EVENT: Start with the library "c1".

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                    THE ASSEMBLY LANGUAGE                                 ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; The fields of the p-state are to be interpreted as follows:
;;    pc is the 0-based program counter;
;;    temp-stk is a stack for moving variables and computing values
;;    ctrl-stk is a list of frames with the local variables;
;;    data-segment is a var alist of global variables
;;    psw is 'run, 'halt, or some indication of error
```

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.

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)

```
;;(defn errorp (psw)
;;  (and (not (equal psw 'run))
;;       (not (equal psw 'halt))))
```

DEFINITION: definition (*name*, *alist*) = assoc (*name*, *alist*)

DEFINITION:
put-value $(val,\ name,\ alist)$ = put-assoc $(val,\ name,\ alist)$

DEFINITION: formal-vars (d) = cadr (d)

DEFINITION: temp-var-dcls (d) = caddr (d)

DEFINITION: program-body (d) = cdddr (d)

```
;;(defn local-vars (d) (append (formal-vars d) (listcars (temp-var-dcls d))))

;; Addresses are tagged "address pairs".  This differs from the earlier formulation
;; in having to worry about the tags.  Thus the pc is now (pc (subr . foo)).
```

DEFINITION: adp-name (adp) = car (adp)

EVENT: Disable adp-name.

DEFINITION: adp-offset (adp) = cdr (adp)

EVENT: Disable adp-offset.

THEOREM: adp-name-cons
adp-name (cons $(x,\ y)$) = x

THEOREM: adp-offset-cons
adp-offset (cons $(x,\ y)$) = y

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*))))

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*)))))

DEFINITION:
add-adp $(adp,\ n)$ = cons (adp-name (adp), 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*)

```
;;(defn sub1-adp (adp) (sub-adp adp 1))
;;
;;(defn add1-adp-adpp (x segment)
;;       (adpp (add1-adp x) segment))
;;
;;(defn add-adp-adpp (x n segment)
;;       (adpp (add-adp x n) segment))
```

DEFINITION:
fetch-adp (*adp*, *segment*)
= get (adp-offset (*adp*), value (adp-name (*adp*), *segment*))

```
;; Puts val in the named segment at addr.
```

DEFINITION:
deposit-adp (*val*, *adp*, *segment*)
= put-value (put (*val*, adp-offset (*adp*), value (adp-name (*adp*), *segment*)),
adp-name (*adp*),
segment)

```
;; These two functions are the analogues of get and put counting from
;; the right.  They are used in the definition of fetch and deposit on the
;; temp-stk.
```

DEFINITION: rget (*n*, *lst*) = get ((length (*lst*) − *n*) − 1, *lst*)

DEFINITION:
rput (*val*, *n*, *lst*) = put (*val*, (length (*lst*) − *n*) − 1, *lst*)

THEOREM: rput-preserves-length
(*y* < length (*z*)) → (length (rput (*x*, *y*, *z*)) = length (*z*))

EVENT: Disable rget.

EVENT: Disable rput.

```
;; Tagged objects
;; All Piton objects are tagged with their "type".  The form is (tag obj)
```

DEFINITION: tag (*type*, *obj*) = list (*type*, *obj*)

DEFINITION: type (*const*) = car (*const*)

DEFINITION: untag (*const*) = cadr (*const*)

THEOREM: type-expansion
type (cons (*x*, *y*)) = *x*

THEOREM: type-tag
type (tag (*x*, *y*)) = *x*

THEOREM: untag-tag
untag (tag (*x*, *y*)) = *y*

THEOREM: untag-cons
untag (list (*x*, *y*)) = *y*

```
;;(prove-lemma tag-length-2 (rewrite)
;;       (equal (cddr (tag x y)) nil)
;;      ((enable tag)))
```

EVENT: Disable type.

EVENT: Disable tag.

EVENT: Disable untag.

```
;; Tagged addresses

;;(defn 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*)

DEFINITION: add1-nat (*nat*) = tag ('nat, 1 + untag (*nat*))

THEOREM: untag-add1-nat
untag (add1-nat (*nat*)) = (1 + untag (*nat*))

```
;; Booleans
;; Piton uses the litatoms 't and 'f for the truth values.
```

DEFINITION: booleanp (*x*) = ((*x* = 't) ∨ (*x* = 'f))

DEFINITION:
bool (*x*)
= tag ('bool,
  if *x* **then** 't
  else 'f **endif**)

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**

```
;; Naturals
;; Notice that, where full Piton allows word-size to be a parameter, I am making it
;; a constant.
```

DEFINITION:
small-naturalp (*i*, *word-size*) = ((*i* ∈ **N**) ∧ (*i* < exp (2, *word-size*)))

EVENT: Disable small-naturalp.

DEFINITION:
bool-to-nat (*flg*)
= **if** *flg* = 'f **then** 0
else 1 **endif**

```
;; Integers
;; These functions are defined so that if given integerps they return integerps.
;; In particular, they never return (minus 0) though they may pass one through.
```

DEFINITION:
small-integerp (i, *word-size*)
= (integerp (i)
$\wedge$ ($\neg$ ilessp (i, $-$ exp (2, *word-size* $-$ 1)))
$\wedge$ ilessp (i, exp (2, *word-size* $-$ 1)))

EVENT: Disable small-integerp.

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**

```
;; I don't currently know which Piton integer ops I'll need on integers so I'm
;; leaving these out.

;; 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**

THEOREM: popn-add1
(popn (1 + n, push (x, y)) = popn (n, y))
$\wedge$ (popn (1 + n, cons (x, y)) = popn (n, y))

THEOREM: popn-zero
$\mathrm{popn}\,(0,\ x) = x$

THEOREM: popn-length
$(n = \mathrm{length}\,(\mathit{lst})) \rightarrow (\mathrm{popn}\,(n,\ \mathrm{append}\,(\mathit{lst},\ \mathit{stk})) = \mathit{stk})$

THEOREM: popn-plus
$(m = \mathrm{length}\,(\mathit{lst1})) \rightarrow (\mathrm{popn}\,(m + n,\ \mathrm{append}\,(\mathit{lst1},\ \mathit{lst2})) = \mathrm{popn}\,(n,\ \mathit{lst2}))$

DEFINITION: $\mathrm{top1}\,(\mathit{stk}) = \mathrm{top}\,(\mathrm{pop}\,(\mathit{stk}))$

DEFINITION: $\mathrm{top2}\,(\mathit{stk}) = \mathrm{top}\,(\mathrm{pop}\,(\mathrm{pop}\,(\mathit{stk})))$

THEOREM: pop-push
$\mathrm{pop}\,(\mathrm{push}\,(x,\ y)) = y$

THEOREM: pop-cons
$\mathrm{pop}\,(\mathrm{cons}\,(x,\ y)) = y$

THEOREM: car-cdr-push
$(\mathrm{car}\,(\mathrm{push}\,(x,\ y)) = x) \wedge (\mathrm{cdr}\,(\mathrm{push}\,(x,\ y)) = y)$

THEOREM: push-listp
$\mathrm{listp}\,(\mathrm{push}\,(x,\ y)) \wedge ((\mathrm{push}\,(x,\ y) \simeq \mathbf{nil}) = \mathbf{f})$

THEOREM: top-push
$\mathrm{top}\,(\mathrm{push}\,(x,\ y)) = x$

THEOREM: top1-push
$\mathrm{top1}\,(\mathrm{push}\,(x,\ \mathrm{push}\,(y,\ z))) = y$

THEOREM: top2-push
$\mathrm{top2}\,(\mathrm{push}\,(x,\ \mathrm{push}\,(y,\ \mathrm{push}\,(z,\ w)))) = z$

THEOREM: length-push
$\mathrm{length}\,(\mathrm{push}\,(x,\ y)) = (1 + \mathrm{length}\,(y))$

THEOREM: top-cons
$\mathrm{top}\,(\mathrm{cons}\,(x,\ y)) = x$

```
;;(prove-lemma top1-cons (rewrite)
;;      (equal (top1 (cons x (cons y z))) y))
;;
;;(prove-lemma top2-cons (rewrite)
;;      (equal (top2 (cons x (cons y (cons z w)))) z))
```

EVENT: Disable pop.

EVENT: Disable push.

EVENT: Disable top.

EVENT: Disable top1.

EVENT: Disable top2.

EVENT: Disable popn.

```
;; Labels
;; Notice that labelled instructions now have comments associated with them.

;;(defn dl (lab comment ins) (list 'dl lab comment ins))
```

DEFINITION: labelledp (x) = (car (x) = `'dl`)

THEOREM: labelledp-expansion
labelledp (cons (`'dl`, *args*)) = **t**

DEFINITION:
unlabel (x)
= **if** labelledp (x) **then** cadddr (x)
else x **endif**

THEOREM: unlabel-expansion
unlabel (cons (`'dl`, *args*)) = caddr (*args*)

THEOREM: unlabel-unlabelledp
$(x \neq$ `'dl`$) \rightarrow$ (unlabel (cons (x, y)) = cons (x, y))

EVENT: Disable unlabel.

DEFINITION:
find-labelp $(x,$ *lst*$)$
= **if** *lst* $\simeq$ **nil then f**
elseif labelledp (car (*lst*)) $\wedge$ (x = cadr (car (*lst*))) **then t**
else find-labelp $(x,$ cdr (*lst*)$)$ **endif**

```
;; This function counts the number of items in the list before the first one
;; labelled with x.  This is my get-pc function.
```

DEFINITION:
find-label $(x,\ lst)$
$=$ **if** $lst \simeq$ **nil** **then** `0`
elseif labelledp (car (lst)) $\wedge$ $(x =$ cadr (car $(lst)))$ **then** `0`
else $1 +$ find-label $(x,$ cdr $(lst))$ **endif**

```
;;(defn all-find-labelp (lab-lst lst)
;;  (if (nlistp lab-lst)
;;      t
;;      (and (find-labelp (car lab-lst) lst)
;;           (all-find-labelp (cdr lab-lst) lst))))
```

DEFINITION:
pc $(label,\ program)$
$=$ tag (`'pc`, cons (car $(program)$, find-label $(label,$ program-body $(program))))$

```
;; Piton primitive objects
```

DEFINITION:
p-objectp $(x,\ p)$
$=$ (listp (x)
$\wedge$ (cddr $(x) =$ **nil**)
$\wedge$ **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 $=$ *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 $(type,\ x,\ p) = ((\text{type}\ (x) = type) \wedge$ p-objectp $(x,\ p))$

DEFINITION: add1-p-pc (p) = add1-addr (p-pc (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*)

THEOREM: bindings-frame
(bindings (cons (x, y)) = x) $\wedge$ (bindings (p-frame (x, y)) = x)

THEOREM: ret-pc-frame
(ret-pc (list (x, y)) = y) $\wedge$ (ret-pc (p-frame (x, y)) = y)

EVENT: Disable p-frame.

EVENT: Disable bindings.

EVENT: Disable ret-pc.

DEFINITION:
p-frame-size (*frame*) = (2 + length (bindings (*frame*)))

EVENT: Disable p-frame-size.

DEFINITION:
p-ctrl-stk-size (*ctrl-stk*)
= **if** *ctrl-stk* $\simeq$ **nil then** 0
else p-frame-size (top (*ctrl-stk*))
+ p-ctrl-stk-size (cdr (*ctrl-stk*)) **endif**

```
;;(defn 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*))

DEFINITION:
first-n (n, x)
= **if** $n \simeq$ `0` **then nil**
else cons (car (x), first-n ($n - 1$, cdr (x))) **endif**

THEOREM: first-n-add1
(first-n ($1 + n$, push (x, y)) = cons (x, first-n (n, y)))
$\wedge$ (first-n ($1 + n$, cons (x, y)) = cons (x, first-n (n, y)))
$\wedge$ (first-n (`0`, y) = **nil**)

THEOREM: first-n-plus-append
(n = length (*lst1*))
$\rightarrow$ (first-n ($n + m$, append (*lst1*, *lst2*)) = append (*lst1*, first-n (m, *lst2*)))

EVENT: Disable first-n-plus-append.

THEOREM: first-n-append2
(plistp (*lst1*) $\wedge$ (n = length (*lst1*)))
$\rightarrow$ (first-n (n, append (*lst1*, *lst2*)) = *lst1*)

EVENT: Disable first-n-append2.

DEFINITION:
pairlist (*lst1*, *lst2*)
= **if** *lst1* $\simeq$ **nil then nil**
else cons (cons (car (*lst1*), car (*lst2*)),
pairlist (cdr (*lst1*), cdr (*lst2*))) **endif**

THEOREM: pairlist-plistp
plistp (pairlist (x, y))

THEOREM: length-pairlist
length (pairlist (x, y)) = length (x)

THEOREM: pairlist-distributes
(length (*lst1*) = length (*lst3*))
$\rightarrow$ (pairlist (append (*lst1*, *lst2*), append (*lst3*, *lst4*))
= append (pairlist (*lst1*, *lst3*), pairlist (*lst2*, *lst4*)))

EVENT: Disable pairlist-distributes.

```
;; Given formal-vars '(X Y Z) we first push X, then Y, then Z.
```

DEFINITION:
pair-formal-vars-with-actuals (*formal-vars*, *temp-stk*)
= pairlist (*formal-vars*, reverse (first-n (length (*formal-vars*), *temp-stk*)))

```
;; This function converts from the external form of the temp-alists to
;; the internal.  That is, in programs, the temp alist is written as
;; (... (tempi vali) ...).  Internally, this is stored as
;; (... (temp1 . vali) ...) to allow them to be accessed as all the
;; other alists.
```

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**

THEOREM: length-pair-temps-with-initial-values
length (pair-temps-with-initial-values (*lst*)) = length (*lst*)

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)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                       ;;
;;                    THE PITON INSTRUCTIONS                             ;;
;;                                                                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;  (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)

```
;  (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**

```
;  (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')

```
;  (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)

```
;  (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)

```
;  (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)

```
;  (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)

```
;  (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)

```
;  (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)

```
;  (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`)

```
;   (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`)

```
;   (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`)

```
;  (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`)

```
;   (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)

```
;  (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)

```
; (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 $x \neq 0$ **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)

```
; (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 $x = 0$
case = *not-zero*
then $x \neq 0$
case = *neg*
then negativep (*x*)
case = *not-neg*
then $\neg$ negativep (*x*)
case = *pos*
then $(x \in \mathbf{N}) \wedge (x \neq 0)$
otherwise $(x = 0) \vee$ 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)

```
; (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)

```
;   (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)

;   (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))
    ∧   listp (pop (p-temp-stk (p)))
    ∧   (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)

;   (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**

```
;;  (SUB-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)

;   (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**

```
;   (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)

```
;   (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)

```
;   (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)

```
;   (ADD-NAT)           Pop top and top1 off of temp-stk.
;                       Both top and top1 must be NATs.  Push
;                       their NAT sum.
```

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

;   (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.


DEFINITION:
p-sub-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)
    ∧  (untag (top1 (p-temp-stk (p))) ≮ 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**

```
;   (SUB1-NAT)          Pop top off of temp-stk.
;                       Top must be a NAT other than
;                       0.  Push top-1.
```

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)

```
;   (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)

;   (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)

```
;   (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`)

```
;;(defn piton-opcodes nil
;;  '(call ret push-constant push-local push-global
;;    push-temp-stk-index pop pop* pop-local pop-global
;;    fetch-temp-stk deposit-temp-stk jump jump-case set-local
;;    test-nat-and-jump test-int-and-jump test-bool-and-jump no-op
;;    eq add-int-with-carry sub-int sub-int-with-carry neg-int
;;    lt-int int-to-nat add-nat sub-nat sub1-nat lt-nat 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 = *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-temp-stk-index*
 then p-push-temp-stk-index-okp (*ins*, *p*)
case = *pop*
 then p-pop-okp (*ins*, *p*)
case = *pop**
 then p-pop*-okp (*ins*, *p*)
case = *pop-local*
 then p-pop-local-okp (*ins*, *p*)
case = *pop-global*
 then p-pop-global-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 = *set-local*
 then p-set-local-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 = *no-op*
 then p-no-op-okp (*ins*, *p*)
case = *eq*
 then p-eq-okp (*ins*, *p*)
case = *add-int-with-carry*
 then p-add-int-with-carry-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 = *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 = *sub-nat*
then p-sub-nat-okp (*ins*, *p*)
case = *sub1-nat*
then p-sub1-nat-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

```
; 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 = *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-temp-stk-index*
then p-push-temp-stk-index-step (*ins*, *p*)
case = *pop*
then p-pop-step (*ins*, *p*)
case = *pop**
then p-pop*-step (*ins*, *p*)
case = *pop-local*
then p-pop-local-step (*ins*, *p*)

case = *pop-global*
 then p-pop-global-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 = *set-local*
 then p-set-local-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 = *no-op*
 then p-no-op-step (*ins*, *p*)
case = *eq*
 then p-eq-step (*ins*, *p*)
case = *add-int-with-carry*
 then p-add-int-with-carry-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 = *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 = *sub-nat*
 then p-sub-nat-step (*ins*, *p*)
case = *sub1-nat*
 then p-sub1-nat-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**

EVENT: Disable p-step.

```
; 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**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                          Lemmas on Piton                                 ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

EVENT: Disable p-step.

EVENT: Disable p-halt.

EVENT: Disable p-ins-step.

EVENT: Disable p-ins-okp.

EVENT: Disable p-step1.

EVENT: Disable x-y-error-msg.

THEOREM: p-step-expansion

```
p-step (p-state (pc,
                 ctr-stk,
                 temp-stk,
                 prog-seg,
                 data-seg,
                 p-max-ctrl-stk-size,
                 p-max-temp-stk-size,
                 p-word-size,
                 psw))
=  if psw = 'run
   then p-step1 (p-current-instruction (p-state (pc,
                                                 ctr-stk,
                                                 temp-stk,
                                                 prog-seg,
                                                 data-seg,
                                                 p-max-ctrl-stk-size,
                                                 p-max-temp-stk-size,
                                                 p-word-size,
                                                 psw)),
                                        p-state (pc,
                                                 ctr-stk,
                                                 temp-stk,
                                                 prog-seg,
                                                 data-seg,
                                                 p-max-ctrl-stk-size,
                                                 p-max-temp-stk-size,
                                                 p-word-size,
                                                 psw))
   else p-state (pc,
                 ctr-stk,
```

temp-stk,
prog-seg,
data-seg,
p-max-ctrl-stk-size,
p-max-temp-stk-size,
p-word-size,
psw) **endif**

THEOREM: p-ins-step-expansion
p-ins-step (cons (pack (*opr*), *args*), *p*)
= **case on** pack (*opr*):
case = *call*
then p-call-step (cons (pack (*opr*), *args*), *p*)
case = *ret*
then p-ret-step (cons (pack (*opr*), *args*), *p*)
case = *push-constant*
then p-push-constant-step (cons (pack (*opr*), *args*), *p*)
case = *push-local*
then p-push-local-step (cons (pack (*opr*), *args*), *p*)
case = *push-global*
then p-push-global-step (cons (pack (*opr*), *args*), *p*)
case = *push-temp-stk-index*
then p-push-temp-stk-index-step (cons (pack (*opr*), *args*), *p*)
case = *pop*
then p-pop-step (cons (pack (*opr*), *args*), *p*)
case = *pop**
then p-pop*-step (cons (pack (*opr*), *args*), *p*)
case = *pop-local*
then p-pop-local-step (cons (pack (*opr*), *args*), *p*)
case = *pop-global*
then p-pop-global-step (cons (pack (*opr*), *args*), *p*)
case = *fetch-temp-stk*
then p-fetch-temp-stk-step (cons (pack (*opr*), *args*), *p*)
case = *deposit-temp-stk*
then p-deposit-temp-stk-step (cons (pack (*opr*), *args*), *p*)
case = *jump*
then p-jump-step (cons (pack (*opr*), *args*), *p*)
case = *jump-case*
then p-jump-case-step (cons (pack (*opr*), *args*), *p*)
case = *set-local*
then p-set-local-step (cons (pack (*opr*), *args*), *p*)
case = *test-nat-and-jump*
then p-test-nat-and-jump-step (cons (pack (*opr*), *args*), *p*)
case = *test-int-and-jump*

then p-test-int-and-jump-step (cons (pack (*opr*), *args*), *p*)
case = *test-bool-and-jump*
then p-test-bool-and-jump-step (cons (pack (*opr*), *args*), *p*)
case = *no-op*
then p-no-op-step (cons (pack (*opr*), *args*), *p*)
case = *eq*
then p-eq-step (cons (pack (*opr*), *args*), *p*)
case = *add-int-with-carry*
then p-add-int-with-carry-step (cons (pack (*opr*), *args*), *p*)
case = *sub-int*
then p-sub-int-step (cons (pack (*opr*), *args*), *p*)
case = *sub-int-with-carry*
then p-sub-int-with-carry-step (cons (pack (*opr*), *args*), *p*)
case = *neg-int*
then p-neg-int-step (cons (pack (*opr*), *args*), *p*)
case = *lt-int*
then p-lt-int-step (cons (pack (*opr*), *args*), *p*)
case = *int-to-nat*
then p-int-to-nat-step (cons (pack (*opr*), *args*), *p*)
case = *add-nat*
then p-add-nat-step (cons (pack (*opr*), *args*), *p*)
case = *sub-nat*
then p-sub-nat-step (cons (pack (*opr*), *args*), *p*)
case = *sub1-nat*
then p-sub1-nat-step (cons (pack (*opr*), *args*), *p*)
case = *or-bool*
then p-or-bool-step (cons (pack (*opr*), *args*), *p*)
case = *and-bool*
then p-and-bool-step (cons (pack (*opr*), *args*), *p*)
case = *not-bool*
then p-not-bool-step (cons (pack (*opr*), *args*), *p*)
otherwise p-halt (*p*, `'run`) **endcase**

THEOREM: p-ins-okp-expansion
p-ins-okp (cons (pack (*opr*), *args*), *p*)
= **case on** pack (*opr*):
case = *call*
then p-call-okp (cons (pack (*opr*), *args*), *p*)
case = *ret*
then p-ret-okp (cons (pack (*opr*), *args*), *p*)
case = *push-constant*
then p-push-constant-okp (cons (pack (*opr*), *args*), *p*)
case = *push-local*
then p-push-local-okp (cons (pack (*opr*), *args*), *p*)

case = *push-global*
 then p-push-global-okp (cons (pack (*opr*), *args*), *p*)
case = *push-temp-stk-index*
 then p-push-temp-stk-index-okp (cons (pack (*opr*), *args*), *p*)
case = *pop*
 then p-pop-okp (cons (pack (*opr*), *args*), *p*)
case = *pop**
 then p-pop*-okp (cons (pack (*opr*), *args*), *p*)
case = *pop-local*
 then p-pop-local-okp (cons (pack (*opr*), *args*), *p*)
case = *pop-global*
 then p-pop-global-okp (cons (pack (*opr*), *args*), *p*)
case = *fetch-temp-stk*
 then p-fetch-temp-stk-okp (cons (pack (*opr*), *args*), *p*)
case = *deposit-temp-stk*
 then p-deposit-temp-stk-okp (cons (pack (*opr*), *args*), *p*)
case = *jump*
 then p-jump-okp (cons (pack (*opr*), *args*), *p*)
case = *jump-case*
 then p-jump-case-okp (cons (pack (*opr*), *args*), *p*)
case = *set-local*
 then p-set-local-okp (cons (pack (*opr*), *args*), *p*)
case = *test-nat-and-jump*
 then p-test-nat-and-jump-okp (cons (pack (*opr*), *args*), *p*)
case = *test-int-and-jump*
 then p-test-int-and-jump-okp (cons (pack (*opr*), *args*), *p*)
case = *test-bool-and-jump*
 then p-test-bool-and-jump-okp (cons (pack (*opr*), *args*), *p*)
case = *no-op*
 then p-no-op-okp (cons (pack (*opr*), *args*), *p*)
case = *eq*
 then p-eq-okp (cons (pack (*opr*), *args*), *p*)
case = *add-int-with-carry*
 then p-add-int-with-carry-okp (cons (pack (*opr*), *args*), *p*)
case = *sub-int*
 then p-sub-int-okp (cons (pack (*opr*), *args*), *p*)
case = *sub-int-with-carry*
 then p-sub-int-with-carry-okp (cons (pack (*opr*), *args*), *p*)
case = *neg-int*
 then p-neg-int-okp (cons (pack (*opr*), *args*), *p*)
case = *lt-int*
 then p-lt-int-okp (cons (pack (*opr*), *args*), *p*)
case = *int-to-nat*
 then p-int-to-nat-okp (cons (pack (*opr*), *args*), *p*)

case = *add-nat*
then p-add-nat-okp (cons (pack (*opr*), *args*), *p*)
case = *sub-nat*
then p-sub-nat-okp (cons (pack (*opr*), *args*), *p*)
case = *sub1-nat*
then p-sub1-nat-okp (cons (pack (*opr*), *args*), *p*)
case = *or-bool*
then p-or-bool-okp (cons (pack (*opr*), *args*), *p*)
case = *and-bool*
then p-and-bool-okp (cons (pack (*opr*), *args*), *p*)
case = *not-bool*
then p-not-bool-okp (cons (pack (*opr*), *args*), *p*)
otherwise f endcase

THEOREM: p-0-unwinding-lemma
p (*s*, 0) = *s*

THEOREM: p-plus-lemma
p (*state*, *j* + *k*) = p (p (*state*, *j*), *k*)

THEOREM: p-add1
p (*state*, 1 + *x*) = p (p (*state*, *x*), 1)

EVENT: Disable p-add1.

THEOREM: p-rearrange-times-lemma
p (p (*state*, *k*), *j*) = p (p (*state*, *j*), *k*)

EVENT: Disable p-rearrange-times-lemma.

THEOREM: p-add1-3
p (*state*, 1 + *x*) = p (p-step (*state*), *x*)

THEOREM: find-label-append
(¬ find-labelp (*label*, *code1*))
→ (find-label (*label*, append (*code1*, *code2*))
= (length (*code1*) + find-label (*label*, *code2*)))

EVENT: Disable find-label-append.

THEOREM: find-labelp-append2
find-labelp (*n*, append (*lst1*, *lst2*))
= (find-labelp (*n*, *lst1*) ∨ find-labelp (*n*, *lst2*))

EVENT: Disable find-labelp-append2.

THEOREM: rget-inverts-rput
rget (*n*, rput (*value*, *n*, *lst*)) = *value*

THEOREM: rput-same-value-doesnt-disturb-temp-stk
(*n* = length (*temp-stk*))
→ (rput (*value*, *n*, append (*lst*, cons (*value*, *temp-stk*)))
= append (*lst*, cons (*value*, *temp-stk*)))

EVENT: Disable ilessp.

EVENT: Disable not-bool.

EVENT: Disable and-bool.

EVENT: Disable or-bool.

EVENT: Disable inegate.

EVENT: Disable fix-small-integer.

EVENT: Disable iplus.

EVENT: Disable idifference.

EVENT: Make the library `"c2"`.

Index