-objectp-type (`'nat`, *x*, *p*)
= ((type (*x*) = `'nat`)
∧ (cddr (*x*) = **nil**)
∧ small-naturalp (untag (*x*), p-word-size (*p*))))
∧ (p-objectp-type (`'int`, *x*, *p*)
= ((type (*x*) = `'int`)
∧ (cddr (*x*) = **nil**)

$$
\begin{array}{ll}
 & \quad \wedge \quad \text{small-integerp}\,(\text{untag}\,(x),\ \text{p-word-size}\,(p)))) \\
\wedge & (\text{p-objectp-type}\,(\texttt{'bitv},\ x,\ p) \\
 & \quad = \quad ((\text{type}\,(x) = \texttt{'bitv}) \\
 & \qquad \wedge \quad (\text{cddr}\,(x) = \textbf{nil}) \\
 & \qquad \wedge \quad \text{bit-vectorp}\,(\text{untag}\,(x),\ \text{p-word-size}\,(p)))) \\
\wedge & (\text{p-objectp-type}\,(\texttt{'bool},\ x,\ p) \\
 & \quad = \quad ((\text{type}\,(x) = \texttt{'bool}) \\
 & \qquad \wedge \quad (\text{cddr}\,(x) = \textbf{nil}) \\
 & \qquad \wedge \quad \text{booleanp}\,(\text{untag}\,(x)))) \\
\wedge & (\text{p-objectp-type}\,(\texttt{'addr},\ x,\ p) \\
 & \quad = \quad ((\text{type}\,(x) = \texttt{'addr}) \\
 & \qquad \wedge \quad (\text{cddr}\,(x) = \textbf{nil}) \\
 & \qquad \wedge \quad \text{adpp}\,(\text{untag}\,(x),\ \text{p-data-segment}\,(p)))) \\
\wedge & (\text{p-objectp-type}\,(\texttt{'pc},\ x,\ p) \\
 & \quad = \quad ((\text{type}\,(x) = \texttt{'pc}) \\
 & \qquad \wedge \quad (\text{cddr}\,(x) = \textbf{nil}) \\
 & \qquad \wedge \quad \text{pcpp}\,(\text{untag}\,(x),\ \text{p-prog-segment}\,(p)))) \\
\wedge & (\text{p-objectp-type}\,(\texttt{'subr},\ x,\ p) \\
 & \quad = \quad ((\text{type}\,(x) = \texttt{'subr}) \\
 & \qquad \wedge \quad (\text{cddr}\,(x) = \textbf{nil}) \\
 & \qquad \wedge \quad \text{definedp}\,(\text{untag}\,(x),\ \text{p-prog-segment}\,(p))))
\end{array}
$$

EVENT: Disable p-objectp-type.

```
;; This is from Piton but with PROPERP instead of PLISTP
```

THEOREM: proper-p-alistp-append

$$
\begin{array}{ll}
 & \text{plistp}\,(a) \\
\rightarrow & (\text{proper-p-alistp}\,(\text{append}\,(a,\ b),\ p) \\
 & \quad = \quad (\text{proper-p-alistp}\,(a,\ p) \wedge \text{proper-p-alistp}\,(b,\ p)))
\end{array}
$$

THEOREM: strip-cars-put-assoc

$$\text{strip-cars}\,(\text{put-assoc}\,(val,\ var,\ alist)) = \text{strip-cars}\,(alist)$$

```
; So after the rule is applied, we have to prove two proper-p-alistps.
; The first is for the pairlist of the formals to the actuals.
; The second is for the initialization of the temp vars.

; The first is handled by:
```

THEOREM: proper-p-alistp-pairlist

$$
\begin{array}{ll}
 & (\text{all-litatoms}\,(formals) \\
 & \quad \wedge \quad \text{all-p-objectps}\,(actuals,\ p) \\
 & \quad \wedge \quad (\text{length}\,(formals) = \text{length}\,(actuals))) \\
\rightarrow & \quad \text{proper-p-alistp}\,(\text{pairlist}\,(formals,\ actuals),\ p)
\end{array}
$$

```
; where the hypotheses above are relieved as follows.  The all-litatoms
; of the formals is derived from proper-p-prog-segmentp:
```

THEOREM: all-litatoms-formal-vars-generalized
(proper-p-prog-segmentp (*segment*, *p*) ∧ definedp (*name*, *segment*))
→ all-litatoms (formal-vars (assoc (*name*, *segment*)))

THEOREM: all-litatoms-formal-vars
(proper-p-prog-segmentp (p-prog-segment (*p*), *p*)
∧ definedp (*name*, p-prog-segment (*p*)))
→ all-litatoms (formal-vars (assoc (*name*, p-prog-segment (*p*))))

```
; The third hypothesis of the pairlist lemma above is that the lengths of
; the formals and actuals are the same.  That follows from what we've
; already proved.

; Now we move to the second proper-p-alistp task, namely, for the
; initialization of the temp vars.  That is handled by:
```

THEOREM: proper-p-prog-segmentp-implies-proper-p-temp-var-dclsp
(proper-p-prog-segmentp (*segment*, *p*) ∧ definedp (*name*, *segment*))
→ proper-p-temp-var-dclsp (temp-var-dcls (assoc (*name*, *segment*)), *p*)

THEOREM: proper-p-alistp-pair-temps-with-initial-values
proper-p-temp-var-dclsp (*var-dcls*, *p*)
→ proper-p-alistp (pair-temps-with-initial-values (*var-dcls*), *p*)

EVENT: Disable p-call-okp.

EVENT: Disable p-call-step.

EVENT: Disable p-ret-okp.

EVENT: Disable p-ret-step.

EVENT: Disable p-locn-okp.

EVENT: Disable p-locn-step.

EVENT: Disable p-push-constant-okp.

EVENT: Disable p-push-constant-step.

EVENT: Disable p-push-local-okp.

EVENT: Disable p-push-local-step.

EVENT: Disable p-push-global-okp.

EVENT: Disable p-push-global-step.

EVENT: Disable p-push-ctrl-stk-free-size-okp.

EVENT: Disable p-push-ctrl-stk-free-size-step.

EVENT: Disable p-push-temp-stk-free-size-okp.

EVENT: Disable p-push-temp-stk-free-size-step.

EVENT: Disable p-push-temp-stk-index-okp.

EVENT: Disable p-push-temp-stk-index-step.

EVENT: Disable p-jump-if-temp-stk-full-okp.

EVENT: Disable p-jump-if-temp-stk-full-step.

EVENT: Disable p-jump-if-temp-stk-empty-okp.

EVENT: Disable p-jump-if-temp-stk-empty-step.

EVENT: Disable p-pop-okp.

EVENT: Disable p-pop-step.

EVENT: Disable p-pop*-okp.

EVENT: Disable p-pop*-step.

EVENT: Disable p-popn-okp.

EVENT: Disable p-popn-step.

EVENT: Disable p-pop-local-okp.

EVENT: Disable p-pop-local-step.

EVENT: Disable p-pop-global-okp.

EVENT: Disable p-pop-global-step.

EVENT: Disable p-pop-locn-okp.

EVENT: Disable p-pop-locn-step.

EVENT: Disable p-pop-call-okp.

EVENT: Disable p-pop-call-step.

EVENT: Disable p-fetch-temp-stk-okp.

EVENT: Disable p-fetch-temp-stk-step.

EVENT: Disable p-deposit-temp-stk-okp.

EVENT: Disable p-deposit-temp-stk-step.

EVENT: Disable p-jump-okp.

EVENT: Disable p-jump-step.

EVENT: Disable p-jump-case-okp.

EVENT: Disable p-jump-case-step.

EVENT: Disable p-pushj-okp.

EVENT: Disable p-pushj-step.

EVENT: Disable p-popj-okp.

EVENT: Disable p-popj-step.

EVENT: Disable p-set-local-okp.

EVENT: Disable p-set-local-step.

EVENT: Disable p-set-global-okp.

EVENT: Disable p-set-global-step.

EVENT: Disable p-test-nat-and-jump-okp.

EVENT: Disable p-test-nat-and-jump-step.

EVENT: Disable p-test-int-and-jump-okp.

EVENT: Disable p-test-int-and-jump-step.

EVENT: Disable p-test-bool-and-jump-okp.

EVENT: Disable p-test-bool-and-jump-step.

EVENT: Disable p-test-bitv-and-jump-okp.

EVENT: Disable p-test-bitv-and-jump-step.

EVENT: Disable p-no-op-okp.

EVENT: Disable p-no-op-step.

EVENT: Disable p-add-addr-okp.

EVENT: Disable p-add-addr-step.

EVENT: Disable p-sub-addr-okp.

EVENT: Disable p-sub-addr-step.

EVENT: Disable p-eq-okp.

EVENT: Disable p-eq-step.

EVENT: Disable p-lt-addr-okp.

EVENT: Disable p-lt-addr-step.

EVENT: Disable p-fetch-okp.

EVENT: Disable p-fetch-step.

EVENT: Disable p-deposit-okp.

EVENT: Disable p-deposit-step.

EVENT: Disable p-add-int-okp.

EVENT: Disable p-add-int-step.

EVENT: Disable p-add-int-with-carry-okp.

EVENT: Disable p-add-int-with-carry-step.

EVENT: Disable p-add1-int-okp.

EVENT: Disable p-add1-int-step.

EVENT: Disable p-sub-int-okp.

EVENT: Disable p-sub-int-step.

EVENT: Disable p-sub-int-with-carry-okp.

EVENT: Disable p-sub-int-with-carry-step.

EVENT: Disable p-sub1-int-okp.

EVENT: Disable p-sub1-int-step.

EVENT: Disable p-neg-int-okp.

EVENT: Disable p-neg-int-step.

EVENT: Disable p-lt-int-okp.

EVENT: Disable p-lt-int-step.

EVENT: Disable p-int-to-nat-okp.

EVENT: Disable p-int-to-nat-step.

EVENT: Disable p-add-nat-okp.

EVENT: Disable p-add-nat-step.

EVENT: Disable p-add-nat-with-carry-okp.

EVENT: Disable p-add-nat-with-carry-step.

EVENT: Disable p-add1-nat-okp.

EVENT: Disable p-add1-nat-step.

EVENT: Disable p-sub-nat-okp.

EVENT: Disable p-sub-nat-step.

EVENT: Disable p-sub-nat-with-carry-okp.

EVENT: Disable p-sub-nat-with-carry-step.

EVENT: Disable p-sub1-nat-okp.

EVENT: Disable p-sub1-nat-step.

EVENT: Disable p-lt-nat-okp.

EVENT: Disable p-lt-nat-step.

EVENT: Disable p-mult2-nat-okp.

EVENT: Disable p-mult2-nat-step.

EVENT: Disable p-mult2-nat-with-carry-out-okp.

EVENT: Disable p-mult2-nat-with-carry-out-step.

EVENT: Disable p-div2-nat-okp.

EVENT: Disable p-div2-nat-step.

EVENT: Disable p-or-bitv-okp.

EVENT: Disable p-or-bitv-step.

EVENT: Disable p-and-bitv-okp.

EVENT: Disable p-and-bitv-step.

EVENT: Disable p-not-bitv-okp.

EVENT: Disable p-not-bitv-step.

EVENT: Disable p-xor-bitv-okp.

EVENT: Disable p-xor-bitv-step.

EVENT: Disable p-rsh-bitv-okp.

EVENT: Disable p-rsh-bitv-step.

EVENT: Disable p-lsh-bitv-okp.

EVENT: Disable p-lsh-bitv-step.

EVENT: Disable p-or-bool-okp.

EVENT: Disable p-or-bool-step.

EVENT: Disable p-and-bool-okp.

EVENT: Disable p-and-bool-step.

EVENT: Disable p-not-bool-okp.

EVENT: Disable p-not-bool-step.

DEFINITION:
p-ins-okp2 (*ins*, *p*)
= **case on** car (*ins*):
case = *eq*
then p-eq-okp (*ins*, *p*)
case = *lt-addr*

then p-lt-addr-okp (*ins*, *p*)
case = *fetch*
then p-fetch-okp (*ins*, *p*)
case = *deposit*
then p-deposit-okp (*ins*, *p*)
case = *add-int*
then p-add-int-okp (*ins*, *p*)
case = *add-int-with-carry*
then p-add-int-with-carry-okp (*ins*, *p*)
case = *add1-int*
then p-add1-int-okp (*ins*, *p*)
case = *sub-int*
then p-sub-int-okp (*ins*, *p*)
case = *sub-int-with-carry*
then p-sub-int-with-carry-okp (*ins*, *p*)
case = *sub1-int*
then p-sub1-int-okp (*ins*, *p*)
case = *neg-int*
then p-neg-int-okp (*ins*, *p*)
case = *lt-int*
then p-lt-int-okp (*ins*, *p*)
case = *int-to-nat*
then p-int-to-nat-okp (*ins*, *p*)
case = *add-nat*
then p-add-nat-okp (*ins*, *p*)
case = *add-nat-with-carry*
then p-add-nat-with-carry-okp (*ins*, *p*)
case = *add1-nat*
then p-add1-nat-okp (*ins*, *p*)
case = *sub-nat*
then p-sub-nat-okp (*ins*, *p*)
case = *sub-nat-with-carry*
then p-sub-nat-with-carry-okp (*ins*, *p*)
case = *sub1-nat*
then p-sub1-nat-okp (*ins*, *p*)
case = *lt-nat*
then p-lt-nat-okp (*ins*, *p*)
case = *mult2-nat*
then p-mult2-nat-okp (*ins*, *p*)
case = *mult2-nat-with-carry-out*
then p-mult2-nat-with-carry-out-okp (*ins*, *p*)
case = *div2-nat*
then p-div2-nat-okp (*ins*, *p*)
case = *or-bitv*

then p-or-bitv-okp (*ins*, *p*)
case = *and-bitv*
then p-and-bitv-okp (*ins*, *p*)
case = *not-bitv*
then p-not-bitv-okp (*ins*, *p*)
case = *xor-bitv*
then p-xor-bitv-okp (*ins*, *p*)
case = *rsh-bitv*
then p-rsh-bitv-okp (*ins*, *p*)
case = *lsh-bitv*
then p-lsh-bitv-okp (*ins*, *p*)
case = *or-bool*
then p-or-bool-okp (*ins*, *p*)
case = *and-bool*
then p-and-bool-okp (*ins*, *p*)
case = *not-bool*
then p-not-bool-okp (*ins*, *p*)
otherwise f endcase

DEFINITION:
p-ins-okp1 (*ins*, *p*)
= **case on** car (*ins*):
case = *call*
then p-call-okp (*ins*, *p*)
case = *ret*
then p-ret-okp (*ins*, *p*)
case = *locn*
then p-locn-okp (*ins*, *p*)
case = *push-constant*
then p-push-constant-okp (*ins*, *p*)
case = *push-local*
then p-push-local-okp (*ins*, *p*)
case = *push-global*
then p-push-global-okp (*ins*, *p*)
case = *push-ctrl-stk-free-size*
then p-push-ctrl-stk-free-size-okp (*ins*, *p*)
case = *push-temp-stk-free-size*
then p-push-temp-stk-free-size-okp (*ins*, *p*)
case = *push-temp-stk-index*
then p-push-temp-stk-index-okp (*ins*, *p*)
case = *jump-if-temp-stk-full*
then p-jump-if-temp-stk-full-okp (*ins*, *p*)
case = *jump-if-temp-stk-empty*
then p-jump-if-temp-stk-empty-okp (*ins*, *p*)

case = *pop*
 then p-pop-okp (*ins*, *p*)
case = *pop**
 then p-pop*-okp (*ins*, *p*)
case = *popn*
 then p-popn-okp (*ins*, *p*)
case = *pop-local*
 then p-pop-local-okp (*ins*, *p*)
case = *pop-global*
 then p-pop-global-okp (*ins*, *p*)
case = *pop-locn*
 then p-pop-locn-okp (*ins*, *p*)
case = *pop-call*
 then p-pop-call-okp (*ins*, *p*)
case = *fetch-temp-stk*
 then p-fetch-temp-stk-okp (*ins*, *p*)
case = *deposit-temp-stk*
 then p-deposit-temp-stk-okp (*ins*, *p*)
case = *jump*
 then p-jump-okp (*ins*, *p*)
case = *jump-case*
 then p-jump-case-okp (*ins*, *p*)
case = *pushj*
 then p-pushj-okp (*ins*, *p*)
case = *popj*
 then p-popj-okp (*ins*, *p*)
case = *set-local*
 then p-set-local-okp (*ins*, *p*)
case = *set-global*
 then p-set-global-okp (*ins*, *p*)
case = *test-nat-and-jump*
 then p-test-nat-and-jump-okp (*ins*, *p*)
case = *test-int-and-jump*
 then p-test-int-and-jump-okp (*ins*, *p*)
case = *test-bool-and-jump*
 then p-test-bool-and-jump-okp (*ins*, *p*)
case = *test-bitv-and-jump*
 then p-test-bitv-and-jump-okp (*ins*, *p*)
case = *no-op*
 then p-no-op-okp (*ins*, *p*)
case = *add-addr*
 then p-add-addr-okp (*ins*, *p*)
case = *sub-addr*
 then p-sub-addr-okp (*ins*, *p*)

otherwise p-ins-okp2 (*ins*, *p*) **endcase**

THEOREM: p-ins-okp-is-p-ins-okp1
p-ins-okp (*ins*, *p*) = p-ins-okp1 (*ins*, *p*)

EVENT: Disable p-ins-okp.

EVENT: Disable p-ins-okp-is-p-ins-okp1.

DEFINITION:
p-ins-step2 (*ins*, *p*)
= **case on** car (*ins*):
case = *eq*
then p-eq-step (*ins*, *p*)
case = *lt-addr*
then p-lt-addr-step (*ins*, *p*)
case = *fetch*
then p-fetch-step (*ins*, *p*)
case = *deposit*
then p-deposit-step (*ins*, *p*)
case = *add-int*
then p-add-int-step (*ins*, *p*)
case = *add-int-with-carry*
then p-add-int-with-carry-step (*ins*, *p*)
case = *add1-int*
then p-add1-int-step (*ins*, *p*)
case = *sub-int*
then p-sub-int-step (*ins*, *p*)
case = *sub-int-with-carry*
then p-sub-int-with-carry-step (*ins*, *p*)
case = *sub1-int*
then p-sub1-int-step (*ins*, *p*)
case = *neg-int*
then p-neg-int-step (*ins*, *p*)
case = *lt-int*
then p-lt-int-step (*ins*, *p*)
case = *int-to-nat*
then p-int-to-nat-step (*ins*, *p*)
case = *add-nat*
then p-add-nat-step (*ins*, *p*)
case = *add-nat-with-carry*
then p-add-nat-with-carry-step (*ins*, *p*)
case = *add1-nat*

then p-add1-nat-step (*ins*, *p*)
case = *sub-nat*
then p-sub-nat-step (*ins*, *p*)
case = *sub-nat-with-carry*
then p-sub-nat-with-carry-step (*ins*, *p*)
case = *sub1-nat*
then p-sub1-nat-step (*ins*, *p*)
case = *lt-nat*
then p-lt-nat-step (*ins*, *p*)
case = *mult2-nat*
then p-mult2-nat-step (*ins*, *p*)
case = *mult2-nat-with-carry-out*
then p-mult2-nat-with-carry-out-step (*ins*, *p*)
case = *div2-nat*
then p-div2-nat-step (*ins*, *p*)
case = *or-bitv*
then p-or-bitv-step (*ins*, *p*)
case = *and-bitv*
then p-and-bitv-step (*ins*, *p*)
case = *not-bitv*
then p-not-bitv-step (*ins*, *p*)
case = *xor-bitv*
then p-xor-bitv-step (*ins*, *p*)
case = *rsh-bitv*
then p-rsh-bitv-step (*ins*, *p*)
case = *lsh-bitv*
then p-lsh-bitv-step (*ins*, *p*)
case = *or-bool*
then p-or-bool-step (*ins*, *p*)
case = *and-bool*
then p-and-bool-step (*ins*, *p*)
case = *not-bool*
then p-not-bool-step (*ins*, *p*)
otherwise p-halt (*p*, `'run`) **endcase**

DEFINITION:
p-ins-step1 (*ins*, *p*)
= **case on** car (*ins*):
case = *call*
then p-call-step (*ins*, *p*)
case = *ret*
then p-ret-step (*ins*, *p*)
case = *locn*
then p-locn-step (*ins*, *p*)

case = *push-constant*
 then p-push-constant-step (*ins*, *p*)
case = *push-local*
 then p-push-local-step (*ins*, *p*)
case = *push-global*
 then p-push-global-step (*ins*, *p*)
case = *push-ctrl-stk-free-size*
 then p-push-ctrl-stk-free-size-step (*ins*, *p*)
case = *push-temp-stk-free-size*
 then p-push-temp-stk-free-size-step (*ins*, *p*)
case = *push-temp-stk-index*
 then p-push-temp-stk-index-step (*ins*, *p*)
case = *jump-if-temp-stk-full*
 then p-jump-if-temp-stk-full-step (*ins*, *p*)
case = *jump-if-temp-stk-empty*
 then p-jump-if-temp-stk-empty-step (*ins*, *p*)
case = *pop*
 then p-pop-step (*ins*, *p*)
case = *pop**
 then p-pop*-step (*ins*, *p*)
case = *popn*
 then p-popn-step (*ins*, *p*)
case = *pop-local*
 then p-pop-local-step (*ins*, *p*)
case = *pop-global*
 then p-pop-global-step (*ins*, *p*)
case = *pop-locn*
 then p-pop-locn-step (*ins*, *p*)
case = *pop-call*
 then p-pop-call-step (*ins*, *p*)
case = *fetch-temp-stk*
 then p-fetch-temp-stk-step (*ins*, *p*)
case = *deposit-temp-stk*
 then p-deposit-temp-stk-step (*ins*, *p*)
case = *jump*
 then p-jump-step (*ins*, *p*)
case = *jump-case*
 then p-jump-case-step (*ins*, *p*)
case = *pushj*
 then p-pushj-step (*ins*, *p*)
case = *popj*
 then p-popj-step (*ins*, *p*)
case = *set-local*
 then p-set-local-step (*ins*, *p*)

case = *set-global*
 then p-set-global-step (*ins*, *p*)
case = *test-nat-and-jump*
 then p-test-nat-and-jump-step (*ins*, *p*)
case = *test-int-and-jump*
 then p-test-int-and-jump-step (*ins*, *p*)
case = *test-bool-and-jump*
 then p-test-bool-and-jump-step (*ins*, *p*)
case = *test-bitv-and-jump*
 then p-test-bitv-and-jump-step (*ins*, *p*)
case = *no-op*
 then p-no-op-step (*ins*, *p*)
case = *add-addr*
 then p-add-addr-step (*ins*, *p*)
case = *sub-addr*
 then p-sub-addr-step (*ins*, *p*)
otherwise p-ins-step2 (*ins*, *p*) **endcase**

THEOREM: p-ins-step-is-p-ins-step1
p-ins-step (*ins*, *p*) = p-ins-step1 (*ins*, *p*)

EVENT: Disable p-ins-step.

EVENT: Disable p-ins-step-is-p-ins-step1.

```
; We disable p-ins-okp because the main owc theorem involves p-ins-okp
; and p-ins-step, and opening either is sufficient to drive the case
; analysis.  If both open, then we get cross-multiplied cases and
; there is a lot of silly propositional work to do.  So we arbitrarily
; choose to keep p-ins-okp disabled.  Now we could choose to rewrite
; it to p-xxx-okp when the car of ins is 'xxx, analogously to what we
; did for r-ins-okp above.  But the proof of the owc should be faster
; if we do this: keep p-ins-okp disabled and show that p-xxx-okp holds
; if the car of ins is 'xxx.  The reason is that when we fire the
; xxx-one-way-correspondence-p-r lemma we will want to get p-xxx-okp
; and we'll be governed by a p-ins-okp and a hyp about car ins.  But
; the hyp about car ins won't have been raised all the way to the top
; and so won't yet be available to tell the p-ins-okp which way to go.
; This is the same strategy used for r-ins-okp in the r-i level proof.
```

THEOREM: p-ins-okp2-backchainer
p-ins-okp2 (*ins*, *p*)
→ (((car (*ins*) = 'eq) → p-eq-okp (*ins*, *p*))

$\wedge$ ((car (*ins*) = 'lt-addr) → p-lt-addr-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'fetch) → p-fetch-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'deposit) → p-deposit-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'add-int) → p-add-int-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'add-int-with-carry)
→ p-add-int-with-carry-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'add1-int) → p-add1-int-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'sub-int) → p-sub-int-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'sub-int-with-carry)
→ p-sub-int-with-carry-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'sub1-int) → p-sub1-int-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'neg-int) → p-neg-int-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'lt-int) → p-lt-int-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'int-to-nat) → p-int-to-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'add-nat) → p-add-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'add-nat-with-carry)
→ p-add-nat-with-carry-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'add1-nat) → p-add1-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'sub-nat) → p-sub-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'sub-nat-with-carry)
→ p-sub-nat-with-carry-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'sub1-nat) → p-sub1-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'lt-nat) → p-lt-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'mult2-nat) → p-mult2-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'mult2-nat-with-carry-out)
→ p-mult2-nat-with-carry-out-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'div2-nat) → p-div2-nat-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'or-bitv) → p-or-bitv-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'and-bitv) → p-and-bitv-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'not-bitv) → p-not-bitv-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'xor-bitv) → p-xor-bitv-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'rsh-bitv) → p-rsh-bitv-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'lsh-bitv) → p-lsh-bitv-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'or-bool) → p-or-bool-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'and-bool) → p-and-bool-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'not-bool) → p-not-bool-okp (*ins*, *p*)))

EVENT: Disable p-ins-okp2.

THEOREM: p-ins-okp-backchainer
p-ins-okp (*ins*, *p*)
→ (((car (*ins*) = 'call) → p-call-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'ret) → p-ret-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = 'locn) → p-locn-okp (*ins*, *p*))

$\wedge$ ((car (*ins*) = '`push-constant`)
→ p-push-constant-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`push-local`) → p-push-local-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`push-global`) → p-push-global-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`push-ctrl-stk-free-size`)
→ p-push-ctrl-stk-free-size-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`push-temp-stk-free-size`)
→ p-push-temp-stk-free-size-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`push-temp-stk-index`)
→ p-push-temp-stk-index-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`jump-if-temp-stk-full`)
→ p-jump-if-temp-stk-full-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`jump-if-temp-stk-empty`)
→ p-jump-if-temp-stk-empty-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`pop`) → p-pop-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`pop*`) → p-pop*-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`popn`) → p-popn-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`pop-local`) → p-pop-local-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`pop-global`) → p-pop-global-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`pop-locn`) → p-pop-locn-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`pop-call`) → p-pop-call-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`fetch-temp-stk`)
→ p-fetch-temp-stk-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`deposit-temp-stk`)
→ p-deposit-temp-stk-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`jump`) → p-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`jump-case`) → p-jump-case-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`pushj`) → p-pushj-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`popj`) → p-popj-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`set-local`) → p-set-local-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`set-global`) → p-set-global-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`test-nat-and-jump`)
→ p-test-nat-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`test-int-and-jump`)
→ p-test-int-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`test-bool-and-jump`)
→ p-test-bool-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`test-bitv-and-jump`)
→ p-test-bitv-and-jump-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`no-op`) → p-no-op-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`add-addr`) → p-add-addr-okp (*ins*, *p*))
$\wedge$ ((car (*ins*) = '`sub-addr`) → p-sub-addr-okp (*ins*, *p*)))

```
; The only problem with these backchainers is that we don't know that the
```

```
; above cases are exhaustive.  That is, suppose the current instruction
; is none of the above.  Then p-ins-okp is false.  We need to know that.
; The most natural expression of this fact is that if the instruction
; is not call and not ret and, etc., then p-ins-okp is false.  But that
; is an inefficient way to hang it, because every time we see p-ins-okp
; we'll try to show the instruction is none of the above and most of the time
; it will be one of the ones listed half-way through the list.  So we'll
; hang it on one of the instructions!  It reads: if p-ins-okp is true
; and the instruction is not call, not ret, etc., then the instruction
; is 'not-bool, except we split it into two lemmas.
```

THEOREM: p-ins-okp-exhausted
(p-ins-okp (*ins*, *p*)
 ∧ (car (*ins*) ≠ 'call)
 ∧ (car (*ins*) ≠ 'ret)
 ∧ (car (*ins*) ≠ 'locn)
 ∧ (car (*ins*) ≠ 'push-constant)
 ∧ (car (*ins*) ≠ 'push-local)
 ∧ (car (*ins*) ≠ 'push-global)
 ∧ (car (*ins*) ≠ 'push-ctrl-stk-free-size)
 ∧ (car (*ins*) ≠ 'push-temp-stk-free-size)
 ∧ (car (*ins*) ≠ 'push-temp-stk-index)
 ∧ (car (*ins*) ≠ 'jump-if-temp-stk-full)
 ∧ (car (*ins*) ≠ 'jump-if-temp-stk-empty)
 ∧ (car (*ins*) ≠ 'pop)
 ∧ (car (*ins*) ≠ 'pop*)
 ∧ (car (*ins*) ≠ 'popn)
 ∧ (car (*ins*) ≠ 'pop-local)
 ∧ (car (*ins*) ≠ 'pop-global)
 ∧ (car (*ins*) ≠ 'pop-locn)
 ∧ (car (*ins*) ≠ 'pop-call)
 ∧ (car (*ins*) ≠ 'fetch-temp-stk)
 ∧ (car (*ins*) ≠ 'deposit-temp-stk)
 ∧ (car (*ins*) ≠ 'jump)
 ∧ (car (*ins*) ≠ 'jump-case)
 ∧ (car (*ins*) ≠ 'pushj)
 ∧ (car (*ins*) ≠ 'popj)
 ∧ (car (*ins*) ≠ 'set-local)
 ∧ (car (*ins*) ≠ 'set-global)
 ∧ (car (*ins*) ≠ 'test-nat-and-jump)
 ∧ (car (*ins*) ≠ 'test-int-and-jump)
 ∧ (car (*ins*) ≠ 'test-bool-and-jump)
 ∧ (car (*ins*) ≠ 'test-bitv-and-jump)

$\wedge$ (car (*ins*) $\neq$ 'no-op)
$\wedge$ (car (*ins*) $\neq$ 'add-addr)
$\wedge$ (car (*ins*) $\neq$ 'sub-addr))
$\rightarrow$ p-ins-okp2 (*ins*, *p*)

THEOREM: p-ins-okp2-exhausted
(p-ins-okp2 (*ins*, *p*)
$\wedge$ (car (*ins*) $\neq$ 'eq)
$\wedge$ (car (*ins*) $\neq$ 'lt-addr)
$\wedge$ (car (*ins*) $\neq$ 'fetch)
$\wedge$ (car (*ins*) $\neq$ 'deposit)
$\wedge$ (car (*ins*) $\neq$ 'add-int)
$\wedge$ (car (*ins*) $\neq$ 'add-int-with-carry)
$\wedge$ (car (*ins*) $\neq$ 'add1-int)
$\wedge$ (car (*ins*) $\neq$ 'sub-int)
$\wedge$ (car (*ins*) $\neq$ 'sub-int-with-carry)
$\wedge$ (car (*ins*) $\neq$ 'sub1-int)
$\wedge$ (car (*ins*) $\neq$ 'neg-int)
$\wedge$ (car (*ins*) $\neq$ 'lt-int)
$\wedge$ (car (*ins*) $\neq$ 'int-to-nat)
$\wedge$ (car (*ins*) $\neq$ 'add-nat)
$\wedge$ (car (*ins*) $\neq$ 'add-nat-with-carry)
$\wedge$ (car (*ins*) $\neq$ 'add1-nat)
$\wedge$ (car (*ins*) $\neq$ 'sub-nat)
$\wedge$ (car (*ins*) $\neq$ 'sub-nat-with-carry)
$\wedge$ (car (*ins*) $\neq$ 'sub1-nat)
$\wedge$ (car (*ins*) $\neq$ 'lt-nat)
$\wedge$ (car (*ins*) $\neq$ 'mult2-nat)
$\wedge$ (car (*ins*) $\neq$ 'mult2-nat-with-carry-out)
$\wedge$ (car (*ins*) $\neq$ 'div2-nat)
$\wedge$ (car (*ins*) $\neq$ 'or-bitv)
$\wedge$ (car (*ins*) $\neq$ 'and-bitv)
$\wedge$ (car (*ins*) $\neq$ 'not-bitv)
$\wedge$ (car (*ins*) $\neq$ 'xor-bitv)
$\wedge$ (car (*ins*) $\neq$ 'rsh-bitv)
$\wedge$ (car (*ins*) $\neq$ 'lsh-bitv)
$\wedge$ (car (*ins*) $\neq$ 'or-bool)
$\wedge$ (car (*ins*) $\neq$ 'and-bool))
$\rightarrow$ ((car (*ins*) = 'not-bool) = **t**)

EVENT: Disable p-ins-step2.

THEOREM: transitivity-of-same-signature
(same-signature (*segment1*, *segment2*) $\wedge$ same-signature (*segment2*, *segment3*))

$\rightarrow$ same-signature (*segment1*, *segment3*)

```
;; ***** End stuff from p-r.events *****

;; The next two lemmas are from big-add.events by J Moore.
```

THEOREM: p-opener
(p (*s*, 0) = *s*)
$\wedge$ (p (p-state (*pc*, *ctrl*, *temp*, *prog*, *data*, *max-ctrl*, *max-temp*, *word-size*, *psw*),
1 + *n*)
= p (p-step (p-state (*pc*,
ctrl,
temp,
prog,
data,
max-ctrl,
max-temp,
word-size,
psw)),
n))

EVENT: Disable p-opener.

EVENT: Disable p.

THEOREM: p-step1-opener
p-step1 (cons (*opcode*, *operands*), *p*)
= **if** p-ins-okp (cons (*opcode*, *operands*), *p*)
then p-ins-step (cons (*opcode*, *operands*), *p*)
else p-halt (*p*, x-y-error-msg ('p, *opcode*)) **endif**

EVENT: Disable p-step1-opener.

EVENT: Disable p-step1.

```
;; ***** End stuff from big-add.events *****
```

EVENT: Disable same-signature-implies-equal-lengths.

EVENT: Disable same-signature-implies-equal-definedp.

EVENT: Disable transitivity-of-same-signature.

```
;; The following axioms are proven in Piton (in p-r).
```

AXIOM: p-step-preserves-proper-p-statep
(proper-p-statep (*p*) ∧ (¬ errorp (p-psw (p-step (*p*)))))
→ proper-p-statep (p-step (*p*))

AXIOM: once-errorp-always-errorp-step
(¬ errorp (p-psw (p-step (*p*)))) → (¬ errorp (p-psw (*p*)))

AXIOM: once-errorp-always-errorp
(¬ errorp (p-psw (p (*p*, *n*)))) → (¬ errorp (p-psw (*p*)))

AXIOM: p-preserves-proper-p-statep
(proper-p-statep (*p*) ∧ (¬ errorp (p-psw (p (*p*, *n*)))))
→ proper-p-statep (p (*p*, *n*))

```
;; ***** End axioms proven in p-r.events *****
```

```
;; The following lemmas are NOT proven in Piton.
```

DEFINITION:
p-invariant1 (*p0*, *p1*)
= ((p-prog-segment (*p0*) = p-prog-segment (*p1*))
∧ (p-max-ctrl-stk-size (*p0*) = p-max-ctrl-stk-size (*p1*))
∧ (p-max-temp-stk-size (*p0*) = p-max-temp-stk-size (*p1*))
∧ (p-word-size (*p0*) = p-word-size (*p1*)))

DEFINITION:
p-invariant (*p0*, *p1*)
= (same-signature (p-data-segment (*p0*), p-data-segment (*p1*))
∧ p-invariant1 (*p0*, *p1*))

THEOREM: p-invariant-opener
p-invariant (p-state (*pc*,
ctrl-stk,
temp-stk,
p-prog-segment (*p*),
data-seg,
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
psw),
p)
= same-signature (*data-seg*, p-data-segment (*p*))

THEOREM: p-invariant1-opener
p-invariant1 (p-state (*pc*,
ctrl-stk,
temp-stk,
p-prog-segment (*p*),
data-seg,
p-max-ctrl-stk-size (*p*),
p-max-temp-stk-size (*p*),
p-word-size (*p*),
psw),
p)

EVENT: Disable p-invariant.

EVENT: Disable p-invariant1.

```
;; It will be easier later to no that the p-prog-segment and the 3 resource
;; limitations p-max-ctrl-stk-size, p-max-temp-stk-size and p-word-size
;; never change.  It is nice to not have to check the run flag.  We could
;; probably prove that if we have a PROPER-P-STATEP that these are invariant
;; but we get nicer rewrite rules later (at the lr level anyway) if we
;; do not have to assume that.
```

THEOREM: no-op-preserves-p-invariant
(car (*ins*) = 'no-op) → p-invariant (p-no-op-step (*ins*, *p*), *p*)

THEOREM: push-global-preserves-p-invariant
(car (*ins*) = 'push-global)
→ p-invariant (p-push-global-step (*ins*, *p*), *p*)

```
;; The following is inspired by the lemma length-put of Piton.
```

THEOREM: my-length-put
length (put (*val*, *n*, *lst*))
= **if** *n* < length (*lst*) **then** length (*lst*)
else 1 + *n* **endif**

EVENT: Disable my-length-put.

THEOREM: proper-p-statep-proper-implies-proper-p-instructionp
proper-p-statep (*p*)
→ proper-p-instructionp (p-current-instruction (*p*), car (untag (p-pc (*p*))), *p*)

THEOREM: definedp-p-data-segment-implies-listp-generalized
(proper-p-data-segmentp (*segment*, *p*) ∧ definedp (*name*, *segment*))
→ listp (cdr (assoc (*name*, *segment*)))

```
;; The lemma definedp-p-data-segment-implies-listp in p-r was definedp with
;; proper-p-statep1, which we don't use, so and a '-1' [so there are not
;; name conflicts] to the name and use proper-p-statep.  AF
```

THEOREM: definedp-p-data-segment-implies-listp-1
(proper-p-statep (*p*) ∧ definedp (*name*, p-data-segment (*p*)))
→ listp (cdr (assoc (*name*, p-data-segment (*p*))))

EVENT: Disable definedp-p-data-segment-implies-listp-1.

THEOREM: definedp-p-data-segment-implies-litatom-generalized
(proper-p-data-segmentp (*segment*, *p*) ∧ definedp (*name*, *segment*))
→ litatom (*name*)

THEOREM: definedp-p-data-segment-implies-litatom
(proper-p-statep (*p*) ∧ definedp (*name*, p-data-segment (*p*))) → litatom (*name*)

THEOREM: pop-global-preserves-p-invariant1
(car (*ins*) = 'pop-global) → p-invariant1 (p-pop-global-step (*ins*, *p*), *p*)

THEOREM: pop-global-preserves-p-same-signature-data-segment
let *new-p* **be** p-pop-global-step (p-current-instruction (*p*), *p*)
in
((p-psw (*p*) = 'run)
∧ (car (p-current-instruction (*p*)) = 'pop-global)
∧ proper-p-statep (*p*)
∧ p-pop-global-okp (p-current-instruction (*p*), *p*))
→ same-signature (p-data-segment (*new-p*), p-data-segment (*p*)) **endlet**

THEOREM: push-local-preserves-p-invariant
(car (*ins*) = 'push-local) → p-invariant (p-push-local-step (*ins*, *p*), *p*)

THEOREM: pop-local-preserves-p-invariant
(car (*ins*) = 'pop-local) → p-invariant (p-pop-local-step (*ins*, *p*), *p*)

THEOREM: deposit-preserves-p-invariant1
(car (*ins*) = 'deposit) → p-invariant1 (p-deposit-step (*ins*, *p*), *p*)

THEOREM: deposit-preserves-same-signature-data-segment
let *new-p* **be** p-deposit-step (p-current-instruction (*p*), *p*)
in

((p-psw (p) = 'run)
 ∧ (car (p-current-instruction (p)) = 'deposit)
 ∧ proper-p-statep (p)
 ∧ p-deposit-okp (p-current-instruction (p), p))
→ same-signature (p-data-segment (*new-p*), p-data-segment (p)) **endlet**

THEOREM: add1-nat-preserves-p-invariant
(car (ins) = 'add1-nat) → p-invariant (p-add1-nat-step (ins, p), p)

THEOREM: add-addr-preserves-p-invariant
(car (ins) = 'add-addr) → p-invariant (p-add-addr-step (ins, p), p)

THEOREM: add-nat-with-carry-preserves-p-invariant
(car (ins) = 'add-nat-with-carry)
→ p-invariant (p-add-nat-with-carry-step (ins, p), p)

THEOREM: mult2-nat-with-carry-out-preserves-p-invariant
(car (ins) = 'mult2-nat-with-carry-out)
→ p-invariant (p-mult2-nat-with-carry-out-step (ins, p), p)

THEOREM: fetch-temp-stk-preserves-p-invariant
(car (ins) = 'fetch-temp-stk)
→ p-invariant (p-fetch-temp-stk-step (ins, p), p)

THEOREM: deposit-temp-stk-preserves-p-invariant
(car (ins) = 'deposit-temp-stk)
→ p-invariant (p-deposit-temp-stk-step (ins, p), p)

THEOREM: pop*-preserves-p-invariant
(car (ins) = 'pop*) → p-invariant (p-pop*-step (ins, p), p)

THEOREM: popn-preserves-p-invariant
(car (ins) = 'popn) → p-invariant (p-popn-step (ins, p), p)

```
; For what it's worth, this concludes the so-called "test section"
; of instructions identified as interesting in the case of the
; one-way-correspondence proof.
```

THEOREM: set-local-preserves-p-invariant
(car (ins) = 'set-local) → p-invariant (p-set-local-step (ins, p), p)

THEOREM: set-global-preserves-p-invariant1
(car (ins) = 'set-global) → p-invariant1 (p-set-global-step (ins, p), p)

THEOREM: set-global-preserves-same-signature-data-segment
let *new-p* **be** p-set-global-step (p-current-instruction (*p*), *p*)
in
((p-psw (*p*) = 'run)
∧ (car (p-current-instruction (*p*)) = 'set-global)
∧ proper-p-statep (*p*)
∧ p-set-global-okp (p-current-instruction (*p*), *p*))
→ same-signature (p-data-segment (*new-p*), p-data-segment (*p*)) **endlet**

THEOREM: push-constant-preserves-p-invariant
(car (*ins*) = 'push-constant)
→ p-invariant (p-push-constant-step (*ins*, *p*), *p*)

THEOREM: push-ctrl-stk-free-size-preserves-p-invariant
(car (*ins*) = 'push-ctrl-stk-free-size)
→ p-invariant (p-push-ctrl-stk-free-size-step (*ins*, *p*), *p*)

THEOREM: push-temp-stk-free-size-preserves-p-invariant
(car (*ins*) = 'push-temp-stk-free-size)
→ p-invariant (p-push-temp-stk-free-size-step (*ins*, *p*), *p*)

THEOREM: push-temp-stk-index-preserves-p-invariant
(car (*ins*) = 'push-temp-stk-index)
→ p-invariant (p-push-temp-stk-index-step (*ins*, *p*), *p*)

THEOREM: pop-preserves-p-invariant
(car (*ins*) = 'pop) → p-invariant (p-pop-step (*ins*, *p*), *p*)

THEOREM: jump-preserves-p-invariant
(car (*ins*) = 'jump) → p-invariant (p-jump-step (*ins*, *p*), *p*)

THEOREM: pushj-preserves-p-invariant
(car (*ins*) = 'pushj) → p-invariant (p-pushj-step (*ins*, *p*), *p*)

THEOREM: popj-preserves-p-invariant
(car (*ins*) = 'popj) → p-invariant (p-popj-step (*ins*, *p*), *p*)

THEOREM: sub-addr-preserves-p-invariant
(car (*ins*) = 'sub-addr) → p-invariant (p-sub-addr-step (*ins*, *p*), *p*)

THEOREM: eq-preserves-p-invariant
(car (*ins*) = 'eq) → p-invariant (p-eq-step (*ins*, *p*), *p*)

THEOREM: lt-addr-preserves-p-invariant
(car (*ins*) = 'lt-addr) → p-invariant (p-lt-addr-step (*ins*, *p*), *p*)

THEOREM: fetch-preserves-p-invariant
(car (*ins*) = `'fetch`) → p-invariant (p-fetch-step (*ins*, *p*), *p*)

THEOREM: add-int-preserves-p-invariant
(car (*ins*) = `'add-int`) → p-invariant (p-add-int-step (*ins*, *p*), *p*)

THEOREM: add-int-with-carry-preserves-p-invariant
(car (*ins*) = `'add-int-with-carry`)
→ p-invariant (p-add-int-with-carry-step (*ins*, *p*), *p*)

THEOREM: add1-int-preserves-p-invariant
(car (*ins*) = `'add1-int`) → p-invariant (p-add1-int-step (*ins*, *p*), *p*)

THEOREM: sub-int-preserves-p-invariant
(car (*ins*) = `'sub-int`) → p-invariant (p-sub-int-step (*ins*, *p*), *p*)

THEOREM: sub-int-with-carry-preserves-p-invariant
(car (*ins*) = `'sub-int-with-carry`)
→ p-invariant (p-sub-int-with-carry-step (*ins*, *p*), *p*)

THEOREM: sub1-int-preserves-p-invariant
(car (*ins*) = `'sub1-int`) → p-invariant (p-sub1-int-step (*ins*, *p*), *p*)

THEOREM: neg-int-preserves-p-invariant
(car (*ins*) = `'neg-int`) → p-invariant (p-neg-int-step (*ins*, *p*), *p*)

THEOREM: lt-int-preserves-p-invariant
(car (*ins*) = `'lt-int`) → p-invariant (p-lt-int-step (*ins*, *p*), *p*)

THEOREM: int-to-nat-preserves-p-invariant
(car (*ins*) = `'int-to-nat`) → p-invariant (p-int-to-nat-step (*ins*, *p*), *p*)

THEOREM: add-nat-preserves-p-invariant
(car (*ins*) = `'add-nat`) → p-invariant (p-add-nat-step (*ins*, *p*), *p*)

THEOREM: sub-nat-preserves-p-invariant
(car (*ins*) = `'sub-nat`) → p-invariant (p-sub-nat-step (*ins*, *p*), *p*)

THEOREM: sub-nat-with-carry-preserves-p-invariant
(car (*ins*) = `'sub-nat-with-carry`)
→ p-invariant (p-sub-nat-with-carry-step (*ins*, *p*), *p*)

THEOREM: sub1-nat-preserves-p-invariant
(car (*ins*) = `'sub1-nat`) → p-invariant (p-sub1-nat-step (*ins*, *p*), *p*)

THEOREM: lt-nat-preserves-p-invariant
(car (*ins*) = `'lt-nat`) → p-invariant (p-lt-nat-step (*ins*, *p*), *p*)

THEOREM: mult2-nat-preserves-p-invariant
(car (*ins*) = `'mult2-nat`) → p-invariant (p-mult2-nat-step (*ins*, *p*), *p*)

THEOREM: div2-nat-preserves-p-invariant
(car (*ins*) = `'div2-nat`) → p-invariant (p-div2-nat-step (*ins*, *p*), *p*)

THEOREM: or-bitv-preserves-p-invariant
(car (*ins*) = `'or-bitv`) → p-invariant (p-or-bitv-step (*ins*, *p*), *p*)

THEOREM: and-bitv-preserves-p-invariant
(car (*ins*) = `'and-bitv`) → p-invariant (p-and-bitv-step (*ins*, *p*), *p*)

THEOREM: not-bitv-preserves-p-invariant
(car (*ins*) = `'not-bitv`) → p-invariant (p-not-bitv-step (*ins*, *p*), *p*)

THEOREM: xor-bitv-preserves-p-invariant
(car (*ins*) = `'xor-bitv`) → p-invariant (p-xor-bitv-step (*ins*, *p*), *p*)

THEOREM: rsh-bitv-preserves-p-invariant
(car (*ins*) = `'rsh-bitv`) → p-invariant (p-rsh-bitv-step (*ins*, *p*), *p*)

THEOREM: lsh-bitv-preserves-p-invariant
(car (*ins*) = `'lsh-bitv`) → p-invariant (p-lsh-bitv-step (*ins*, *p*), *p*)

THEOREM: or-bool-preserves-p-invariant
(car (*ins*) = `'or-bool`) → p-invariant (p-or-bool-step (*ins*, *p*), *p*)

THEOREM: and-bool-preserves-p-invariant
(car (*ins*) = `'and-bool`) → p-invariant (p-and-bool-step (*ins*, *p*), *p*)

THEOREM: not-bool-preserves-p-invariant
(car (*ins*) = `'not-bool`) → p-invariant (p-not-bool-step (*ins*, *p*), *p*)

THEOREM: jump-if-temp-stk-full-preserves-p-invariant
(car (*ins*) = `'jump-if-temp-stk-full`)
→ p-invariant (p-jump-if-temp-stk-full-step (*ins*, *p*), *p*)

THEOREM: jump-if-temp-stk-empty-preserves-p-invariant
(car (*ins*) = `'jump-if-temp-stk-empty`)
→ p-invariant (p-jump-if-temp-stk-empty-step (*ins*, *p*), *p*)

THEOREM: test-nat-and-jump-preserves-p-invariant
(car (*ins*) = `'test-nat-and-jump`)
→ p-invariant (p-test-nat-and-jump-step (*ins*, *p*), *p*)

THEOREM: test-int-and-jump-preserves-p-invariant
(car (*ins*) = `'test-int-and-jump`)
→ p-invariant (p-test-int-and-jump-step (*ins*, *p*), *p*)

THEOREM: test-bool-and-jump-preserves-p-invariant
(car (*ins*) = `'test-bool-and-jump`)
→ p-invariant (p-test-bool-and-jump-step (*ins*, *p*), *p*)

THEOREM: test-bitv-and-jump-preserves-p-invariant
(car (*ins*) = `'test-bitv-and-jump`)
→ p-invariant (p-test-bitv-and-jump-step (*ins*, *p*), *p*)

THEOREM: pop-locn-preserves-p-invariant
(car (*ins*) = `'pop-locn`) → p-invariant (p-pop-locn-step (*ins*, *p*), *p*)

THEOREM: locn-preserves-p-invariant
(car (*ins*) = `'locn`) → p-invariant (p-locn-step (*ins*, *p*), *p*)

THEOREM: jump-case-preserves-p-invariant
(car (*ins*) = `'jump-case`) → p-invariant (p-jump-case-step (*ins*, *p*), *p*)

THEOREM: call-preserves-p-invariant
(car (*ins*) = `'call`) → p-invariant (p-call-step (*ins*, *p*), *p*)

THEOREM: pop-call-preserves-p-invariant
(car (*ins*) = `'pop-call`) → p-invariant (p-pop-call-step (*ins*, *p*), *p*)

THEOREM: ret-preserves-p-invariant
(car (*ins*) = `'ret`) → p-invariant (p-ret-step (*ins*, *p*), *p*)

THEOREM: p-invariant-p-invariant1
p-invariant (*p0*, *p1*) → p-invariant1 (*p0*, *p1*)

THEOREM: p-ins-step2-preserves-p-invariant1
p-invariant1 (p-ins-step2 (*ins*, *p*), *p*)

THEOREM: p-ins-step-preserves-p-invariant1
p-invariant1 (p-ins-step (*ins*, *p*), *p*)

EVENT: Disable p-invariant-p-invariant1.

THEOREM: p-invariant1-reflexive
p-invariant1 (*p*, *p*)

THEOREM: p-invariant1-p-halt
(p-invariant1 (*p0*, p-halt (*p1*, *error-message*)) = p-invariant1 (*p0*, *p1*))
∧ (p-invariant1 (p-halt (*p0*, *error-message*), *p1*) = p-invariant1 (*p0*, *p1*))

THEOREM: p-step-preserves-p-invariant1
p-invariant1 (p-step (*p*), *p*)

THEOREM: transitivity-of-p-invariant1
(p-invariant1 (*p0*, *p1*) ∧ p-invariant1 (*p1*, *p2*)) → p-invariant1 (*p0*, *p2*)

THEOREM: p-preserves-p-invariant1
p-invariant1 (p (*p*, *n*), *p*)

EVENT: Disable transitivity-of-p-invariant1.

THEOREM: p-preserves-p-resources
(p-prog-segment (p (*p*, *n*)) = p-prog-segment (*p*))
∧ (p-max-ctrl-stk-size (p (*p*, *n*)) = p-max-ctrl-stk-size (*p*))
∧ (p-max-temp-stk-size (p (*p*, *n*)) = p-max-temp-stk-size (*p*))
∧ (p-word-size (p (*p*, *n*)) = p-word-size (*p*))

THEOREM: p-invariant-same-signature-data-segments
p-invariant (*p0*, *p1*)
→ same-signature (p-data-segment (*p0*), p-data-segment (*p1*))

THEOREM: p-ins-step2-preserves-same-signature-data-segment
let *new-p* **be** p-ins-step2 (p-current-instruction (*p*), *p*)
in
((p-psw (*p*) = 'run)
∧ p-ins-okp2 (p-current-instruction (*p*), *p*)
∧ proper-p-statep (*p*))
→ same-signature (p-data-segment (*new-p*), p-data-segment (*p*)) **endlet**

THEOREM: p-ins-step-preserves-same-signature-data-segment
((p-psw (*p*) = 'run)
∧ p-ins-okp (p-current-instruction (*p*), *p*)
∧ proper-p-statep (*p*))
→ same-signature (p-data-segment (p-ins-step (p-current-instruction (*p*), *p*)),
p-data-segment (*p*))

EVENT: Disable p-invariant-same-signature-data-segments.

THEOREM: p-step-preserves-same-signature-data-segment
(proper-p-statep (*p*) ∧ (¬ errorp (p-psw (p-step (*p*)))))
→ same-signature (p-data-segment (p-step (*p*)), p-data-segment (*p*))

THEOREM: p-preserves-same-signature-data-segment
(proper-p-statep (*p*) ∧ (¬ errorp (p-psw (p (*p*, *n*)))))
→ same-signature (p-data-segment (p (*p*, *n*)), p-data-segment (*p*))

```
;; Finally enable the Piton -okp and -step functions, we want them for later.
```

EVENT: Enable p-call-okp.

EVENT: Enable p-call-step.

EVENT: Enable p-ret-okp.

EVENT: Enable p-ret-step.

EVENT: Enable p-locn-okp.

EVENT: Enable p-locn-step.

EVENT: Enable p-push-constant-okp.

EVENT: Enable p-push-constant-step.

EVENT: Enable p-push-local-okp.

EVENT: Enable p-push-local-step.

EVENT: Enable p-push-global-okp.

EVENT: Enable p-push-global-step.

EVENT: Enable p-push-ctrl-stk-free-size-okp.

EVENT: Enable p-push-ctrl-stk-free-size-step.

EVENT: Enable p-push-temp-stk-free-size-okp.

EVENT: Enable p-push-temp-stk-free-size-step.

EVENT: Enable p-push-temp-stk-index-okp.

EVENT: Enable p-push-temp-stk-index-step.

EVENT: Enable p-jump-if-temp-stk-full-okp.

EVENT: Enable p-jump-if-temp-stk-full-step.

EVENT: Enable p-jump-if-temp-stk-empty-okp.

EVENT: Enable p-jump-if-temp-stk-empty-step.

EVENT: Enable p-pop-okp.

EVENT: Enable p-pop-step.

EVENT: Enable p-pop*-okp.

EVENT: Enable p-pop*-step.

EVENT: Enable p-popn-okp.

EVENT: Enable p-popn-step.

EVENT: Enable p-pop-local-okp.

EVENT: Enable p-pop-local-step.

EVENT: Enable p-pop-global-okp.

EVENT: Enable p-pop-global-step.

EVENT: Enable p-pop-locn-okp.

EVENT: Enable p-pop-locn-step.

EVENT: Enable p-pop-call-okp.

EVENT: Enable p-pop-call-step.

EVENT: Enable p-fetch-temp-stk-okp.

EVENT: Enable p-fetch-temp-stk-step.

EVENT: Enable p-deposit-temp-stk-okp.

EVENT: Enable p-deposit-temp-stk-step.

EVENT: Enable p-jump-okp.

EVENT: Enable p-jump-step.

EVENT: Enable p-jump-case-okp.

EVENT: Enable p-jump-case-step.

EVENT: Enable p-pushj-okp.

EVENT: Enable p-pushj-step.

EVENT: Enable p-popj-okp.

EVENT: Enable p-popj-step.

EVENT: Enable p-set-local-okp.

EVENT: Enable p-set-local-step.

EVENT: Enable p-set-global-okp.

EVENT: Enable p-set-global-step.

EVENT: Enable p-test-nat-and-jump-okp.

EVENT: Enable p-test-nat-and-jump-step.

EVENT: Enable p-test-int-and-jump-okp.

EVENT: Enable p-test-int-and-jump-step.

EVENT: Enable p-test-bool-and-jump-okp.

EVENT: Enable p-test-bool-and-jump-step.

EVENT: Enable p-test-bitv-and-jump-okp.

EVENT: Enable p-test-bitv-and-jump-step.

EVENT: Enable p-no-op-okp.

EVENT: Enable p-no-op-step.

EVENT: Enable p-add-addr-okp.

EVENT: Enable p-add-addr-step.

EVENT: Enable p-sub-addr-okp.

EVENT: Enable p-sub-addr-step.

EVENT: Enable p-eq-okp.

EVENT: Enable p-eq-step.

EVENT: Enable p-lt-addr-okp.

EVENT: Enable p-lt-addr-step.

EVENT: Enable p-fetch-okp.

EVENT: Enable p-fetch-step.

EVENT: Enable p-deposit-okp.

EVENT: Enable p-deposit-step.

EVENT: Enable p-add-int-okp.

EVENT: Enable p-add-int-step.

EVENT: Enable p-add-int-with-carry-okp.

EVENT: Enable p-add-int-with-carry-step.

EVENT: Enable p-add1-int-okp.

EVENT: Enable p-add1-int-step.

EVENT: Enable p-sub-int-okp.

EVENT: Enable p-sub-int-step.

EVENT: Enable p-sub-int-with-carry-okp.

EVENT: Enable p-sub-int-with-carry-step.

EVENT: Enable p-sub1-int-okp.

EVENT: Enable p-sub1-int-step.

EVENT: Enable p-neg-int-okp.

EVENT: Enable p-neg-int-step.

EVENT: Enable p-lt-int-okp.

EVENT: Enable p-lt-int-step.

EVENT: Enable p-int-to-nat-okp.

EVENT: Enable p-int-to-nat-step.

EVENT: Enable p-add-nat-okp.

EVENT: Enable p-add-nat-step.

EVENT: Enable p-add-nat-with-carry-okp.

EVENT: Enable p-add-nat-with-carry-step.

EVENT: Enable p-add1-nat-okp.

EVENT: Enable p-add1-nat-step.

EVENT: Enable p-sub-nat-okp.

EVENT: Enable p-sub-nat-step.

EVENT: Enable p-sub-nat-with-carry-okp.

EVENT: Enable p-sub-nat-with-carry-step.

EVENT: Enable p-sub1-nat-okp.

EVENT: Enable p-sub1-nat-step.

EVENT: Enable p-lt-nat-okp.

EVENT: Enable p-lt-nat-step.

EVENT: Enable p-mult2-nat-okp.

EVENT: Enable p-mult2-nat-step.

EVENT: Enable p-mult2-nat-with-carry-out-okp.

EVENT: Enable p-mult2-nat-with-carry-out-step.

EVENT: Enable p-div2-nat-okp.

EVENT: Enable p-div2-nat-step.

EVENT: Enable p-or-bitv-okp.

EVENT: Enable p-or-bitv-step.

EVENT: Enable p-and-bitv-okp.

EVENT: Enable p-and-bitv-step.

EVENT: Enable p-not-bitv-okp.

EVENT: Enable p-not-bitv-step.

EVENT: Enable p-xor-bitv-okp.

EVENT: Enable p-xor-bitv-step.

EVENT: Enable p-rsh-bitv-okp.

EVENT: Enable p-rsh-bitv-step.

EVENT: Enable p-lsh-bitv-okp.

EVENT: Enable p-lsh-bitv-step.

EVENT: Enable p-or-bool-okp.

EVENT: Enable p-or-bool-step.

EVENT: Enable p-and-bool-okp.

EVENT: Enable p-and-bool-step.

EVENT: Enable p-not-bool-okp.

EVENT: Enable p-not-bool-step.

EVENT: Enable p-ins-step.

EVENT: Enable p-ins-okp.

```
; ------------------------------------------------------------
; was l-.events
; ------------------------------------------------------------

;;; This is a prototype compiler for the Logic.  Right now we only include
;;; CONS, CAR, CDR, LISTP and TRUE and TRUEP.  (Also Quote).

;; Functions to test compiler:
;; ------------------------------------------------------------------------------
```

DEFINITION:
app (*x*, *y*)
= **if** listp (*x*) **then** cons (car (*x*), app (cdr (*x*), *y*))
else *y* **endif**

DEFINITION:
rev (*x*)
= **if** listp (*x*) **then** app (rev (cdr (*x*)), list (car (*x*)))
else nil endif

DEFINITION:
frev (*x*, *y*)
= **if** listp (*x*) **then** frev (cdr (*x*), cons (car (*x*), *y*))
else *y* **endif**

```
;; ------------------------------------------------------------------------------
;; End functions for testing

;; Returns a list containing the value or F.
```

DEFINITION:
l-eval (*flag*, *expr*, *alist*, *clk*)
= **if** *flag* = `'list`
then if listp (*expr*)
then cons (l-eval (**t**, car (*expr*), *alist*, *clk*),
l-eval (`'list`, cdr (*expr*), *alist*, *clk*))
else nil endif
elseif *clk* ≃ `0` **then f**
elseif litatom (*expr*) **then** list (cdr (assoc (*expr*, *alist*)))
elseif *expr* ≃ **nil** **then** list (*expr*)
elseif car (*expr*) = `'quote` **then** list (cadr (*expr*))
elseif car (*expr*) = `'if`
then let *test* **be** l-eval (**t**, cadr (*expr*), *alist*, *clk*)
in
if *test*
then if car (*test*) **then** l-eval (**t**, caddr (*expr*), *alist*, *clk*)
else l-eval (**t**, cadddr (*expr*), *alist*, *clk*) **endif**
else f endif endlet
elseif f ∈ l-eval (`'list`, cdr (*expr*), *alist*, *clk*) **then f**
elseif subrp (car (*expr*))
then list (apply-subr (car (*expr*),
strip-cars (l-eval (`'list`, cdr (*expr*), *alist*, *clk*))))
else l-eval (**t**,
body (car (*expr*)),
pairlist (formals (car (*expr*)),
strip-cars (l-eval (`'list`,
cdr (*expr*),
alist,
clk))),
clk − 1) **endif**

DEFINITION:
remove-costs (*list*)
= **if** listp (*list*) **then** cons (list (caar (*list*)), remove-costs (cdr (*list*)))
else nil endif

THEOREM: strip-cars-remove-costs
strip-cars (remove-costs (*l*)) = strip-cars (*l*)

THEOREM: member-strip-cars-definedp
(*x* ∈ strip-cars (*y*)) = definedp (*x*, *y*)

EVENT: Disable member-strip-cars-definedp.

THEOREM: member-plist

$(x \in \text{plist}\,(y)) = (x \in y)$

THEOREM: car-assoc
$\text{definedp}\,(x,\ y) \rightarrow (\text{car}\,(\text{assoc}\,(x,\ y)) = x)$

THEOREM: v&c$-l-eval-equivalence
$$\begin{array}{ll}
(((\mathit{flag} = \texttt{'list}) \wedge (\mathbf{f} \notin \text{l-eval}\,(\mathit{flag},\ \mathit{expr},\ \mathit{alist},\ \mathit{clock})))\\
\vee & ((\mathit{flag} \neq \texttt{'list}) \wedge (\text{l-eval}\,(\mathit{flag},\ \mathit{expr},\ \mathit{alist},\ \mathit{clock}) \neq \mathbf{f})))\\
\rightarrow & ((\text{l-eval}\,(\mathit{flag},\ \mathit{expr},\ \mathit{alist},\ \mathit{clock})\\
& \quad = \quad \textbf{if}\ \mathit{flag} = \texttt{'list}\ \textbf{then}\ \text{remove-costs}\,(\text{v\&c\$}\,(\mathit{flag},\ \mathit{expr},\ \mathit{alist}))\\
& \qquad \textbf{else}\ \text{list}\,(\text{car}\,(\text{v\&c\$}\,(\mathit{flag},\ \mathit{expr},\ \mathit{alist})))\ \textbf{endif})\\
& \wedge \quad (((\mathit{flag} = \texttt{'list}) \wedge (\mathbf{f} \notin \text{v\&c\$}\,(\mathit{flag},\ \mathit{expr},\ \mathit{alist})))\\
& \qquad \vee \quad ((\mathit{flag} \neq \texttt{'list}) \wedge (\text{v\&c\$}\,(\mathit{flag},\ \mathit{expr},\ \mathit{alist}) \neq \mathbf{f}))))
\end{array}$$

THEOREM: cdr-v&c$-lessp-if-cadr
$$\begin{array}{ll}
& \text{v\&c-apply\$}\,(\texttt{'if},\ \text{cons}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}),\ \text{v\&c\$}\,(\texttt{'list},\ \mathit{branches},\ \mathit{alist})))\\
\rightarrow & (\text{cdr}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}))\\
& \quad < \quad \text{cdr}\,(\text{v\&c-apply\$}\,(\texttt{'if},\\
& \qquad\qquad\qquad \text{cons}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}),\\
& \qquad\qquad\qquad\quad \text{v\&c\$}\,(\texttt{'list},\ \mathit{branches},\ \mathit{alist})))))
\end{array}$$

THEOREM: cdr-v&c$-lessp-if-caddr
$$\begin{array}{ll}
& (\text{v\&c-apply\$}\,(\texttt{'if},\ \text{cons}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}),\ \text{v\&c\$}\,(\texttt{'list},\ \mathit{branches},\ \mathit{alist})))\\
\wedge & \text{car}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist})))\\
\rightarrow & (\text{cdr}\,(\text{v\&c\$}\,(\mathbf{t},\ \text{car}\,(\mathit{branches}),\ \mathit{alist}))\\
& \quad < \quad \text{cdr}\,(\text{v\&c-apply\$}\,(\texttt{'if},\\
& \qquad\qquad\qquad \text{cons}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}),\\
& \qquad\qquad\qquad\quad \text{v\&c\$}\,(\texttt{'list},\ \mathit{branches},\ \mathit{alist})))))
\end{array}$$

THEOREM: cdr-v&c$-lessp-if-cadddr
$$\begin{array}{ll}
& (\text{v\&c-apply\$}\,(\texttt{'if},\ \text{cons}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}),\ \text{v\&c\$}\,(\texttt{'list},\ \mathit{branches},\ \mathit{alist})))\\
\wedge & (\neg\ \text{car}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}))))\\
\rightarrow & (\text{cdr}\,(\text{v\&c\$}\,(\mathbf{t},\ \text{cadr}\,(\mathit{branches}),\ \mathit{alist}))\\
& \quad < \quad \text{cdr}\,(\text{v\&c-apply\$}\,(\texttt{'if},\\
& \qquad\qquad\qquad \text{cons}\,(\text{v\&c\$}\,(\mathbf{t},\ \mathit{test},\ \mathit{alist}),\\
& \qquad\qquad\qquad\quad \text{v\&c\$}\,(\texttt{'list},\ \mathit{branches},\ \mathit{alist})))))
\end{array}$$

THEOREM: v&c$-f-l-eval-f-if-helper-1
$$\begin{array}{ll}
& ((\mathit{clock} \not\simeq \texttt{0})\\
\wedge & \text{listp}\,(\mathit{expr})\\
\wedge & (\text{car}\,(\mathit{expr}) = \texttt{'if})\\
\wedge & \text{l-eval}\,(\mathbf{t},\ \text{cadr}\,(\mathit{expr}),\ \mathit{alist},\ \mathit{clock})\\
\wedge & \text{v\&c\$}\,(\mathbf{t},\ \text{cadr}\,(\mathit{expr}),\ \mathit{alist})\\
\wedge & (\neg\ \text{v\&c-apply\$}\,(\texttt{'if},\\
& \qquad\qquad\qquad \text{cons}\,(\text{v\&c\$}\,(\mathbf{t},\ \text{cadr}\,(\mathit{expr}),\ \mathit{alist}),
\end{array}$$

v&c$ ('list, cddr (*expr*), *alist*))))
∧ car (l-eval (**t**, cadr (*expr*), *alist*, *clock*))
∧ v&c$ (**t**, caddr (*expr*), *alist*))
→ (¬ l-eval (**t**, caddr (*expr*), *alist*, *clock*))

EVENT: Disable v&c$-f-l-eval-f-if-helper-1.

THEOREM: v&c$-f-l-eval-f-if-helper-2
((*clock* ≠ 0)
∧ listp (*expr*)
∧ (car (*expr*) = 'if)
∧ l-eval (**t**, cadr (*expr*), *alist*, *clock*)
∧ v&c$ (**t**, cadr (*expr*), *alist*)
∧ (¬ v&c-apply$ ('if,
cons (v&c$ (**t**, cadr (*expr*), *alist*),
v&c$ ('list, cddr (*expr*), *alist*))))
∧ (¬ car (l-eval (**t**, cadr (*expr*), *alist*, *clock*)))
∧ v&c$ (**t**, cadddr (*expr*), *alist*))
→ (¬ l-eval (**t**, cadddr (*expr*), *alist*, *clock*))

EVENT: Disable v&c$-f-l-eval-f-if-helper-2.

THEOREM: v&c$-f-l-eval-flag-list
(**f** ∉ l-eval ('list, *expr*, *alist*, *clock*))
→ (strip-cars (v&c$ ('list, *expr*, *alist*))
= strip-cars (l-eval ('list, *expr*, *alist*, *clock*)))

EVENT: Disable v&c$-f-l-eval-flag-list.

THEOREM: v&c$-f-l-eval-f
(((*flag* = 'list) ∧ (**f** ∈ v&c$ (*flag*, *expr*, *alist*)))
∨ ((*flag* ≠ 'list) ∧ (v&c$ (*flag*, *expr*, *alist*) = **f**)))
→ (((*flag* = 'list) ∧ (**f** ∈ l-eval (*flag*, *expr*, *alist*, *clock*)))
∨ ((*flag* ≠ 'list) ∧ (l-eval (*flag*, *expr*, *alist*, *clock*) = **f**)))

THEOREM: l-eval-v&c$-flag-not-list
((¬ v&c$ (*flag*, *expr*, *alist*)) ∧ (*flag* ≠ 'list))
→ (l-eval (*flag*, *expr*, *alist*, *clock*) = **f**)

EVENT: Disable l-eval-v&c$-flag-not-list.

THEOREM: l-eval-v&c$-flag-list
(**f** ∈ v&c$ ('list, *expr*, *alist*))
→ (**f** ∈ l-eval ('list, *expr*, *alist*, *clock*))

THEOREM: v&c$-l-eval-flag-list
(**f** $\notin$ l-eval ('list, *expr*, *alist*, *clock*))
$\rightarrow$ (**f** $\notin$ v&c$ ('list, *expr*, *alist*))

THEOREM: member-f-v&c$-fact-1
(listp (*expr*)
$\wedge$ (car (*expr*) $\neq$ 'quote)
$\wedge$ (car (*expr*) $\neq$ 'if)
$\wedge$ (**f** $\in$ v&c$ ('list, cdr (*expr*), *alist*))
$\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ ($\neg$ v&c$ (*flag*, *expr*, *alist*))

THEOREM: sum-cdrs-v&c$-list-fact-1
(v&c-apply$ (*fun*, v&c$ ('list, *arg-list*, *alist*))
$\wedge$ (*fun* $\neq$ 'quote)
$\wedge$ (*fun* $\neq$ 'if))
$\rightarrow$ (sum-cdrs (v&c$ ('list, *arg-list*, *alist*))
$<$ cdr (v&c-apply$ (*fun*, v&c$ ('list, *arg-list*, *alist*))))

THEOREM: subrp-expr-v&c-apply$
(subrp (*x*) $\wedge$ (*x* $\neq$ 'if) $\wedge$ (**f** $\notin$ *args*))
$\rightarrow$ (car (v&c-apply$ (*x*, *args*)) = apply-subr (*x*, strip-cars (*args*)))

THEOREM: not-subrp-expr-v&c-apply$
($\neg$ subrp (*x*)) $\wedge$ (**f** $\notin$ *args*))
$\rightarrow$ (car (v&c-apply$ (*x*, *args*))
$=$ car (v&c$ (**t**, body (*x*), pairlist (formals (*x*), strip-cars (*args*)))))

THEOREM: v&c$-body-lessp-fact-1
(($\neg$ subrp (*fun*)) $\wedge$ v&c-apply$ (*fun*, *args*))
$\rightarrow$ (cdr (v&c$ (**t**, body (*fun*), pairlist (formals (*fun*), strip-cars (*args*))))
$<$ cdr (v&c-apply$ (*fun*, *args*)))

THEOREM: v&c$-not-subrp-expand-1
(v&c-apply$ (*fun*, *args*) $\wedge$ ($\neg$ subrp (*fun*)))
$\rightarrow$ v&c$ (**t**, body (*fun*), pairlist (formals (*fun*), strip-cars (*args*)))

THEOREM: not-member-remove-costs
($\neg$ listp (*x*)) $\rightarrow$ (*x* $\notin$ remove-costs (*list*))

THEOREM: l-eval-not-f-v&c$-equivalence
(((*flag* = 'list)
$\wedge$ (**f** $\notin$ v&c$ (*flag*, *expr*, *alist*))
$\wedge$ (sum-cdrs (v&c$ (*flag*, *expr*, *alist*)) $<$ *clock*))
$\vee$ ((*flag* $\neq$ 'list)

$\wedge$ (v&c\$ (*flag*, *expr*, *alist*) $\neq$ **f**)
$\wedge$ (cdr (v&c\$ (*flag*, *expr*, *alist*)) $<$ *clock*)))
$\rightarrow$ ((l-eval (*flag*, *expr*, *alist*, *clock*)
= **if** *flag* = `'list` **then** remove-costs (v&c\$ (*flag*, *expr*, *alist*))
else list (car (v&c\$ (*flag*, *expr*, *alist*))) **endif**)
$\wedge$ (((*flag* = `'list`) $\wedge$ (**f** $\notin$ v&c\$ (*flag*, *expr*, *alist*)))
$\vee$ ((*flag* $\neq$ `'list`) $\wedge$ (v&c\$ (*flag*, *expr*, *alist*) $\neq$ **f**))))

```
; ------------------------------------------------------------
; was s-eval1.events
; ------------------------------------------------------------
```

Event: Disable pack-equal.

Definition:
number-cons (*x*)
= **if** listp (*x*) **then** 1 + (number-cons (car (*x*)) + number-cons (cdr (*x*)))
else `0` **endif**

Theorem: number-cons-car
listp (*x*) $\rightarrow$ (number-cons (car (*x*)) $<$ number-cons (*x*))

Theorem: number-cons-cdr
listp (*x*) $\rightarrow$ (number-cons (cdr (*x*)) $<$ number-cons (*x*))

Theorem: number-cons-cadr-caddr-cadddr
listp (*x*)
$\rightarrow$ ((number-cons (cadr (*x*)) $<$ number-cons (*x*))
$\wedge$ (number-cons (caddr (*x*)) $<$ number-cons (*x*))
$\wedge$ (number-cons (cadddr (*x*)) $<$ number-cons (*x*)))

```
;; ***** The S-level (S for Subexpression).  *****
;; This level is somewhat similar to the logic.  Quoted constants are now
;; looked up in a global alist.
;; An S-STATE has eight components.
;; S-PNAME:  the name of the program currently being executed
;; S-POS:    the "position" of the expression currently being evaluated.
;; S-ANS:    the answer
;; S-PARAMS: an alist with entries of form (<parameter name> . <value>)
;; S-TEMPS:  an alist with entries of form (<expr> <flag> <value>)
;; S-PROGS: an alist with entires of form
;;          (<program name> <formals> <temps> <body>)
;; S-ERR-FLAG: A flag that should be 'RUN, otherwise it means we terminated
;;             with an error.
```

```
;; S-PNAME names the current program, S-POS denotes the position.
;; The position is a list of integers that uniquely determines the
;; current sub-expression.
;; An example in the list:
;;   (a b (c d (f g) (h (i j) k l m)))
;; the position (2 2) denotes the list (f g).
;;
;; The expression and the bodies of the programs (S-PROGS) are
;; slightly different from the logic.
;; Parameters (entries in S-PARAMS) appear as LITATOMs (i.e. they
;; are not changed).
;; Entries in S-TEMPS are triples, the CAR is the expression,
;; the CADR is a boolean flag, if T it means that the CADDR is the value
;; of the expression, otherwise the CADDR is undefined.
;; There are 3 forms that effect or are effected by S-TEMPS.  The CAR of
;; each of these forms is a singleton list, with one of the values:
;; TEMP-EVAL, TEMP-FETCH, or TEMP-TEST.
;; All these forms should have an expression as the only argument
;; (called EXP below).

;; TEMP-EVAL:  if the flag corresoponding to EXP in S-TEMPS is not F,
;;             set S-ERR-FLAG to an error and return, otherwise evaluate
;;             EXP and put it and the value into S-TEMPS and set the flag.
;; TEMP-FETCH: if the flag corresponding to EXP in S-TEMPS is set returns
;;             the value associated with EXP, otherwise sets S-ERR-FLAG
;;             to an error.
;; TEMP-TEST:  if the flag corresponding to EXP in S-TEMPS is set
;;             returns the value associated with EXP, otherwise evaluate
;;             EXP and put it and the value into S-TEMPS, setting the flag.
```

DEFINITION: S-TEMP-EVAL = `'(temp-eval)`

DEFINITION: S-TEMP-FETCH = `'(temp-fetch)`

DEFINITION: S-TEMP-TEST = `'(temp-test)`

DEFINITION: s-temp-setp (*expr*, *temps*) = cadr (assoc (*expr*, *temps*))

EVENT: Disable s-temp-setp.

DEFINITION: s-temp-value (*expr*, *temps*) = caddr (assoc (*expr*, *temps*))

EVENT: Add the shell *s-state*, with recognizer function symbol *s-statep* and 7 accessors: *s-pname*, with type restriction (none-of) and default value zero;

s-pos, with type restriction (none-of) and default value zero; *s-ans*, with type restriction (none-of) and default value zero; *s-params*, with type restriction (none-of) and default value zero; *s-temps*, with type restriction (none-of) and default value zero; *s-progs*, with type restriction (none-of) and default value zero; *s-err-flag*, with type restriction (none-of) and default value zero.

DEFINITION:
cur-expr (*position*, *expr*)
= **if** *position* $\simeq$ **nil then** *expr*
else cur-expr (cdr (*position*), get (car (*position*), *expr*)) **endif**

DEFINITION: butlast (*list*) = firstn (length (*list*) − 1, *list*)

EVENT: Disable butlast.

DEFINITION:
last (*l*)
= **if** listp (*l*)
then if listp (cdr (*l*)) **then** last (cdr (*l*))
else *l* **endif**
else nil endif

DEFINITION: dv (*pos*, *increment*) = append (*pos*, list (*increment*))

EVENT: Disable dv.

DEFINITION:
nx (*pos*) = append (butlast (*pos*), list (1 + car (last (*pos*))))

EVENT: Disable nx.

```
;; We prepend all user names with U-.  This is to avoid name conflicts.
;; We used to just make sure the main program was not one of the names
;; specified to LOGIC->S (by using GENSYM).  However we needed to know that
;; it was not a SUBRP.  This made the correctness theorem almost useless,
;; because it had an hypothesis of the form:
;;   (not (subrp (car (gensym (unpack 'main) nil pnames))))
;; This could not in general be reduced.  Also we will probably need to have
;; our own functions.  The names of these functions should start with I-.
;; U- is for User, I- for Internal.
```

DEFINITION:
USER-FNAME-PREFIX = list (car (unpack (`'u-`)), cadr (unpack (`'u-`)))

DEFINITION:
user-fname (*name*) = pack (append (USER-FNAME-PREFIX, unpack (*name*)))

EVENT: Disable user-fname.

DEFINITION:
user-fnamep (*name*)
= (litatom (*name*)
∧ (car (unpack (*name*)) = car (USER-FNAME-PREFIX))
∧ (cadr (unpack (*name*)) = cadr (USER-FNAME-PREFIX)))

EVENT: Disable user-fnamep.

DEFINITION: logic-fname (*name*) = pack (cddr (unpack (*name*)))

EVENT: Disable logic-fname.

DEFINITION: s-formals (*s-program*) = cadr (*s-program*)

EVENT: Disable s-formals.

DEFINITION: s-temp-list (*s-program*) = caddr (*s-program*)

EVENT: Disable s-temp-list.

DEFINITION: s-body (*s-program*) = cadddr (*s-program*)

EVENT: Disable s-body.

DEFINITION: s-prog (*s*) = definition (s-pname (*s*), s-progs (*s*))

EVENT: Disable s-prog.

DEFINITION: s-expr (*s*) = cur-expr (s-pos (*s*), s-body (s-prog (*s*)))

EVENT: Disable s-expr.

DEFINITION:
s-expr-list (*s*)
= restn (car (last (s-pos (*s*))),
cur-expr (butlast (s-pos (*s*)), s-body (s-prog (*s*)))))

EVENT: Disable s-expr-list.

THEOREM: get-anything-nil
$(\neg\ \text{listp}\,(\mathit{lst})) \rightarrow (\text{get}\,(\mathit{anything},\ \mathit{lst}) = \mathtt{0})$

EVENT: Disable get-anything-nil.

THEOREM: get-cons
$(k \not\simeq \mathtt{0}) \rightarrow (\text{get}\,(k,\ \text{cons}\,(\mathit{anything},\ \mathit{list})) = \text{get}\,(k - 1,\ \mathit{list}))$

EVENT: Disable get-cons.

THEOREM: get-large-index
$(n \not< \text{length}\,(\mathit{list})) \rightarrow (\text{get}\,(n,\ \mathit{list}) = \mathtt{0})$

EVENT: Disable get-large-index.

THEOREM: get-zerop
$(n \simeq \mathtt{0}) \rightarrow (\text{get}\,(n,\ \mathit{list}) = \text{car}\,(\mathit{list}))$

DEFINITION:
s-set-pos $(s,\ \mathit{pos})$
= s-state (s-pname (s),
pos,
s-ans (s),
s-params (s),
s-temps (s),
s-progs (s),
s-err-flag (s))

THEOREM: s-accessors-s-set-pos
(s-pname (s-set-pos $(s,\ \mathit{pos})$) = s-pname (s))
$\wedge$ (s-pos (s-set-pos $(s,\ \mathit{pos})$) = pos)
$\wedge$ (s-ans (s-set-pos $(s,\ \mathit{pos})$) = s-ans (s))
$\wedge$ (s-params (s-set-pos $(s,\ \mathit{pos})$) = s-params (s))
$\wedge$ (s-temps (s-set-pos $(s,\ \mathit{pos})$) = s-temps (s))
$\wedge$ (s-progs (s-set-pos $(s,\ \mathit{pos})$) = s-progs (s))
$\wedge$ (s-err-flag (s-set-pos $(s,\ \mathit{pos})$) = s-err-flag (s))

EVENT: Disable s-set-pos.

DEFINITION:
s-set-temps $(\mathit{state},\ \mathit{new\text{-}temps})$

= s-state (s-pname (*state*),
s-pos (*state*),
s-ans (*state*),
s-params (*state*),
new-temps,
s-progs (*state*),
s-err-flag (*state*))

THEOREM: s-accessors-s-set-temps
(s-pname (s-set-temps (*state*, *temps*)) = s-pname (*state*))
∧ (s-pos (s-set-temps (*state*, *temps*)) = s-pos (*state*))
∧ (s-ans (s-set-temps (*state*, *temps*)) = s-ans (*state*))
∧ (s-temps (s-set-temps (*state*, *temps*)) = *temps*)
∧ (s-params (s-set-temps (*state*, *temps*)) = s-params (*state*))
∧ (s-progs (s-set-temps (*state*, *temps*)) = s-progs (*state*))
∧ (s-err-flag (s-set-temps (*state*, *temps*)) = s-err-flag (*state*))

EVENT: Disable s-set-temps.

DEFINITION:
s-set-error (*state*, *flag*)
= s-state (s-pname (*state*),
s-pos (*state*),
s-ans (*state*),
s-params (*state*),
s-temps (*state*),
s-progs (*state*),
flag)

THEOREM: s-accessors-s-set-error
(s-pname (s-set-error (*state*, *flag*)) = s-pname (*state*))
∧ (s-pos (s-set-error (*state*, *flag*)) = s-pos (*state*))
∧ (s-ans (s-set-error (*state*, *flag*)) = s-ans (*state*))
∧ (s-temps (s-set-error (*state*, *flag*)) = s-temps (*state*))
∧ (s-params (s-set-error (*state*, *flag*)) = s-params (*state*))
∧ (s-progs (s-set-error (*state*, *flag*)) = s-progs (*state*))
∧ (s-err-flag (s-set-error (*state*, *flag*)) = *flag*)

EVENT: Disable s-set-error.

DEFINITION:
s-change-temp (*s*, *expr*, *value*)
= s-set-temps (*s*, put-assoc (list (**t**, *value*), *expr*, s-temps (*s*)))

THEOREM: s-accessors-s-change-temp
(s-pname (s-change-temp (*s*, *e*, *v*)) = s-pname (*s*))
∧ (s-pos (s-change-temp (*s*, *e*, *v*)) = s-pos (*s*))
∧ (s-ans (s-change-temp (*s*, *e*, *v*)) = s-ans (*s*))
∧ (s-params (s-change-temp (*s*, *e*, *v*)) = s-params (*s*))
∧ (s-progs (s-change-temp (*s*, *e*, *v*)) = s-progs (*s*))
∧ (s-err-flag (s-change-temp (*s*, *e*, *v*)) = s-err-flag (*s*))

EVENT: Disable s-change-temp.

DEFINITION:
s-set-ans (*state*, *ans*)
= s-state (s-pname (*state*),
s-pos (*state*),
ans,
s-params (*state*),
s-temps (*state*),
s-progs (*state*),
s-err-flag (*state*))

THEOREM: s-accessors-s-set-ans
(s-pname (s-set-ans (*s*, *ans*)) = s-pname (*s*))
∧ (s-pos (s-set-ans (*s*, *ans*)) = s-pos (*s*))
∧ (s-ans (s-set-ans (*s*, *ans*)) = *ans*)
∧ (s-params (s-set-ans (*s*, *ans*)) = s-params (*s*))
∧ (s-temps (s-set-ans (*s*, *ans*)) = s-temps (*s*))
∧ (s-progs (s-set-ans (*s*, *ans*)) = s-progs (*s*))
∧ (s-err-flag (s-set-ans (*s*, *ans*)) = s-err-flag (*s*))

EVENT: Disable s-set-ans.

DEFINITION:
s-eval-do-temp-fetch (*state*)
= **if** s-temp-setp (cadr (s-expr (*state*)), s-temps (*state*))
then s-set-ans (*state*, s-temp-value (cadr (s-expr (*state*)), s-temps (*state*)))
else s-set-error (*state*, `'temp-fetch-not-set`) **endif**

THEOREM: s-accessors-s-eval-do-temp-fetch
(s-pname (s-eval-do-temp-fetch (*state*)) = s-pname (*state*))
∧ (s-pos (s-eval-do-temp-fetch (*state*)) = s-pos (*state*))
∧ (s-ans (s-eval-do-temp-fetch (*state*))
= **if** s-temp-setp (cadr (s-expr (*state*)), s-temps (*state*))
then s-temp-value (cadr (s-expr (*state*)), s-temps (*state*))
else s-ans (*state*) **endif**)

∧ (s-params (s-eval-do-temp-fetch (*state*)) = s-params (*state*))
∧ (s-temps (s-eval-do-temp-fetch (*state*)) = s-temps (*state*))
∧ (s-progs (s-eval-do-temp-fetch (*state*)) = s-progs (*state*))
∧ (s-err-flag (s-eval-do-temp-fetch (*state*))
= **if** s-temp-setp (cadr (s-expr (*state*)), s-temps (*state*))
then s-err-flag (*state*)
else `'temp-fetch-not-set` **endif**)

EVENT: Disable s-eval-do-temp-fetch.

DEFINITION:
make-temps-entries (*list*)
= **if** listp (*list*)
then cons (list (car (*list*), **f**, **nil**), make-temps-entries (cdr (*list*)))
else nil endif

DEFINITION:
s-fun-call-state (*s*, *pname*)
= s-state (user-fname (*pname*),
nil,
s-ans (*s*),
pairlist (s-formals (assoc (user-fname (*pname*), s-progs (*s*))),
s-ans (*s*)),
make-temps-entries (s-temp-list (assoc (user-fname (*pname*),
s-progs (*s*)))),
s-progs (*s*),
`'run`)

THEOREM: s-accessors-s-fun-call-state
(s-pname (s-fun-call-state (*s*, *pname*)) = user-fname (*pname*))
∧ (s-pos (s-fun-call-state (*s*, *pname*)) = **nil**)
∧ (s-ans (s-fun-call-state (*s*, *pname*)) = s-ans (*s*))
∧ (s-params (s-fun-call-state (*s*, *pname*))
= pairlist (s-formals (assoc (user-fname (*pname*), s-progs (*s*))),
s-ans (*s*)))
∧ (s-temps (s-fun-call-state (*s*, *pname*))
= make-temps-entries (s-temp-list (assoc (user-fname (*pname*),
s-progs (*s*)))))
∧ (s-progs (s-fun-call-state (*s*, *pname*)) = s-progs (*s*))
∧ (s-err-flag (s-fun-call-state (*s*, *pname*)) = `'run`)

EVENT: Disable s-fun-call-state.

DEFINITION:

s-set-expr (*s1*, *s2*, *pos*)
= s-state (s-pname (*s2*),
pos,
s-ans (*s1*),
s-params (*s1*),
s-temps (*s1*),
s-progs (*s2*),
s-err-flag (*s1*))

THEOREM: s-accessors-s-set-expr
(s-pname (s-set-expr (*s1*, *s2*, *pos*)) = s-pname (*s2*))
∧ (s-pos (s-set-expr (*s1*, *s2*, *pos*)) = *pos*)
∧ (s-ans (s-set-expr (*s1*, *s2*, *pos*)) = s-ans (*s1*))
∧ (s-params (s-set-expr (*s1*, *s2*, *pos*)) = s-params (*s1*))
∧ (s-temps (s-set-expr (*s1*, *s2*, *pos*)) = s-temps (*s1*))
∧ (s-progs (s-set-expr (*s1*, *s2*, *pos*)) = s-progs (*s2*))
∧ (s-err-flag (s-set-expr (*s1*, *s2*, *pos*)) = s-err-flag (*s1*))

EVENT: Disable s-set-expr.

THEOREM: lessp-number-cons-restn-get
(*n* < length (*x*)) → (number-cons (get (*n*, *x*)) < number-cons (restn (*n*, *x*)))

THEOREM: get-add1-opener
get (1 + *x*, *list*) = get (*x*, cdr (*list*))

THEOREM: restn-add1-opener
listp (*list*) → (restn (1 + *x*, *list*) = restn (*x*, cdr (*list*)))

THEOREM: restn-cdr
(*n* < length (*x*)) → (restn (*n*, cdr (*x*)) = cdr (restn (*n*, *x*)))

EVENT: Disable restn-cdr.

THEOREM: car-restn-get
car (restn (*n*, *list*)) = get (*n*, *list*)

THEOREM: s-prog-s-set-expr
s-prog (s-set-expr (*s1*, *s2*, *pos*)) = s-prog (*s2*)

THEOREM: s-prog-s-set-pos
s-prog (s-set-pos (*s*, *pos*)) = s-prog (*s*)

THEOREM: cur-expr-append
cur-expr (append (*pos1*, *pos2*), *body*) = cur-expr (*pos2*, cur-expr (*pos1*, *body*))

THEOREM: s-expr-s-set-expr
s-expr (s-set-expr (*s1*, *s2*, dv (s-pos (*s2*), *n*))) = get (*n*, s-expr (*s2*))

THEOREM: s-expr-s-set-pos-t
s-expr (s-set-pos (*s*, dv (s-pos (*s*), *n*))) = get (*n*, s-expr (*s*))

THEOREM: butlast-append
listp (*l2*) → (butlast (append (*l1*, *l2*)) = append (*l1*, butlast (*l2*)))

THEOREM: restn-sub1-length-last
((*n* = (length (*l*) − 1)) ∧ listp (*l*)) → (restn (*n*, *l*) = last (*l*))

THEOREM: append-butlast-lastcdr
listp (*x*) → (append (butlast (*x*), last (*x*)) = *x*)

THEOREM: not-listp-cdr-last
listp (cdr (last (*x*))) = **f**

THEOREM: listp-last-listp
listp (last (*x*)) = listp (*x*)

THEOREM: cur-expr-last
(listp (*pos*) ∧ (car (last (*pos*)) < length (*body*)))
→ (cur-expr (last (*pos*), *body*) = get (car (last (*pos*)), *body*))

THEOREM: number-cons-cur-expr
((car (last (*pos*)) < length (cur-expr (butlast (*pos*), *body*))) ∧ listp (*pos*))
→ (number-cons (cur-expr (*pos*, *body*))
< number-cons (restn (car (last (*pos*)), cur-expr (butlast (*pos*), *body*)))))

THEOREM: last-append
listp (*y*) → (last (append (*x*, *y*)) = last (*y*))

THEOREM: butlast-singleton-list
butlast (list (*x*)) = **nil**

THEOREM: last-nx
listp (*pos*) → (last (nx (*pos*)) = list (1 + car (last (*pos*))))

THEOREM: butlast-nx
listp (*pos*) → (butlast (nx (*pos*)) = butlast (*pos*))

THEOREM: cur-expr-plist
cur-expr (plist (*pos*), *body*) = cur-expr (*pos*, *body*)

THEOREM: last-dv
last (dv (*pos*, *n*)) = list (*n*)

THEOREM: butlast-dv
butlast (dv (*pos*, *n*)) = plist (*pos*)

THEOREM: s-expr-list-s-s-set-expr-nx
(listp (s-pos (*s*)) ∧ listp (s-expr-list (*s*)))
→ (s-expr-list (s-set-expr (*s1*, *s*, nx (s-pos (*s*)))) = cdr (s-expr-list (*s*)))

THEOREM: lessp-number-cons-s-expr-s-expr-list
(listp (s-pos (*s*)) ∧ listp (s-expr-list (*s*)))
→ (number-cons (s-expr (*s*)) < number-cons (s-expr-list (*s*)))

THEOREM: s-expr-list-s-set-pos-dv
s-expr-list (s-set-pos (*s*, dv (s-pos (*s*), *n*))) = restn (*n*, s-expr (*s*))

```
;; if flag is 'list then state contains a list of expressions,
;; otherwise it is just one.
;; Returns a s-state.  If flag is LIST the S-ANS of the s-state is
;; a list results.  Otherwise S-ANS of the result is the answer.
```

DEFINITION:
s-eval (*flag*, *s*, *c*)
= **if** s-err-flag (*s*) ≠ 'run **then** *s*
elseif *flag* = 'list
then if s-pos (*s*) ≃ **nil** **then** s-set-error (*s*, 'bad-list-position)
elseif listp (s-expr-list (*s*))
then let *car-s* **be** s-eval (**t**, *s*, *c*)
in
let *cdr-s* **be** s-eval ('list,
s-set-expr (*car-s*,
s,
nx (s-pos (*s*))),
c)
in
s-set-ans (*cdr-s*,
cons (s-ans (*car-s*), s-ans (*cdr-s*))) **endlet** **endlet**
else s-set-ans (*s*, **nil**) **endif**
elseif *c* ≃ 0 **then** s-set-error (*s*, 'out-of-time)
elseif litatom (s-expr (*s*))
then s-set-ans (*s*, value (s-expr (*s*), s-params (*s*)))
elseif s-expr (*s*) ≃ **nil** **then** s-set-error (*s*, 'bad-expression)
elseif car (s-expr (*s*)) = 'if
then let *test* **be** s-eval (**t**, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)
in
if s-err-flag (*test*) = 'run
then if s-ans (*test*)

```
            then s-eval (t, s-set-expr (test, s, dv (s-pos (s), 2)), c)
            else s-eval (t,
                         s-set-expr (test, s, dv (s-pos (s), 3)),
                         c) endif
       else test endif endlet
  elseif car (s-expr (s)) = S-TEMP-EVAL
  then let new-s be s-eval (t, s-set-pos (s, dv (s-pos (s), 1)), c)
       in
       s-change-temp (new-s, cadr (s-expr (s)), s-ans (new-s)) endlet
  elseif car (s-expr (s)) = S-TEMP-TEST
  then if s-temp-setp (cadr (s-expr (s)), s-temps (s))
       then s-eval-do-temp-fetch (s)
       else let new-s be s-eval (t,
                                 s-set-pos (s, dv (s-pos (s), 1)),
                                 c)
            in
            s-change-temp (new-s,
                           cadr (s-expr (s)),
                           s-ans (new-s)) endlet endif
  elseif car (s-expr (s)) = S-TEMP-FETCH then s-eval-do-temp-fetch (s)
  elseif car (s-expr (s)) = 'quote then s-set-ans (s, cadr (s-expr (s)))
  elseif s-err-flag (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c))
       ≠ 'run
  then s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)
  elseif subrp (car (s-expr (s)))
  then let arg-s be s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)
       in
       s-set-ans (arg-s, apply-subr (car (s-expr (s)), s-ans (arg-s))) endlet
  elseif litatom (car (s-expr (s)))
  then let arg-s be s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)
       in
       let new-s be s-eval (t,
                            s-fun-call-state (arg-s,
                                              car (s-expr (s))),
                            c − 1)
       in
       if s-err-flag (new-s) = 'run
       then s-set-expr (s-set-ans (arg-s, s-ans (new-s)),
                        s,
                        s-pos (s))
       else new-s endif endlet endlet
  else s-set-error (s, 'bad-instruction) endif
```

THEOREM: s-progs-s-eval

s-progs (s-eval (*flag*, *state*, *clock*)) = s-progs (*state*)

THEOREM: s-params-s-eval
(s-err-flag (s-eval (*flag*, *state*, *clock*)) = `'run`)
→ (s-params (s-eval (*flag*, *state*, *clock*)) = s-params (*state*))

THEOREM: s-pname-s-eval
(s-err-flag (s-eval (*flag*, *state*, *clock*)) = `'run`)
→ (s-pname (s-eval (*flag*, *state*, *clock*)) = s-pname (*state*))

```
;; Takes body and expands away temporary variables.
;; BODY is a S-STATE expression.
;; Returns the expanded body
```

DEFINITION:
s-expand-temps (*flag*, *body*)
= **if** *flag* = `'list`
then if listp (*body*)
then cons (s-expand-temps (**t**, car (*body*)),
s-expand-temps (`'list`, cdr (*body*)))
else nil endif
elseif *body* ≃ **nil then** *body*
elseif (car (*body*) = S-TEMP-EVAL)
∨ (car (*body*) = S-TEMP-FETCH)
∨ (car (*body*) = S-TEMP-TEST)
then s-expand-temps (**t**, cadr (*body*))
elseif car (*body*) = `'quote` **then** *body*
else cons (car (*body*), s-expand-temps (`'list`, cdr (*body*))) **endif**

DEFINITION:
SUBR-ARITY-ALIST
=
```
'((add1 1)
  (add-to-set 2)
  (and 2)
  (append 2)
  (apply-subr 2)
  (apply$ 2)
  (assoc 2)
  (body 1)
  (car 1)
  (cdr 1)
  (cons 2)
  (count 1)
  (difference 2)
  (equal 2)
```

```
(eval$ 3)
(false 0)
(falsep 1)
(fix 1)
(fix-cost 2)
(for 6)
(formals 1)
(geq 2)
(greaterp 2)
(if 3)
(iff 2)
(implies 2)
(leq 2)
(lessp 2)
(listp 1)
(litatom 1)
(max 2)
(member 2)
(minus 1)
(negativep 1)
(negative-guts 1)
(nlistp 1)
(not 1)
(numberp 1)
(or 2)
(ordinalp 1)
(ord-lessp 2)
(pack 1)
(pairlist 2)
(plus 2)
(quantifier-initial-value 1)
(quantifier-operation 3)
(quotient 2)
(remainder 2)
(strip-cars 1)
(sub1 1)
(subrp 1)
(sum-cdrs 1)
(times 2)
(true 0)
(truep 1)
(union 2)
(unpack 1)
(v&c$ 3)
```

```
(v&c-apply$ 2)
(zero 0)
(zerop 1))
```

DEFINITION:
arity (*function*)
= **if** subrp (*function*) **then** cadr (assoc (*function*, SUBR-ARITY-ALIST))
elseif *function* = `'quote` **then** `1`
else length (formals (*function*)) **endif**

EVENT: Disable arity.

DEFINITION:
s-proper-exprp (*flag*, *expr*, *p-names*, *formals*, *temp-list*)
= **if** *flag* = `'list`
then if listp (*expr*)
then s-proper-exprp (**t**, car (*expr*), *p-names*, *formals*, *temp-list*)
∧ s-proper-exprp (`'list`,
cdr (*expr*),
p-names,
formals,
temp-list)
else *expr* = **nil endif**
elseif listp (*expr*)
then if ¬ plistp (*expr*) **then f**
elseif (car (*expr*) = S-TEMP-FETCH)
∨ (car (*expr*) = S-TEMP-EVAL)
∨ (car (*expr*) = S-TEMP-TEST)
then (cadr (*expr*) ∈ *temp-list*)
∧ (length (*expr*) = `2`)
∧ s-proper-exprp (**t**,
cadr (*expr*),
p-names,
formals,
temp-list)
elseif car (*expr*) = `'quote`
then length (cdr (*expr*)) = arity (car (*expr*))
elseif subrp (car (*expr*))
then (length (cdr (*expr*)) = arity (car (*expr*)))
∧ (car (*expr*) ∉ *p-names*)
∧ s-proper-exprp (`'list`,
cdr (*expr*),
p-names,
formals,

temp-list)
elseif body (car (*expr*))
then (length (cdr (*expr*)) = arity (car (*expr*)))
∧ (car (*expr*) ∈ *p-names*)
∧ s-proper-exprp (`'list`,
temp-list)
else f endif
elseif litatom (*expr*) **then** *expr* ∈ *formals*
else f endif

DEFINITION:
s-programs-properp (*programs*, *program-names*)
= **if** listp (*programs*)
then all-litatoms (s-formals (car (*programs*)))
∧ s-proper-exprp (**t**,
s-body (car (*programs*)),
program-names,
s-formals (car (*programs*)),
s-temp-list (car (*programs*)))
∧ s-programs-properp (cdr (*programs*), *program-names*)
else t endif

DEFINITION:
s-programs-okp (*programs*)
= **if** listp (*programs*)
then (s-formals (car (*programs*)) ≠ **f**)
∧ user-fnamep (car (car (*programs*)))
∧ (s-formals (car (*programs*))
= formals (logic-fname (car (car (*programs*)))))
∧ (s-expand-temps (**t**, s-body (car (*programs*)))
= body (logic-fname (car (car (*programs*)))))
∧ s-programs-okp (cdr (*programs*))
else t endif

DEFINITION:
temps-okp (*temps*, *params*, *c*)
= **if** listp (*temps*)
then ((¬ cadr (car (*temps*)))
∨ (l-eval (**t**, s-expand-temps (**t**, car (car (*temps*))), *params*, *c*)
∧ (car (l-eval (**t**,
s-expand-temps (**t**, car (car (*temps*))),
params,

c))
$=$ caddr (car (*temps*)))))
$\wedge$ temps-okp (cdr (*temps*), *params*, c)
else t endif

DEFINITION:
good-posp1 (*pos*, *expr*)
$=$ **if** *pos* $\simeq$ **nil then t**
elseif *expr* $\simeq$ **nil then f**
elseif car (*expr*) = S-TEMP-FETCH **then f**
elseif (car (*expr*) = S-TEMP-EVAL) $\vee$ (car (*expr*) = S-TEMP-TEST)
then (car (*pos*) = `1`) $\wedge$ good-posp1 (cdr (*pos*), cadr (*expr*))
elseif car (*expr*) = `'quote` **then f**
else (car (*pos*) $\not\simeq$ `0`)
$\wedge$ (car (*pos*) $<$ length (*expr*))
$\wedge$ good-posp1 (cdr (*pos*), get (car (*pos*), *expr*)) **endif**

EVENT: Disable good-posp1.

DEFINITION:
good-posp-list (n, *expr*)
$=$ **if** *expr* $\simeq$ **nil then f**
elseif (car (*expr*) = S-TEMP-FETCH)
$\vee$ (car (*expr*) = S-TEMP-EVAL)
$\vee$ (car (*expr*) = S-TEMP-TEST)
$\vee$ (car (*expr*) = `'quote`) **then f**
else ($n \not\simeq$ `0`) $\wedge$ (length (*expr*) $\not<$ n) **endif**

DEFINITION:
good-posp (*flag*, *pos*, *expr*)
$=$ **if** *flag* = `'list`
then good-posp-list (car (last (*pos*)), cur-expr (butlast (*pos*), *expr*))
$\wedge$ good-posp1 (butlast (*pos*), *expr*)
else good-posp1 (*pos*, *expr*) **endif**

EVENT: Disable good-posp.

DEFINITION:
all-user-fnamesp (*list*)
$=$ **if** listp (*list*)
then user-fnamep (car (*list*)) $\wedge$ all-user-fnamesp (cdr (*list*))
else t endif

DEFINITION:

strip-logic-fnames (*list*)
= **if** listp (*list*)
then cons (logic-fname (caar (*list*)), strip-logic-fnames (cdr (*list*)))
else nil endif

```
;; This is a bit bogus.  We check S-PROGRAMS-OKP of the CDR of S-PROGS.  This
;; is because S-PROGRAMS-OKP checks that BODY and FORMALS are defined on
;; the names of the PROGRAMS.  Therefore it is NECESSARY that the expression
;; given to the compiler be the first program in S-PROGS.
;; Note that all functions calls must be defined, but can not be the
;; first program.
```

DEFINITION:
s-good-statep (*s*, *c*)
= (definedp (s-pname (*s*), s-progs (*s*))
∧ (car (car (s-progs (*s*))) = 'main)
∧ s-programs-properp (s-progs (*s*),
strip-logic-fnames (cdr (s-progs (*s*))))
∧ s-programs-okp (cdr (s-progs (*s*)))
∧ (strip-cars (s-temps (*s*)) = plist (s-temp-list (s-prog (*s*))))
∧ (s-err-flag (*s*) = 'run)
∧ temps-okp (s-temps (*s*), s-params (*s*), *c*))

THEOREM: s-good-statep-backchainer-1
s-good-statep (*s*, *c*) → definedp (s-pname (*s*), s-progs (*s*))

THEOREM: not-user-fnamep-not-definedp-s-programs-okp
((¬ user-fnamep (*name*)) ∧ definedp (*name*, *progs*))
→ (¬ s-programs-okp (*progs*))

THEOREM: s-good-statep-backchainer-2
s-good-statep (*s*, *c*) → (¬ definedp ('main, cdr (s-progs (*s*))))

EVENT: Disable s-good-statep-backchainer-2.

THEOREM: s-good-statep-backchainer-2-5
s-good-statep (*s*, *c*) → (caar (s-progs (*s*)) = 'main)

EVENT: Disable s-good-statep-backchainer-2-5.

THEOREM: s-good-statep-backchainer-3
s-good-statep (*s*, *c*)
→ s-programs-properp (s-progs (*s*), strip-logic-fnames (cdr (s-progs (*s*))))

THEOREM: s-good-statep-backchainer-4
s-good-statep (*s*, *c*) → s-programs-okp (cdr (s-progs (*s*)))

THEOREM: s-good-statep-backchainer-5
s-good-statep (*s*, *c*) → temps-okp (s-temps (*s*), s-params (*s*), *c*)

THEOREM: s-good-statep-backchainer-6
s-good-statep (*s*, *c*) → (s-err-flag (*s*) = 'run)

THEOREM: s-good-statep-strip-cars-temps
s-good-statep (*s*, *c*)
→ (strip-cars (s-temps (*s*)) = plist (s-temp-list (s-prog (*s*))))

THEOREM: s-good-statep-s-set-pos
s-good-statep (s-set-pos (*s*, *pos*), *c*) = s-good-statep (*s*, *c*)

THEOREM: temps-okp-put-assoc
(temps-okp (*temps*, *params*, *c*) ∧ l-eval (**t**, s-expand-temps (**t**, *expr*), *params*, *c*))
→ temps-okp (put-assoc (list (**t**,
car (l-eval (**t**, s-expand-temps (**t**, *expr*), *params*, *c*))),
expr,
temps),
params,
c)

THEOREM: s-prog-s-set-temps
s-prog (s-set-temps (*s*, *new-temps*)) = s-prog (*s*)

THEOREM: s-good-statep-s-change-temp
let *expr* **be** cadr (s-expr (*s*)),
s1 **be** s-eval (**t**, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)
in
(s-good-statep (*s1*, *c*)
∧ l-eval (**t**, s-expand-temps (**t**, *expr*), s-params (*s*), *c*)
∧ (*x* = car (l-eval (**t**, s-expand-temps (**t**, *expr*), s-params (*s*), *c*))))
→ s-good-statep (s-change-temp (*s1*, *expr*, *x*), *c*) **endlet**

THEOREM: s-prog-s-set-ans
s-prog (s-set-ans (*s*, *ans*)) = s-prog (*s*)

THEOREM: s-good-statep-s-set-ans
s-good-statep (s-set-ans (*s*, *ans*), *c*) = s-good-statep (*s*, *c*)

THEOREM: s-good-statep-s-set-expr
(s-good-statep (*s1*, *c*) ∧ s-good-statep (*s2*, *c*) ∧ (s-prog (*s1*) = s-prog (*s2*)))
→ s-good-statep (s-set-expr (*s1*, *s2*, *pos*), *c*)

EVENT: Disable s-good-statep-strip-cars-temps.

THEOREM: s-programs-properp-s-proper-exprp
(s-programs-properp (*progs*, *program-names*) ∧ (*prog* ∈ *progs*))
→ s-proper-exprp (**t**,
s-body (*prog*),
program-names,
s-formals (*prog*),
s-temp-list (*prog*))

THEOREM: s-good-statep-s-proper-exprp
s-good-statep (*s*, *c*)
→ s-proper-exprp (**t**,
s-body (s-prog (*s*)),
strip-logic-fnames (cdr (s-progs (*s*))),
s-formals (s-prog (*s*)),
s-temp-list (s-prog (*s*)))

THEOREM: s-good-statep-s-temp-list
s-good-statep (*s*, *c*)
→ ((*x* ∈ s-temp-list (s-prog (*s*))) = definedp (*x*, s-temps (*s*)))

EVENT: Disable s-good-statep.

THEOREM: s-proper-exprp-list-s-proper-get-t
((*n* < length (*expr*))
∧ plistp (*expr*)
∧ s-proper-exprp (`'list`, *expr*, *p-names*, *formals*, *temp-list*))
→ s-proper-exprp (**t**, get (*n*, *expr*), *p-names*, *formals*, *temp-list*)

THEOREM: s-proper-exprp-t-s-proper-get-t
((car (*expr*) ≠ S-TEMP-FETCH)
∧ (car (*expr*) ≠ S-TEMP-EVAL)
∧ (car (*expr*) ≠ S-TEMP-TEST)
∧ (car (*expr*) ≠ `'quote`)
∧ s-proper-exprp (**t**, *expr*, *p-names*, *formals*, *temp-list*)
∧ (*n* ≯ `0`)
∧ (*n* < length (*expr*)))
→ s-proper-exprp (**t**, get (*n*, *expr*), *p-names*, *formals*, *temp-list*)

EVENT: Disable s-proper-exprp-list-s-proper-get-t.

THEOREM: s-proper-exprp-s-proper-exprp-cur-expr
(s-proper-exprp (**t**, *body*, *p-names*, *formals*, *temp-list*) ∧ good-posp1 (*pos*, *body*))
→ s-proper-exprp (**t**, cur-expr (*pos*, *body*), *p-names*, *formals*, *temp-list*)

THEOREM: s-good-statep-s-proper-exprp-cur-expr
(s-good-statep (s, c) ∧ good-posp1 (s-pos (s), s-body (s-prog (s))))
→ s-proper-exprp (**t**,
s-expr (s),
strip-logic-fnames (cdr (s-progs (s))),
s-formals (s-prog (s)),
s-temp-list (s-prog (s)))

THEOREM: s-proper-exprp-definedp
(s-proper-exprp (**t**, *body*, *p-names*, *formals*, *temp-list*)
∧ ((car (*body*) = S-TEMP-EVAL)
∨ (car (*body*) = S-TEMP-FETCH)
∨ (car (*body*) = S-TEMP-TEST)))
→ (cadr (*body*) ∈ *temp-list*)

THEOREM: s-good-statep-definedp-temps
(s-good-statep (s, c)
∧ good-posp1 (s-pos (s), s-body (s-prog (s)))
∧ ((car (s-expr (s)) = S-TEMP-EVAL)
∨ (car (s-expr (s)) = S-TEMP-FETCH)
∨ (car (s-expr (s)) = S-TEMP-TEST)))
→ definedp (cadr (s-expr (s)), s-temps (s))

THEOREM: strip-cars-s-temps-s-eval
(s-err-flag (s-eval (*flag*, *state*, *clock*)) = 'run)
→ (strip-cars (s-temps (s-eval (*flag*, *state*, *clock*)))
= strip-cars (s-temps (*state*)))

THEOREM: temps-okp-make-temps-entries
temps-okp (make-temps-entries (x), *params*, c)

THEOREM: l-eval-flag-not-list
(*flag* ≠ 'list)
→ (l-eval (*flag*, *expr*, *params*, *clock*) = l-eval (**t**, *expr*, *params*, *clock*))

THEOREM: strip-cars-make-temps-entries
strip-cars (make-temps-entries (x)) = plist (x)

THEOREM: s-err-flag-s-eval-flag-list-flag-t
((s-err-flag (s-eval ('list, s-set-expr (s-eval (**t**, s, c), s, nx (s-pos (s))), c))
= 'run)
∧ listp (s-expr-list (s)))
→ (s-err-flag (s-eval (**t**, s, c)) = 'run)

THEOREM: good-posp1-append
good-posp1 (append (*pos1*, *pos2*), *body*)
= (good-posp1 (*pos1*, *body*) ∧ good-posp1 (*pos2*, cur-expr (*pos1*, *body*)))

THEOREM: cur-expr-nlistp
(*expr* $\simeq$ **nil**) $\rightarrow$ (listp (cur-expr (*pos*, *expr*)) = **f**)

THEOREM: good-posp1-nlistp
($\neg$ listp (*pos*)) $\rightarrow$ good-posp1 (*pos*, *body*)

THEOREM: good-posp1-list-good-posp-list-t
(good-posp1 (butlast (*pos*), *body*)
 $\wedge$ good-posp-list (car (last (*pos*)), cur-expr (butlast (*pos*), *body*))
 $\wedge$ listp (restn (car (last (*pos*)), cur-expr (butlast (*pos*), *body*)))
 $\wedge$ listp (*pos*))
$\rightarrow$ good-posp1 (*pos*, *body*)

THEOREM: good-posp-flag-not-list-good-posp1
(*flag* $\neq$ 'list) $\rightarrow$ (good-posp (*flag*, *pos*, *body*) = good-posp1 (*pos*, *body*))

THEOREM: good-posp-list-t
(good-posp ('list, s-pos (*s*), s-body (s-prog (*s*)))
 $\wedge$ listp (s-expr-list (*s*))
 $\wedge$ listp (s-pos (*s*)))
$\rightarrow$ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))

THEOREM: s-prog-s-eval-do-s-temp-fetch
s-prog (s-eval-do-temp-fetch (*state*)) = s-prog (*state*)

THEOREM: s-prog-s-eval
(s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run)
$\rightarrow$ (s-prog (s-eval (*flag*, *s*, *c*)) = s-prog (*s*))

THEOREM: good-posp-list-nx
(good-posp ('list, s-pos (*s*), s-body (s-prog (*s*)))
 $\wedge$ listp (s-pos (*s*))
 $\wedge$ listp (s-expr-list (*s*)))
$\rightarrow$ good-posp ('list, nx (s-pos (*s*)), s-body (s-prog (*s*)))

THEOREM: car-s-expr-list-s-expr
(listp (s-expr-list (*s*)) $\wedge$ listp (s-pos (*s*)))
$\rightarrow$ (car (s-expr-list (*s*)) = s-expr (*s*))

THEOREM: l-eval-s-expand-temps-flag-list-fact-1
l-eval ('list, s-expand-temps ('list, *body*), *params*, *clock*)
= **if** listp (*body*)
  then cons (l-eval (**t**, s-expand-temps (**t**, car (*body*)), *params*, *clock*),
          l-eval ('list,
                s-expand-temps ('list, cdr (*body*)),
                params,
                clock))
  else nil endif

```
;; The rewrite rule l-eval-s-expand-temps-flag-list-fact-1 had body
;; in place of (restn (car (last (s-pos s))) (s-expr s)), but this cause
;; infinite rewriting, so I changed it to block rewriting after doing
;; it once.
```

THEOREM: l-eval-s-expand-temps-flag-list-s-expr-fact-1
l-eval ('list, s-expand-temps ('list, s-expr-list (s)), *params*, *clock*)
= **if** listp (s-expr-list (s))
then cons (l-eval (**t**, s-expand-temps (**t**, car (s-expr-list (s))), *params*, *clock*),
l-eval ('list,
s-expand-temps ('list, cdr (s-expr-list (s))),
params,
clock))
else nil endif

THEOREM: l-eval-s-expand-temps-litatom-fact-1
((*clock* $\not\simeq$ 0) $\wedge$ litatom (*body*) $\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ (l-eval (*flag*, s-expand-temps (*flag*, *body*), *params*, *clock*)
= list (cdr (assoc (*body*, *params*))))

THEOREM: s-proper-exprp-length-cur-expr
(s-proper-exprp (**t**, *expr*, *p-names*, *formals*, *temp-list*)
$\wedge$ listp (*expr*)
$\wedge$ (subrp (car (*expr*)) $\vee$ body (car (*expr*)))
$\wedge$ (car (*expr*) $\neq$ 'quote))
$\rightarrow$ (length (*expr*) = (1 + arity (car (*expr*)))))

THEOREM: good-posp1-cons-lessp-4-if
((car (s-expr (s)) = 'if) $\wedge$ s-good-statep (s, c))
$\rightarrow$ ((good-posp1 (dv (s-pos (s), 1), s-body (s-prog (s)))
= good-posp1 (s-pos (s), s-body (s-prog (s))))
$\wedge$ (good-posp1 (dv (s-pos (s), 2), s-body (s-prog (s)))
= good-posp1 (s-pos (s), s-body (s-prog (s))))
$\wedge$ (good-posp1 (dv (s-pos (s), 3), s-body (s-prog (s)))
= good-posp1 (s-pos (s), s-body (s-prog (s)))))

THEOREM: l-eval-if-fact-1
((car (*expr*) = 'if) $\wedge$ (*flag* $\neq$ 'list))
$\rightarrow$ (l-eval (*flag*, s-expand-temps (*flag*, *expr*), *params*, *clock*)
= **if** l-eval (**t**, s-expand-temps (**t**, cadr (*expr*)), *params*, *clock*)
then if car (l-eval (**t**,
s-expand-temps (**t**, cadr (*expr*)),
params,
clock))
then l-eval (**t**, s-expand-temps (**t**, caddr (*expr*)), *params*, *clock*)

else l-eval (**t**,
s-expand-temps (**t**, cadddr (*expr*)),
params,
clock) **endif**
else f endif)

THEOREM: s-good-statep-s-eval-do-temp-fetch
(s-good-statep (*s*, *c*) ∧ s-temp-setp (cadr (s-expr (*s*)), s-temps (*s*)))
→ s-good-statep (s-eval-do-temp-fetch (*s*), *c*)

THEOREM: good-posp1-dv-1-temps
(((car (s-expr (*s*)) = S-TEMP-EVAL) ∨ (car (s-expr (*s*)) = S-TEMP-TEST))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*))))
→ good-posp1 (dv (s-pos (*s*), 1), s-body (s-prog (*s*)))

THEOREM: s-eval-l-eval-s-temps
((*flag* ≠ 'list)
∧ ((car (*expr*) = S-TEMP-EVAL)
∨ (car (*expr*) = S-TEMP-FETCH)
∨ (car (*expr*) = S-TEMP-TEST)))
→ (l-eval (*flag*, s-expand-temps (*flag*, *expr*), *params*, *clock*)
= l-eval (**t**, s-expand-temps (**t**, cadr (*expr*)), *params*, *clock*))

THEOREM: l-eval-quote-fact-1
((car (*expr*) = 'quote) ∧ (*flag* ≠ 'list) ∧ (*clock* ≠ 0))
→ (l-eval (*flag*, s-expand-temps (*flag*, *expr*), *params*, *clock*)
= list (cadr (*expr*)))

THEOREM: listp-not-lessp-length-1
listp (*x*) → ((length (*x*) < 1) = **f**)

EVENT: Disable listp-not-lessp-length-1.

THEOREM: good-posp1-plist
good-posp1 (plist (*pos*), *body*) = good-posp1 (*pos*, *body*)

THEOREM: good-posp-dv-1-funcall
(listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ 'if)
∧ (car (s-expr (*s*)) ≠ S-TEMP-EVAL)
∧ (car (s-expr (*s*)) ≠ S-TEMP-TEST)
∧ (car (s-expr (*s*)) ≠ S-TEMP-FETCH)
∧ (car (s-expr (*s*)) ≠ 'quote)
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*))))
→ good-posp ('list, dv (s-pos (*s*), 1), s-body (s-prog (*s*)))

THEOREM: l-eval-s-expand-temps-subrp-fact-1
(subrp (car (*e*))
 ∧ (car (*e*) ≠ 'if)
 ∧ (*clock* ≠ 0)
 ∧ (**f** ∉ l-eval ('list, s-expand-temps ('list, cdr (*e*)), *params*, *clock*))
 ∧ (*flag* ≠ 'list))
→ (l-eval (*flag*, s-expand-temps (*flag*, *e*), *params*, *clock*)
 = list (apply-subr (car (*e*),
 strip-cars (l-eval ('list,
 s-expand-temps ('list,
 cdr (*e*)),
 params,
 clock)))))

THEOREM: logic-fname-user-fname-identity
litatom (*x*) → (logic-fname (user-fname (*x*)) = *x*)

THEOREM: user-fname-logic-fname-identity
user-fnamep (*x*) → (user-fname (logic-fname (*x*)) = *x*)

THEOREM: member-strip-logic-fnames-definedp
(all-user-fnamesp (strip-cars (*y*)) ∧ litatom (*x*))
→ ((*x* ∈ strip-logic-fnames (*y*)) = definedp (user-fname (*x*), *y*))

THEOREM: s-programs-okp-all-user-fnamesp-strip-cars
s-programs-okp (*programs*) → all-user-fnamesp (strip-cars (*programs*))

THEOREM: s-proper-exprp-definedp-programs
(s-proper-exprp (**t**, *body*, strip-logic-fnames (*programs*), *formals*, *temp-list*)
 ∧ listp (*body*)
 ∧ s-programs-okp (*programs*)
 ∧ (¬ subrp (car (*body*)))
 ∧ (car (*body*) ≠ 'quote)
 ∧ litatom (car (*body*)))
→ definedp (user-fname (car (*body*)), *programs*)

THEOREM: s-good-statep-s-fun-call-state
(s-good-statep (*s1*, *c*)
 ∧ s-good-statep (*s2*, *c*)
 ∧ listp (s-expr (*s1*))
 ∧ (car (s-expr (*s1*)) ≠ 'if)
 ∧ (car (s-expr (*s1*)) ≠ 'quote)
 ∧ (¬ subrp (car (s-expr (*s1*))))
 ∧ litatom (car (s-expr (*s1*)))
 ∧ good-posp1 (s-pos (*s1*), s-body (s-prog (*s1*)))
 ∧ (s-progs (*s1*) = s-progs (*s2*)))
→ s-good-statep (s-fun-call-state (*s2*, car (s-expr (*s1*))), *c* − 1)

THEOREM: good-posp1-flag-not-list-nil
good-posp1 (**nil**, *body*)

THEOREM: l-eval-s-expand-temps-not-subrp-fact-1
((¬ subrp (car (*e*)))
 ∧ listp (*e*)
 ∧ (¬ listp (car (*e*)))
 ∧ (*clock* ≠ 0)
 ∧ (car (*e*) ≠ 'quote)
 ∧ (**f** ∉ l-eval ('list, s-expand-temps ('list, cdr (*e*)), *params*, *clock*))
 ∧ (*flag* ≠ 'list))
→ (l-eval (*flag*, s-expand-temps (*flag*, *e*), *params*, *clock*)
 = l-eval (**t**,
 body (car (*e*)),
 pairlist (formals (car (*e*)),
 strip-cars (l-eval ('list,
 s-expand-temps ('list,
 cdr (*e*)),
 params,
 clock))),
 clock − 1))

THEOREM: s-programs-okp-formals-body
(s-programs-okp (*progs*) ∧ (*prog* ∈ *progs*))
→ ((s-formals (*prog*) = formals (logic-fname (car (*prog*))))
 ∧ (s-expand-temps (**t**, s-body (*prog*))
 = body (logic-fname (car (*prog*)))))

THEOREM: s-good-statep-not-car-s-expr-caar-s-progs
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
 ∧ listp (s-expr (*s*))
 ∧ (car (s-expr (*s*)) ≠ 'quote)
 ∧ (¬ subrp (car (s-expr (*s*))))
 ∧ litatom (car (s-expr (*s*)))
 ∧ listp (s-progs (*s*))
 ∧ (user-fname (car (s-expr (*s*))) = caar (s-progs (*s*))))
→ (¬ s-good-statep (*s*, *c*))

EVENT: Disable s-good-statep-not-car-s-expr-caar-s-progs.

THEOREM: s-expand-temps-s-expr-s-fun-call-state
(s-good-statep (*s1*, *c*)
 ∧ s-good-statep (*s2*, *c*)
 ∧ listp (s-expr (*s1*)))

$\wedge$ good-posp1 (s-pos (*s1*), s-body (s-prog (*s1*)))
$\wedge$ (car (s-expr (*s1*)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (s-expr (*s1*))))
$\wedge$ litatom (car (s-expr (*s1*)))
$\wedge$ (s-progs (*s1*) = s-progs (*s2*)))
$\rightarrow$ (s-expand-temps (**t**, s-expr (s-fun-call-state (*s2*, car (s-expr (*s1*))))))
= body (car (s-expr (*s1*))))

THEOREM: s-good-statep-formals
(s-good-statep (*s*, *c*)
$\wedge$ listp (s-expr (*s*))
$\wedge$ (car (s-expr (*s*)) $\neq$ 'if)
$\wedge$ (car (s-expr (*s*)) $\neq$ 'quote)
$\wedge$ ($\neg$ subrp (car (s-expr (*s*))))
$\wedge$ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ litatom (car (s-expr (*s*))))
$\rightarrow$ (s-formals (assoc (user-fname (car (s-expr (*s*))), s-progs (*s*)))
= formals (car (s-expr (*s*))))

THEOREM: definedp-temps-okp
(temps-okp (*temps*, *params*, *c*)
$\wedge$ definedp (*expr*, *temps*)
$\wedge$ s-temp-setp (*expr*, *temps*))
$\rightarrow$ (l-eval (**t**, s-expand-temps (**t**, *expr*), *params*, *c*)
$\wedge$ (car (l-eval (**t**, s-expand-temps (**t**, *expr*), *params*, *c*))
= s-temp-value (*expr*, *temps*)))

THEOREM: l-eval-s-eval-do-temp-fetch-fact-1
(good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ listp (s-expr (*s*))
$\wedge$ ((car (s-expr (*s*)) = S-TEMP-FETCH)
$\vee$ (car (s-expr (*s*)) = S-TEMP-TEST))
$\wedge$ s-temp-setp (cadr (s-expr (*s*)), s-temps (*s*))
$\wedge$ s-good-statep (*s*, *c*))
$\rightarrow$ (l-eval (**t**, s-expand-temps (**t**, cadr (s-expr (*s*))), s-params (*s*), *c*)
$\wedge$ (car (l-eval (**t**, s-expand-temps (**t**, cadr (s-expr (*s*))), s-params (*s*), *c*))
= s-ans (s-eval-do-temp-fetch (*s*))))

THEOREM: s-eval-l-eval-equivalence
(s-good-statep (*s*, *c*)
$\wedge$ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
$\wedge$ (s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run))
$\rightarrow$ (s-good-statep (s-eval (*flag*, *s*, *c*), *c*)
$\wedge$ **if** *flag* = 'list
then (**f** $\notin$ l-eval (*flag*,

```
                    s-expand-temps (flag, s-expr-list (s)),
                    s-params (s),
                    c))
          ∧   (s-ans (s-eval (flag, s, c))
              =   strip-cars (l-eval (flag,
                                      s-expand-temps (flag,
                                                      s-expr-list (s)),
                                      s-params (s),
                                      c)))
     else l-eval (flag,
                  s-expand-temps (flag, s-expr (s)),
                  s-params (s),
                  c)
          ∧   (s-ans (s-eval (flag, s, c))
              =   car (l-eval (flag,
                               s-expand-temps (flag,
                                               s-expr (s)),
                               s-params (s),
                               c))) endif)
```

THEOREM: s-eval-s-good-statep
(s-good-statep (*s*, *c*)
∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
∧ (s-err-flag (s-eval (*flag*, *s*, *c*)) = `'run`))
→ s-good-statep (s-eval (*flag*, *s*, *c*), *c*)

THEOREM: s-eval-l-eval-flag-t
(s-good-statep (*s*, *c*)
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*)))
∧ (s-err-flag (s-eval (**t**, *s*, *c*)) = `'run`))
→ (l-eval (**t**, s-expand-temps (**t**, s-expr (*s*)), s-params (*s*), *c*)
∧ (s-ans (s-eval (**t**, *s*, *c*))
= car (l-eval (**t**,
s-expand-temps (**t**, s-expr (*s*)),
s-params (*s*),
c))))

DEFINITION:
s-collect-all-temps (*flag*, *expr*)
= **if** *flag* = `'list`
then if listp (*expr*)
then append (s-collect-all-temps (**t**, car (*expr*)),
s-collect-all-temps (`'list`, cdr (*expr*)))
else nil endif
elseif listp (*expr*)

```
    then if (car (expr) = S-TEMP-TEST) ∨ (car (expr) = S-TEMP-EVAL)
         then cons (cadr (expr), s-collect-all-temps (t, cadr (expr)))
         elseif car (expr) = S-TEMP-FETCH
         then s-collect-all-temps (t, cadr (expr))
         elseif car (expr) = 'quote then nil
         elseif car (expr) = 'if
         then append (s-collect-all-temps (t, cadr (expr)),
                      bagint (s-collect-all-temps (t, caddr (expr)),
                              s-collect-all-temps (t, cadddr (expr))))
         else s-collect-all-temps ('list, cdr (expr)) endif
    else nil endif

;; This takes an expression (appropriate for s-eval) and makes sure
;; that all the temp-fetch forms will have the setp flag set.
;; temp-set is a set of expressions that can be assumed to have been evaluated

DEFINITION:
s-all-temps-setp (flag, expr, temp-set)
=   if flag = 'list
    then if listp (expr)
         then s-all-temps-setp (t, car (expr), temp-set)
              ∧   s-all-temps-setp ('list,
                                    cdr (expr),
                                    append (s-collect-all-temps (t,
                                                                 car (expr)),
                                            temp-set))
         else t endif
    elseif listp (expr)
    then if (car (expr) = S-TEMP-EVAL) ∨ (car (expr) = S-TEMP-TEST)
         then s-all-temps-setp (t, cadr (expr), temp-set)
         elseif car (expr) = S-TEMP-FETCH then cadr (expr) ∈ temp-set
         elseif car (expr) = 'quote then t
         elseif car (expr) = 'if
         then if s-all-temps-setp (t, cadr (expr), temp-set)
              then let test-temp-set be s-collect-all-temps (t,
                                                             cadr (expr))
                   in
                   s-all-temps-setp (t,
                                     caddr (expr),
                                     append (test-temp-set, temp-set))
                   ∧   s-all-temps-setp (t,
                                         cadddr (expr),
                                         append (test-temp-set,
                                                 temp-set)) endlet
```

else f endif
else s-all-temps-setp (`'list`, cdr (*expr*), *temp-set*) **endif**
else t endif

DEFINITION:
temp-alist-to-set-1 (*alist*, *acc*)
= **if** listp (*alist*)
then if listp (car (*alist*)) ∧ cadar (*alist*) ∧ (caar (*alist*) ∉ *acc*)
then cons (caar (*alist*),
temp-alist-to-set-1 (cdr (*alist*), cons (caar (*alist*), *acc*)))
else temp-alist-to-set-1 (cdr (*alist*), cons (caar (*alist*), *acc*)) **endif**
else nil endif

THEOREM: temp-alist-to-set-1-gives-members
(*x* ∈ temp-alist-to-set-1 (*alist*, *acc*))
= (listp (assoc (*x*, *alist*)) ∧ s-temp-setp (*x*, *alist*) ∧ (*x* ∉ *acc*))

DEFINITION:
temp-alist-to-set (*alist*) = temp-alist-to-set-1 (*alist*, **nil**)

EVENT: Disable temp-alist-to-set.

THEOREM: temp-alist-to-set-gives-members
(*x* ∈ temp-alist-to-set (*alist*))
= (listp (assoc (*x*, *alist*)) ∧ s-temp-setp (*x*, *alist*))

DEFINITION:
s-check-temps-setp-1 (*exprs*, *temp-set*)
= **if** listp (*exprs*)
then ((car (*exprs*) ∉ *temp-set*)
∨ subsetp (s-collect-all-temps (**t**, car (*exprs*)), *temp-set*))
∧ s-check-temps-setp-1 (cdr (*exprs*), *temp-set*)
else t endif

DEFINITION:
good-alistp (*alist*)
= **if** listp (*alist*) **then** listp (car (*alist*)) ∧ good-alistp (cdr (*alist*))
else t endif

THEOREM: good-alistp-listp-assoc
good-alistp (*alist*) → (definedp (*expr*, *alist*) ↔ listp (assoc (*expr*, *alist*)))

EVENT: Disable good-alistp-listp-assoc.

THEOREM: good-alistp-put-assoc
good-alistp (*alist*) → good-alistp (put-assoc (*val*, *expr*, *alist*))

DEFINITION:
s-check-temps-setp (*temp-alist*)
= (good-alistp (*temp-alist*)
∧ s-check-temps-setp-1 (strip-cars (*temp-alist*),
temp-alist-to-set (*temp-alist*)))

EVENT: Disable s-check-temps-setp.

DEFINITION:
s-all-progs-temps-setp (*progs*)
= **if** listp (*progs*)
then s-all-temps-setp (**t**, s-body (car (*progs*)), **nil**)
∧ s-check-temps-setp (make-temps-entries (s-temp-list (car (*progs*))))
∧ s-all-progs-temps-setp (cdr (*progs*))
else t endif

EVENT: Disable s-all-progs-temps-setp.

THEOREM: subsetp-member
(subsetp (*set1*, *set2*) ∧ (*x* ∈ *set1*)) → (*x* ∈ *set2*)

EVENT: Disable subsetp-member.

THEOREM: subsetp-append-1
subsetp (append (*set1*, *set2*), *set3*)
= (subsetp (*set1*, *set3*) ∧ subsetp (*set2*, *set3*))

THEOREM: subsetp-cons
subsetp (*set1*, *set2*) → subsetp (*set1*, cons (*x*, *set2*))

THEOREM: subsetp-reflexive
subsetp (*set*, *set*)

THEOREM: subsetp-append-2
(subsetp (*set1*, *set2*) ∨ subsetp (*set1*, *set3*))
→ subsetp (*set1*, append (*set2*, *set3*))

THEOREM: temp-alist-to-set-1-subsetp
subsetp (*acc1*, *acc2*)
→ subsetp (temp-alist-to-set-1 (*temp-alist*, *acc2*),
temp-alist-to-set-1 (*temp-alist*, *acc1*))

THEOREM: temp-alist-to-set-1-nlistp
(¬ listp (temp-alist-to-set-1 (*alist*, *acc*)))
→ (temp-alist-to-set-1 (*alist*, *acc*) = **nil**)

THEOREM: temp-alist-to-set-1-cons-nil
(temp-alist-to-set-1 (*temp-alist*, *acc*) = **nil**)
→ (temp-alist-to-set-1 (*temp-alist*, cons (*x*, *acc*)) = **nil**)

THEOREM: temp-alist-to-set-1-make-temp-entries
temp-alist-to-set-1 (make-temps-entries (*temp-alist*), *acc*) = **nil**

THEOREM: temp-alist-to-set-make-temp-entries
temp-alist-to-set (make-temps-entries (*temp-alist*)) = **nil**

THEOREM: s-all-temps-setp-subsetp
(subsetp (*temp-set1*, *temp-set2*) ∧ s-all-temps-setp (*flag*, *expr*, *temp-set1*))
→ s-all-temps-setp (*flag*, *expr*, *temp-set2*)

THEOREM: subsetp-transistive
(subsetp (*set1*, *set2*) ∧ subsetp (*set2*, *set3*)) → subsetp (*set1*, *set3*)

EVENT: Disable subsetp-transistive.

THEOREM: s-temp-setp-put-assoc-1
s-temp-setp (*expr1*, put-assoc (cons (**t**, *anything*), *expr2*, *temp-alist*))
↔ **if** *expr1* = *expr2* **then t**
else s-temp-setp (*expr1*, *temp-alist*) **endif**

THEOREM: listp-assoc-fact-1
listp (assoc (*expr1*, *temp-alist*))
→ listp (assoc (*expr1*, put-assoc (cons (*x*, *y*), *expr2*, *temp-alist*)))

THEOREM: subsetp-temp-alist-to-set-put-assoc-1
definedp (*expr*, *temp-alist*)
→ (subsetp (*set1*,
temp-alist-to-set (put-assoc (cons (**t**, *anything*), *expr*, *temp-alist*)))
= subsetp (*set1*, cons (*expr*, temp-alist-to-set (*temp-alist*))))

THEOREM: subsetp-temp-alist-to-set-s-change-temp
definedp (*expr*, s-temps (*st*))
→ (subsetp (*set1*, temp-alist-to-set (s-temps (s-change-temp (*st*, *expr*, *val*))))
= subsetp (*set1*, cons (*expr*, temp-alist-to-set (s-temps (*st*)))))

THEOREM: not-definedp-put-assoc
(¬ definedp (*expr*, *temp-alist*))
→ (put-assoc (*anything*, *expr*, *temp-alist*) = *temp-alist*)

THEOREM: subsetp-trans-fact-2
subsetp (*set*, temp-alist-to-set (s-temps (*st*)))
→ subsetp (*set*, temp-alist-to-set (s-temps (s-change-temp (*st*, *expr*, *value*))))

THEOREM: s-eval-temps-subsetp
(s-err-flag (s-eval (*flag*, *s*, *c*)) = 'run)
→ subsetp (temp-alist-to-set (s-temps (*s*)),
temp-alist-to-set (s-temps (s-eval (*flag*, *s*, *c*))))

THEOREM: subsetp-delete
subsetp (*set1*, *set2*) → subsetp (delete (*x*, *set1*), *set2*)

THEOREM: subsetp-bagint
(subsetp (*set1*, *set3*) ∨ subsetp (*set2*, *set3*))
→ subsetp (bagint (*set1*, *set2*), *set3*)

THEOREM: l-eval-zerop-clock
((*clock* ≃ 0) ∧ (*flag* ≠ 'list))
→ (l-eval (*flag*, *expr*, *params*, *clock*) = **f**)

THEOREM: s-eval-temps-subsetp-s-set-pos
(s-err-flag (s-eval (**t**, s-set-pos (*s*, *pos*), *c*)) = 'run)
→ subsetp (temp-alist-to-set (s-temps (*s*)),
temp-alist-to-set (s-temps (s-eval (**t**, s-set-pos (*s*, *pos*), *c*))))

THEOREM: s-all-temps-setp-subsetp-if-caddr
let *test* **be** s-eval (**t**, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)
in
let *set1* **be** s-collect-all-temps (**t**, cadr (s-expr (*s*))),
set2 **be** temp-alist-to-set (s-temps (*test*)),
set3 **be** temp-alist-to-set (s-temps (*s*))
in
((s-err-flag (*test*) = 'run)
∧ (car (s-expr (*s*)) = 'if)
∧ s-all-temps-setp (**t**, caddr (s-expr (*s*)), append (*set1*, *set3*))
∧ subsetp (*set1*, *set2*))
→ s-all-temps-setp (**t**, caddr (s-expr (*s*)), *set2*) **endlet endlet**

THEOREM: s-all-temps-setp-subsetp-if-cadddr
let *test* **be** s-eval (**t**, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)
in
let *set1* **be** s-collect-all-temps (**t**, cadr (s-expr (*s*))),
set2 **be** temp-alist-to-set (s-temps (*test*)),
set3 **be** temp-alist-to-set (s-temps (*s*))
in
((s-err-flag (*test*) = 'run)
∧ (car (s-expr (*s*)) = 'if)
∧ s-all-temps-setp (**t**, cadddr (s-expr (*s*)), append (*set1*, *set3*))
∧ subsetp (s-collect-all-temps (**t**, cadr (s-expr (*s*))), *set2*))
→ s-all-temps-setp (**t**, cadddr (s-expr (*s*)), *set2*) **endlet endlet**

THEOREM: s-eval-temps-subsetp-s-set-expr
(s-err-flag (s-eval (**t**, s-set-expr (*s1*, *s2*, *e*), *c*)) = 'run)
→ subsetp (temp-alist-to-set (s-temps (*s1*)),
temp-alist-to-set (s-temps (s-eval (**t**, s-set-expr (*s1*, *s2*, *e*), *c*))))

THEOREM: s-eval-l-eval-flag-run-helper-1
let *test* **be** s-eval (**t**, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)
in
((s-err-flag (*s*) = 'run)
∧ (car (s-expr (*s*)) = 'if)
∧ (s-err-flag (*test*) = 'run)
∧ (s-err-flag (s-eval (**t**, s-set-expr (*test*, *s*, dv (s-pos (*s*), 2)), *c*))
= 'run)
∧ subsetp (s-collect-all-temps (**t**, caddr (s-expr (*s*))),
temp-alist-to-set (s-temps (s-eval (**t**,
s-set-expr (*test*,
s,
dv (s-pos (*s*),
2)),
c))))
∧ subsetp (s-collect-all-temps (**t**, cadr (s-expr (*s*))),
temp-alist-to-set (s-temps (*test*))))
→ subsetp (s-collect-all-temps (**t**, cadr (s-expr (*s*))),
temp-alist-to-set (s-temps (s-eval (**t**,
s-set-expr (*test*,
s,
dv (s-pos (*s*),
2)),
c)))) **endlet**

EVENT: Disable s-eval-l-eval-flag-run-helper-1.

THEOREM: s-eval-l-eval-flag-run-helper-2
let *test* **be** s-eval (**t**, s-set-pos (*s*, dv (s-pos (*s*), 1)), *c*)
in
((s-err-flag (*s*) = 'run)
∧ (car (s-expr (*s*)) = 'if)
∧ (s-err-flag (*test*) = 'run)
∧ (s-err-flag (s-eval (**t**, s-set-expr (*test*, *s*, dv (s-pos (*s*), 3)), *clock*))
= 'run)
∧ subsetp (s-collect-all-temps (**t**, cadddr (s-expr (*s*))),
temp-alist-to-set (s-temps (s-eval (**t**,
s-set-expr (*test*,
s,

dv (s-pos (*s*),
3)),
clock)))))
∧ subsetp (s-collect-all-temps (**t**, cadr (s-expr (*s*))),
temp-alist-to-set (s-temps (*test*))))
→ subsetp (s-collect-all-temps (**t**, cadr (s-expr (*s*))),
temp-alist-to-set (s-temps (s-eval (**t**,
s-set-expr (*test*,
s,
dv (s-pos (*s*),
3)),
clock)))) **endlet**

EVENT: Disable s-eval-l-eval-flag-run-helper-2.

THEOREM: s-temp-setp-s-change-temp
s-temp-setp (*expr1*, s-temps (s-change-temp (*state*, *expr2*, *value*)))
↔ **if** *expr1* = *expr2* **then t**
else s-temp-setp (*expr1*, s-temps (*state*)) **endif**

THEOREM: s-good-statep-listp-assoc-temps
(s-good-statep (*s*, *c*)
∧ s-check-temps-setp (s-temps (*s*))
∧ ((car (s-expr (*s*)) = S-TEMP-FETCH)
∨ (car (s-expr (*s*)) = S-TEMP-EVAL)
∨ (car (s-expr (*s*)) = S-TEMP-TEST))
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*))))
→ listp (assoc (cadr (s-expr (*s*)), s-temps (*s*)))

THEOREM: listp-assoc-s-change-temp
(s-good-statep (*s1*, *c*)
∧ ((car (s-expr (*s1*)) = S-TEMP-FETCH)
∨ (car (s-expr (*s1*)) = S-TEMP-EVAL)
∨ (car (s-expr (*s1*)) = S-TEMP-TEST))
∧ good-posp1 (s-pos (*s1*), s-body (s-prog (*s1*)))
∧ (strip-cars (s-temps (*s1*)) = strip-cars (s-temps (*s2*))))
→ listp (assoc (cadr (s-expr (*s1*)),
s-temps (s-change-temp (*s2*, cadr (s-expr (*s1*)), *value*))))

THEOREM: s-check-temps-setp-1-cons-non-member
(s-check-temps-setp-1 (*temp-list*, *temp-set*) ∧ (*expr* ∉ *temp-list*))
→ s-check-temps-setp-1 (*temp-list*, cons (*expr*, *temp-set*))

THEOREM: s-check-temps-setp-1-cons-member

```
(s-check-temps-setp-1 (temp-list, temp-set)
 ∧   (expr ∈ temp-list)
 ∧   subsetp (s-collect-all-temps (t, expr), temp-set))
 →   s-check-temps-setp-1 (temp-list, cons (expr, temp-set))
```

THEOREM: s-check-temps-setp-1-put-assoc

```
(definedp (expr, temp-alist)
 ∧   s-check-temps-setp-1 (temp-list,
                           cons (expr, temp-alist-to-set (temp-alist))))
 →   s-check-temps-setp-1 (temp-list,
                           temp-alist-to-set (put-assoc (cons (t, any),
                                                         expr,
                                                         temp-alist)))
```

THEOREM: s-check-temps-setp-put-assoc

```
let cadr-s be s-eval (t, s-set-pos (s, dv (s-pos (s), 1)), c)
in
(s-good-statep (s, c)
 ∧   good-posp1 (s-pos (s), s-body (s-prog (s)))
 ∧   s-check-temps-setp (s-temps (cadr-s))
 ∧   subsetp (s-collect-all-temps (t, cadr (s-expr (s))),
              temp-alist-to-set (s-temps (cadr-s)))
 ∧   ((car (s-expr (s)) = S-TEMP-TEST)
      ∨   (car (s-expr (s)) = S-TEMP-EVAL)))
 →   s-check-temps-setp (s-temps (s-change-temp (cadr-s,
                                                 cadr (s-expr (s)),
                                                 value))) endlet
```

THEOREM: member-check-s-temp-setp-1-subsetp

```
(s-check-temps-setp-1 (temp-list, set)
 ∧   (expr ∈ set)
 ∧   (expr ∈ temp-list))
 →   subsetp (s-collect-all-temps (t, expr), set)
```

THEOREM: s-eval-l-eval-flag-run-helper-3

```
(s-check-temps-setp (s-temps (s))
 ∧   s-temp-setp (cadr (s-expr (s)), s-temps (s))
 ∧   s-good-statep (s, c)
 ∧   good-posp1 (s-pos (s), s-body (s-prog (s)))
 ∧   ((car (s-expr (s)) = S-TEMP-TEST)
      ∨   (car (s-expr (s)) = S-TEMP-FETCH)))
 →   subsetp (s-collect-all-temps (t, cadr (s-expr (s))),
              temp-alist-to-set (s-temps (s)))
```

THEOREM: s-eval-l-eval-flag-run-helper-5

((**f** ∈ l-eval (`'list`,
s-expand-temps (`'list`, cdr (s-expr (*state*))),
s-params (*state*),
clock))
∧ listp (s-expr (*state*))
∧ (car (s-expr (*state*)) ≠ `'if`)
∧ (car (s-expr (*state*)) ≠ `'quote`)
∧ (car (s-expr (*state*)) ≠ S-TEMP-EVAL)
∧ (car (s-expr (*state*)) ≠ S-TEMP-FETCH)
∧ (car (s-expr (*state*)) ≠ S-TEMP-TEST)
∧ (*flag* ≠ `'list`))
→ (¬ l-eval (*flag*,
s-expand-temps (*flag*, s-expr (*state*)),
s-params (*state*),
clock))

THEOREM: s-check-temps-setp-member-progs
(s-all-progs-temps-setp (*progs*) ∧ (*x* ∈ *progs*))
→ s-check-temps-setp (make-temps-entries (s-temp-list (*x*)))

THEOREM: s-all-temps-setp-member-progs
(s-all-progs-temps-setp (*progs*) ∧ (*x* ∈ *progs*))
→ s-all-temps-setp (**t**, s-body (*x*), **nil**)

THEOREM: s-check-temps-setp-s-all-progs-temps-setp
(s-all-progs-temps-setp (s-progs (*s*))
∧ (¬ subrp (car (s-expr (*s*))))
∧ litatom (car (s-expr (*s*)))
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ `'quote`)
∧ s-good-statep (*s*, *c*)
∧ good-posp1 (s-pos (*s*), s-body (s-prog (*s*))))
→ s-check-temps-setp (make-temps-entries (s-temp-list (assoc (user-fname (car (s-expr (*s*))),
s-progs (*s*))))))

THEOREM: s-eval-l-eval-flag-run-helper-6
(l-eval (*flag*, s-expand-temps (*flag*, s-expr (*s*)), s-params (*s*), *c*)
∧ (*flag* ≠ `'list`)
∧ listp (s-expr (*s*))
∧ (car (s-expr (*s*)) ≠ S-TEMP-EVAL)
∧ (car (s-expr (*s*)) ≠ S-TEMP-TEST)
∧ (car (s-expr (*s*)) ≠ S-TEMP-FETCH)
∧ (¬ subrp (car (s-expr (*s*))))
∧ (car (s-expr (*s*)) ≠ `'if`)
∧ litatom (car (s-expr (*s*)))

$\wedge$ (car (s-expr (s)) $\neq$ 'quote)
$\wedge$ s-good-statep (s, c)
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ (s-err-flag (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c)) = 'run))
$\rightarrow$ l-eval (**t**,
body (car (s-expr (s))),
pairlist (formals (car (s-expr (s))),
s-ans (s-eval ('list, s-set-pos (s, dv (s-pos (s), 1)), c))),
$c - 1$)

THEOREM: s-all-temps-setp-s-all-progs-temps-setp
(s-all-progs-temps-setp (s-progs (*s1*))
$\wedge$ ($\neg$ subrp (car (s-expr (*s1*))))
$\wedge$ litatom (car (s-expr (*s1*)))
$\wedge$ (car (s-expr (*s1*)) $\neq$ 'quote)
$\wedge$ s-good-statep (*s1*, c)
$\wedge$ good-posp1 (s-pos (*s1*), s-body (s-prog (*s1*)))
$\wedge$ (s-progs (*s1*) = s-progs (*s2*)))
$\rightarrow$ s-all-temps-setp (**t**, s-expr (s-fun-call-state (*s2*, car (s-expr (*s1*)))), **nil**)

THEOREM: s-proper-exprp-fact-2
(good-posp1 (s-pos (s), s-body (s-prog (s)))
$\wedge$ ($\neg$ litatom (s-expr (s)))
$\wedge$ ($\neg$ listp (s-expr (s))))
$\rightarrow$ ($\neg$ s-good-statep (s, c))

THEOREM: s-all-temps-setp-subsetp-flag-list
((s-err-flag (s-eval (**t**, s, c)) = 'run)
$\wedge$ listp (s-expr-list (s))
$\wedge$ listp (s-pos (s))
$\wedge$ subsetp (s-collect-all-temps (**t**, s-expr (s)),
temp-alist-to-set (s-temps (s-eval (**t**, s, c))))
$\wedge$ s-all-temps-setp ('list,
cdr (s-expr-list (s)),
append (s-collect-all-temps (**t**, s-expr (s)),
temp-alist-to-set (s-temps (s)))))
$\rightarrow$ s-all-temps-setp ('list,
cdr (s-expr-list (s)),
temp-alist-to-set (s-temps (s-eval (**t**, s, c)))))

THEOREM: not-s-good-statep-bad-car-expr
(listp (s-expr (s))
$\wedge$ (car (s-expr (s)) $\neq$ 'if)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-EVAL)
$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-TEST)

$\wedge$ (car (s-expr (s)) $\neq$ S-TEMP-FETCH)
$\wedge$ (car (s-expr (s)) $\neq$ `'quote`)
$\wedge$ ($\neg$ litatom (car (s-expr (s))))
$\wedge$ good-posp1 (s-pos (s), s-body (s-prog (s))))
$\rightarrow$ ($\neg$ s-good-statep (s, c))

THEOREM: s-eval-l-eval-flag-run-helper-7
((s-err-flag (s-eval (`'list`, s-set-expr (s-eval (**t**, s, c), s, nx (s-pos (s))), c))
= `'run`)
$\wedge$ listp (s-expr-list (s))
$\wedge$ listp (s-pos (s))
$\wedge$ subsetp (s-collect-all-temps (**t**, s-expr (s)),
temp-alist-to-set (s-temps (s-eval (**t**, s, c))))
$\wedge$ subsetp (s-collect-all-temps (`'list`, cdr (s-expr-list (s))),
temp-alist-to-set (s-temps (s-eval (`'list`,
s-set-expr (s-eval (**t**, s, c),
s,
nx (s-pos (s))),
c)))))
$\rightarrow$ subsetp (s-collect-all-temps (**t**, s-expr (s)),
temp-alist-to-set (s-temps (s-eval (`'list`,
s-set-expr (s-eval (**t**, s, c),
s,
nx (s-pos (s))),
c))))

THEOREM: not-listp-pos-not-good-posp
($\neg$ listp (*pos*)) $\rightarrow$ ($\neg$ good-posp (`'list`, *pos*, *body*))

THEOREM: s-eval-l-eval-flag-run
(s-good-statep (s, c)
$\wedge$ good-posp (*flag*, s-pos (s), s-body (s-prog (s)))
$\wedge$ s-all-temps-setp (*flag*,
if *flag* = `'list` **then** s-expr-list (s)
else s-expr (s) **endif**,
temp-alist-to-set (s-temps (s)))
$\wedge$ s-all-progs-temps-setp (s-progs (s))
$\wedge$ **if** *flag* = `'list`
then f $\notin$ l-eval (*flag*,
s-expand-temps (*flag*, s-expr-list (s)),
s-params (s),
c)
else l-eval (*flag*, s-expand-temps (*flag*, s-expr (s)), s-params (s), c) **endif**
$\wedge$ s-check-temps-setp (s-temps (s)))
$\rightarrow$ ((s-err-flag (s-eval (*flag*, s, c)) = `'run`)

```
    ∧   s-check-temps-setp (s-temps (s-eval (flag, s, c)))
    ∧   subsetp (s-collect-all-temps (flag,
                                      if flag = 'list
                                      then s-expr-list (s)
                                      else s-expr (s) endif),
                 temp-alist-to-set (s-temps (s-eval (flag, s, c)))))

DEFINITION:
s-construct-programs (fun-list)
=   if listp (fun-list)
    then cons (list (user-fname (car (fun-list)),
                     formals (car (fun-list)),
                     nil,
                     body (car (fun-list))),
               s-construct-programs (cdr (fun-list)))
    else nil endif

DEFINITION:
delete-all (x, l)
=   if listp (l)
    then if x = car (l) then delete-all (x, cdr (l))
         else cons (car (l), delete-all (x, cdr (l))) endif
    else l endif

THEOREM: delete-all-non-decreasing-count
 count (l) ≮ count (delete-all (x, l))

DEFINITION:
remove-duplicates (l)
=   if listp (l)
    then cons (car (l), remove-duplicates (delete-all (car (l), cdr (l))))
    else l endif

DEFINITION:
logic->s (expr, alist, fun-names)
=   s-state ('main,
             nil,
             nil,
             alist,
             nil,
             cons (list ('main, strip-cars (alist), nil, expr),
                   s-construct-programs (remove-duplicates (fun-names))),
             'run)

; (setq st (logic->s '(plus (length (app '(a) x)) '2)
```

```
;         '((x . (b c)))
;          '(app length)))
```

DEFINITION:
l-proper-exprp (*flag*, *body*, *program-names*, *formals*)
= **if** *flag* = `'list`
then if listp (*body*)
then l-proper-exprp (**t**, car (*body*), *program-names*, *formals*)
∧ l-proper-exprp (`'list`,
cdr (*body*),
program-names,
formals)
else *body* = **nil endif**
elseif listp (*body*)
then if ¬ plistp (*body*) **then f**
elseif car (*body*) = `'quote`
then length (cdr (*body*)) = arity (car (*body*))
elseif subrp (car (*body*))
then (length (cdr (*body*)) = arity (car (*body*)))
∧ (car (*body*) ∉ *program-names*)
∧ l-proper-exprp (`'list`,
cdr (*body*),
program-names,
formals)
elseif body (car (*body*))
then (length (cdr (*body*)) = arity (car (*body*)))
∧ (car (*body*) ∈ *program-names*)
∧ l-proper-exprp (`'list`,
cdr (*body*),
program-names,
formals)
else f endif
elseif litatom (*body*) **then** *body* ∈ *formals*
else f endif

DEFINITION:
l-proper-programsp-1 (*program-names*, *all-program-names*)
= **if** listp (*program-names*)
then formals (car (*program-names*))
∧ all-litatoms (formals (car (*program-names*)))
∧ l-proper-exprp (**t**,
body (car (*program-names*)),
all-program-names,

formals (car (*program-names*)))
∧ l-proper-programsp-1 (cdr (*program-names*), *all-program-names*)
else t endif

DEFINITION:
l-proper-programsp (*program-names*)
= l-proper-programsp-1 (*program-names*, *program-names*)

EVENT: Disable l-proper-programsp.

THEOREM: definedp-assoc-fact-1
(¬ definedp (*expr*, *alist*)) → (¬ assoc (*expr*, *alist*))

EVENT: Disable definedp-assoc-fact-1.

DEFINITION:
all-litatoms-not-plist (*list*)
= **if** listp (*list*)
then litatom (car (*list*)) ∧ all-litatoms-not-plist (cdr (*list*))
else t endif

THEOREM: strip-logic-fnames-s-construct-programs
all-litatoms-not-plist (*program-names*)
→ (strip-logic-fnames (s-construct-programs (*program-names*))
= plist (*program-names*))

THEOREM: l-proper-expr-functions-definedp-s-proper-expr
l-proper-exprp (*flag*, *prog*, *all-program-names*, *formals*)
→ s-proper-exprp (*flag*, *prog*, *all-program-names*, *formals*, *temp-list*)

THEOREM: s-programs-properp-s-construct-programs
l-proper-programsp-1 (*program-names*, *all-program-names*)
→ s-programs-properp (s-construct-programs (*program-names*),
all-program-names)

THEOREM: s-proper-exprp-plist
s-proper-exprp (*flag*, *expr*, plist (*program-names*), *formals*, *temp-list*)
= s-proper-exprp (*flag*, *expr*, *program-names*, *formals*, *temp-list*)

THEOREM: s-proper-programsp-plist
s-programs-properp (*programs*, plist (*program-names*))
= s-programs-properp (*programs*, *program-names*)

THEOREM: s-expand-temps-body-equal-body
l-proper-exprp (*flag*, *expr*, *all-program-names*, *formals*)
→ (s-expand-temps (*flag*, *expr*) = *expr*)

THEOREM: l-proper-programsp-1-delete-all
l-proper-programsp-1 (*program-names*, *all-program-names*)
→ l-proper-programsp-1 (delete-all (*name*, *program-names*), *all-program-names*)

THEOREM: user-fnamep-user-fname
user-fnamep (user-fname (*x*))

THEOREM: logic-fname-user-fname-identity-formals
formals (*x*) → (logic-fname (user-fname (*x*)) = *x*)

THEOREM: s-programs-okp-s-construct-programs
l-proper-programsp-1 (*program-names*, *all-program-names*)
→ s-programs-okp (s-construct-programs (remove-duplicates (*program-names*)))

EVENT: Disable logic-fname-user-fname-identity-formals.

THEOREM: s-eval-l-eval-flag-run-flag-t
(s-good-statep (*s*, *c*)
∧ good-posp (*flag*, s-pos (*s*), s-body (s-prog (*s*)))
∧ s-all-temps-setp (*flag*, s-expr (*s*), temp-alist-to-set (s-temps (*s*)))
∧ s-all-progs-temps-setp (s-progs (*s*))
∧ l-eval (*flag*, s-expand-temps (*flag*, s-expr (*s*)), s-params (*s*), *c*)
∧ s-check-temps-setp (s-temps (*s*))
∧ (*flag* ≠ `'list`))
→ ((s-err-flag (s-eval (*flag*, *s*, *c*)) = `'run`)
∧ s-check-temps-setp (s-temps (s-eval (*flag*, *s*, *c*)))
∧ subsetp (s-collect-all-temps (*flag*, s-expr (*s*)),
temp-alist-to-set (s-temps (s-eval (*flag*, *s*, *c*)))))

THEOREM: l-proper-expr-s-all-temps-setp
l-proper-exprp (*flag*, *expr*, *prog-names*, *formals*)
→ s-all-temps-setp (*flag*, *expr*, *temp-set*)

THEOREM: l-proper-programsp-1-s-all-progs-temps-setp
l-proper-programsp-1 (*program-names*, *all-program-names*)
→ s-all-progs-temps-setp (s-construct-programs (*program-names*))

THEOREM: l-proper-programsp-s-all-progs-temps-setp
l-proper-programsp (*program-names*)
→ s-all-progs-temps-setp (s-construct-programs (*program-names*))

THEOREM: member-delete-all
(*x* ∈ delete-all (*y*, *l*))
= **if** *x* ∈ *l* **then** *x* ≠ *y*
else f endif

THEOREM: member-remove-duplicates
$(e \in \text{remove-duplicates}\,(x)) = (e \in x)$

THEOREM: s-proper-exprp-remove-duplicates
s-proper-exprp (*flag*, *expr*, remove-duplicates (*prog-names*), *formals*, *temp-list*)
= s-proper-exprp (*flag*, *expr*, *prog-names*, *formals*, *temp-list*)

THEOREM: l-proper-exprp-remove-duplicates
l-proper-exprp (*flag*, *expr*, remove-duplicates (*prog-names*), *formals*)
= l-proper-exprp (*flag*, *expr*, *prog-names*, *formals*)

THEOREM: l-proper-programs-remove-duplicates-arg2
l-proper-programsp-1 (*program-names*, remove-duplicates (*all-program-names*))
= l-proper-programsp-1 (*program-names*, *all-program-names*)

THEOREM: l-proper-programs-remove-duplicates-arg1
l-proper-programsp-1 (*program-names*, *all-program-names*)
→ l-proper-programsp-1 (remove-duplicates (*program-names*),
all-program-names)

THEOREM: all-litatoms-not-plist-delete-all
all-litatoms-not-plist (*list*) → all-litatoms-not-plist (delete-all (*x*, *list*))

THEOREM: all-litatoms-not-plist-remove-duplicates
all-litatoms-not-plist (*list*)
→ all-litatoms-not-plist (remove-duplicates (*list*))

THEOREM: all-litatoms-not-plist-lr-proper-programsp-1
l-proper-programsp-1 (*prog-names*, *all-prog-names*)
→ all-litatoms-not-plist (*prog-names*)

THEOREM: s-good-state-s-construct-programs
(l-proper-programsp (*prog-names*)
∧ l-proper-exprp (**t**, *expr*, *prog-names*, strip-cars (*alist*))
∧ all-litatoms (strip-cars (*alist*)))
→ s-good-statep (s-state (`'main`,
nil,
nil,
alist,
nil,
cons (list (`'main`, strip-cars (*alist*), **nil**, *expr*),
s-construct-programs (remove-duplicates (*prog-names*))),
`'run`),
c)

THEOREM: l-proper-programsp-remove-duplicates
l-proper-programsp (*program-names*)
→ l-proper-programsp (remove-duplicates (*program-names*))

THEOREM: logic->s-ok
(l-proper-exprp (**t**, *expr*, *program-names*, strip-cars (*alist*))
∧ l-proper-programsp (*program-names*)
∧ all-litatoms (strip-cars (*alist*))
∧ l-eval (**t**, *expr*, *alist*, *clock*))
→ (s-ans (s-eval (**t**, logic->s (*expr*, *alist*, *program-names*), *clock*))
= car (l-eval (**t**, *expr*, *alist*, *clock*)))

THEOREM: logic->s-ok-really
(l-proper-exprp (**t**, *expr*, *program-names*, strip-cars (*alist*))
∧ l-proper-programsp (*program-names*)
∧ all-litatoms (strip-cars (*alist*))
∧ v&c$ (**t**, *expr*, *alist*)
∧ (cdr (v&c$ (**t**, *expr*, *alist*)) < *clock*))
→ (s-ans (s-eval (**t**, logic->s (*expr*, *alist*, *program-names*), *clock*))
= car (v&c$ (**t**, *expr*, *alist*)))

EVENT: Make the library `"app-c-d-e"` and compile it.

Index