Event: Start with the library "c2".

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;                     PREFIX ACCESSOR FUNCTIONS                      ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; These functions simply allow the accessing of various components
;; of Micro-Gypsy structures.
```

Definition: loop-body (*stmt*) = cadr (*stmt*)

Definition: prog2-left-branch (*stmt*) = cadr (*stmt*)

Definition: prog2-right-branch (*stmt*) = caddr (*stmt*)

Definition: signalled-condition (*stmt*) = cadr (*stmt*)

Theorem: signalled-condition-expansion2
signalled-condition (cons (*signal*, *lst*)) = car (*lst*)

Definition: if-condition (*stmt*) = cadr (*stmt*)

Definition: if-true-branch (*stmt*) = caddr (*stmt*)

Definition: if-false-branch (*stmt*) = cadddr (*stmt*)

Definition: begin-body (*stmt*) = cadr (*stmt*)

Definition: when-labels (*stmt*) = caddr (*stmt*)

Definition: when-handler (*stmt*) = cadddr (*stmt*)

Definition: call-name (*stmt*) = cadr (*stmt*)

Theorem: call-name-expansion
call-name (cons (*proc-call*, cons (*name*, *y*))) = *name*

Definition: call-actuals (*stmt*) = caddr (*stmt*)

Definition: call-conds (*stmt*) = cadddr (*stmt*)

Definition: formal-type (*exp*) = cadr (*exp*)

Definition: formal-initial-value (*local*) = caddr (*local*)

```
;; current count 13
```

DEFINITION:
PREDEFINED-PROCEDURE-LIST

```
=  '(mg-simple-variable-assignment
     mg-simple-constant-assignment mg-simple-variable-eq
     mg-simple-constant-eq mg-integer-le
     mg-integer-unary-minus mg-integer-add
     mg-integer-subtract mg-boolean-or mg-boolean-and
     mg-boolean-not mg-index-array mg-array-element-assignment)
```

```
;; The procedure list is an alist of the form
;; ( ... (name proc-definition) ... ).  Notice that this will change when
;; I return to prefix form since the name is not in the car position in that
;; representation.
```

DEFINITION:
predefined-procp (*name*) = (*name* $\in$ PREDEFINED-PROCEDURE-LIST)

DEFINITION:
user-defined-procp (*name*, *proc-list*)
= ((*name* $\notin$ PREDEFINED-PROCEDURE-LIST) $\land$ definedp (*name*, *proc-list*))

DEFINITION:
defined-procp (*name*, *proc-list*)
= (predefined-procp (*name*) $\lor$ user-defined-procp (*name*, *proc-list*))

DEFINITION: fetch-def (*name*, *proc-list*) = assoc (*name*, *proc-list*)

DEFINITION:
fetch-called-def (*stmt*, *proc-list*) = fetch-def (call-name (*stmt*), *proc-list*)

```
;; (name data-formals cond-formals data-locals cond-locals body)
```

DEFINITION: def-name (*def*) = car (*def*)

EVENT: Disable def-name.

DEFINITION: def-formals (*def*) = cadr (*def*)

EVENT: Disable def-formals.

DEFINITION: def-conds (*def*) = caddr (*def*)

EVENT: Disable def-conds.

DEFINITION: def-locals (*def*) = cadddr (*def*)

EVENT: Disable def-locals.

DEFINITION: def-cond-locals (*def*) = caddddr (*def*)

EVENT: Disable def-cond-locals.

DEFINITION: def-body (*def*) = cadddddr (*def*)

EVENT: Disable def-body.

DEFINITION: array-elemtype (*type*) = cadr (*type*)

EVENT: Disable array-elemtype.

DEFINITION: array-length (*type*) = caddr (*type*)

EVENT: Disable array-length.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                           ;;
;;                          THE RECOGNIZER                                   ;;
;;                                                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

DEFINITION:
RESERVED-NAMES-LIST = `'(leave normal routineerror)`

DEFINITION: reserved-word (*wd*) = (*wd* $\in$ RESERVED-NAMES-LIST)

DEFINITION:
ok-mg-namep (*ident*)
= (litatom (*ident*)
$\wedge$ (45 $\notin$ unpack (*ident*))
$\wedge$ ($\neg$ reserved-word (*ident*)))

EVENT: Disable ok-mg-namep.

```
;; Notice that the word size is selected to be exactly that of
;; Piton.  This eliminates some of the difficulties of the mappings.
```

DEFINITION: MG-WORD-SIZE = 32

EVENT: Introduce the function symbol *mg-max-ctrl-stk-size* of 0 arguments.

AXIOM: mg-max-ctrl-stk-size-small-naturalp
(MG-MAX-CTRL-STK-SIZE $\in$ **N**)
$\land$ (MG-MAX-CTRL-STK-SIZE $<$ exp (2, MG-WORD-SIZE))

EVENT: Introduce the function symbol *mg-max-temp-stk-size* of 0 arguments.

AXIOM: mg-max-temp-stk-size-numberp
(MG-MAX-TEMP-STK-SIZE $\in$ **N**)
$\land$ (MG-MAX-TEMP-STK-SIZE $<$ exp (2, MG-WORD-SIZE))

```
;; Notice also the I can simply use the Piton function
;; small-integerp rather than define a specific MG notion of what
;; is an acceptable integer.
```

DEFINITION: MAXINT = (exp (2, MG-WORD-SIZE $-$ 1) $-$ 1)

DEFINITION: MININT = ($-$ exp (2, MG-WORD-SIZE $-$ 1))

```
;; An integer literal is of the form (INT-MG n) where n is in the
;; range (minint..maxint).
```

DEFINITION:
int-literalp (*exp*)
= (('`int-mg` = car (*exp*))
$\land$ length-plistp (*exp*, 2)
$\land$ small-integerp (cadr (*exp*), MG-WORD-SIZE))

DEFINITION:
boolean-literalp (*exp*)
= (('`boolean-mg` = car (*exp*))
$\land$ length-plistp (*exp*, 2)
$\land$ (cadr (*exp*) $\in$ `'(true-mg false-mg)`))

Definition:

```
character-literalp (exp)
$=$ (('character-mg $=$ car (exp))
     $\wedge$ length-plistp (exp, 2)
     $\wedge$ (cadr (exp) $\in \mathbf{N}$)
     $\wedge$ (cadr (exp) $\leq$ 127))
```

Event: Disable int-literalp.

Event: Disable boolean-literalp.

Event: Disable character-literalp.

Definition:

```
simple-mg-type-refp (typref)
$=$ (typref $\in$ '(int-mg boolean-mg character-mg))
```

Definition:

```
array-mg-type-refp (typref)
$=$ (length-plistp (typref, 3)
     $\wedge$ (car (typref) $=$ 'array-mg)
     $\wedge$ simple-mg-type-refp (array-elemtype (typref))
     $\wedge$ (array-length (typref) $\not\simeq$ 0))
```

Event: Disable array-mg-type-refp.

Definition:

```
mg-type-refp (typref)
$=$ (simple-mg-type-refp (typref) $\vee$ array-mg-type-refp (typref))
```

Definition:

```
simple-typed-literalp (lit, type)
$=$ if type $=$ 'int-mg then int-literalp (lit)
    elseif type $=$ 'boolean-mg then boolean-literalp (lit)
    elseif type $=$ 'character-mg then character-literalp (lit)
    else f endif
```

Definition:

```
simple-typed-literal-plistp (lst, type)
$=$ if lst $\simeq$ nil then lst $=$ nil
    else simple-typed-literalp (car (lst), type)
         $\wedge$ simple-typed-literal-plistp (cdr (lst), type) endif
```

Definition:
array-literalp (*exp*, *length*, *elemtype*)

```
=    (simple-typed-literal-plistp (exp, elemtype)
       ∧   (length (exp) = length))
```

Event: Disable array-literalp.

Definition:
ok-mg-array-value (*exp*, *type*)
= array-literalp (*exp*, array-length (*type*), array-elemtype (*type*))

Event: Disable ok-mg-array-value.

Definition:
ok-mg-valuep (*exp*, *type*)

```
=    if simple-mg-type-refp (type) then simple-typed-literalp (exp, type)
     elseif array-mg-type-refp (type) then ok-mg-array-value (exp, type)
     else f endif
```

Event: Disable ok-mg-valuep.

```
;; The recognizer has a structure called the name-alist of the following
;; form:
;; (... (name type) ...)
;; This allows the identification of the types of variables.  It should be
;; the case that the values of the variables in the meaning alist correspond
;; to their types on the name-alist.
```

Definition: m-type (x) = cadr (x)

Definition: get-m-type (*name*, *alist*) = m-type (assoc (*name*, *alist*))

Definition:
has-array-type (*name*, *alist*)
= (car (get-m-type (*name*, *alist*)) = `'array-mg`)

Definition:
mg-name-alist-elementp (x)
= (ok-mg-namep (car (x)) ∧ mg-type-refp (m-type (x)))

Definition:
mg-name-alistp (*alist*)

```
=    if alist ≃ nil then alist = nil
     else mg-name-alist-elementp (car (alist))
            ∧   mg-name-alistp (cdr (alist)) endif
```

Definition: identifierp (*name*) = ok-mg-namep (*name*)

Definition:
defined-identifierp (*name*, *alist*)
= (identifierp (*name*) ∧ definedp (*name*, *alist*))

```
;; I'm restricting the if-condition to be a boolean identifier rather
;; than a boolean expression just so that the compilation will take a
;; fixed number of steps.  That is, I want every expression within the
;; code to be a reference to a variable.  This can obviously be relaxed
;; but is general since I allow the initialization of locals.  Thus,
;; any literal values can be stored in local variables.

;; >>> These should probably be removed in favor of one function with a
;;     type argument.
```

Definition:
boolean-identifierp (*name*, *alist*)
= (identifierp (*name*) ∧ (get-m-type (*name*, *alist*) = `'boolean-mg`))

Definition:
int-identifierp (*name*, *alist*)
= (identifierp (*name*) ∧ (get-m-type (*name*, *alist*) = `'int-mg`))

Definition:
character-identifierp (*name*, *alist*)
= (identifierp (*name*) ∧ (get-m-type (*name*, *alist*) = `'character-mg`))

Definition:
array-identifierp (*name*, *alist*)
= (defined-identifierp (*name*, *alist*) ∧ has-array-type (*name*, *alist*))

Event: Disable boolean-identifierp.

Event: Disable int-identifierp.

Event: Disable character-identifierp.

Event: Disable array-identifierp.

Definition:
simple-identifierp (*name*, *alist*)

```
=  (boolean-identifierp (name, alist)
     ∨  int-identifierp (name, alist)
     ∨  character-identifierp (name, alist))
```

DEFINITION:

```
simple-typed-identifierp (ident, type, alist)
=  if type = 'int-mg then int-identifierp (ident, alist)
   elseif type = 'boolean-mg then boolean-identifierp (ident, alist)
   elseif type = 'character-mg
   then character-identifierp (ident, alist)
   else f endif
```

THEOREM: int-identifierp-simple
int-identifierp $(x,\ alist)$ → simple-identifierp $(x,\ alist)$

THEOREM: boolean-identifierp-simple
boolean-identifierp $(x,\ alist)$ → simple-identifierp $(x,\ alist)$

THEOREM: character-identifierp-simple
character-identifierp $(x,\ alist)$ → simple-identifierp $(x,\ alist)$

EVENT: Disable simple-identifierp.

EVENT: Disable simple-typed-identifierp.

THEOREM: int-identifierp-implies-definedp
int-identifierp $(x,\ alist)$ → definedp $(x,\ alist)$

EVENT: Disable int-identifierp-implies-definedp.

THEOREM: boolean-identifierp-implies-definedp
boolean-identifierp $(x,\ alist)$ → definedp $(x,\ alist)$

EVENT: Disable boolean-identifierp-implies-definedp.

THEOREM: character-identifierp-implies-definedp
character-identifierp $(x,\ alist)$ → definedp $(x,\ alist)$

EVENT: Disable character-identifierp-implies-definedp.

THEOREM: simple-identifierp-implies-definedp
simple-identifierp $(x,\ alist)$ → definedp $(x,\ alist)$

Event: Disable simple-identifierp-implies-definedp.

Theorem: simple-typed-identifierp-implies-definedp
simple-typed-identifierp $(x,\ \mathit{type},\ \mathit{alist}) \rightarrow$ definedp $(x,\ \mathit{alist})$

Event: Disable simple-typed-identifierp-implies-definedp.

Theorem: array-identifierp-implies-definedp
array-identifierp $(x,\ \mathit{alist}) \rightarrow$ definedp $(x,\ \mathit{alist})$

Event: Disable array-identifierp-implies-definedp.

```
;; I have decided to make references to individual array elements impossible.
;; That is, a user program can only pass a whole array to a subroutine.  Arrays
;; are viewed as abstract data types and the only accesses to them are via
;; the predefined functions which select elements and write elements.  This will
;; guarantee that the param list of a routine is unchanged from the non-structured
;; situation.
```

Definition:
identifier-plistp (lst)
$=$ **if** $\mathit{lst} \simeq$ **nil** **then** $\mathit{lst} =$ **nil**
else identifierp (car $(\mathit{lst})) \wedge$ identifier-plistp (cdr $(\mathit{lst}))$ **endif**

Theorem: identifier-plistp-plistp
identifier-plistp $(x) \rightarrow$ plistp (x)

Event: Disable identifier-plistp-plistp.

Theorem: identifier-plistp-distributes
(identifier-plistp $(x) \wedge$ identifier-plistp $(y))$
$\rightarrow$ identifier-plistp (append $(x,\ y))$

Theorem: leave-not-in-identifier-plistp
identifier-plistp $(\mathit{lst}) \rightarrow$ ('leave $\notin \mathit{lst})$

Event: Disable leave-not-in-identifier-plistp.

Definition:
cond-identifierp $(x,\ \mathit{cond\text{-}list})$
$=$ $((x =$ 'routineerror$) \vee$ (identifierp $(x) \wedge (x \in \mathit{cond\text{-}list})))$

Event: Disable cond-identifierp.

Definition:
cond-identifier-plistp (*lst*, *cond-list*)
$=$ **if** *lst* $\simeq$ **nil** **then** *lst* $=$ **nil**
else cond-identifierp (car (*lst*), *cond-list*)
$\land$ cond-identifier-plistp (cdr (*lst*), *cond-list*) **endif**

Event: Disable cond-identifier-plistp.

Theorem: cond-identifier-plistp-cond-subsetp
cond-identifier-plistp (*lst*, *cond-list*) $\rightarrow$ cond-subsetp (*lst*, *cond-list*)

Event: Disable cond-identifier-plistp-cond-subsetp.

Theorem: normal-not-in-cond-identifier-plistp
cond-identifier-plistp (*lst*, *cond-list*) $\rightarrow$ (`'normal` $\notin$ *lst*)

Event: Disable normal-not-in-cond-identifier-plistp.

Theorem: adding-element-preserves-cond-identifier-plistp
cond-identifier-plistp (y, *cond-list*)
$\rightarrow$ cond-identifier-plistp (y, cons (x, *cond-list*))

Event: Disable adding-element-preserves-cond-identifier-plistp.

Theorem: superset-preserves-cond-identifier-plistp
(subset (x, y) $\land$ cond-identifier-plistp (*lst*, x))
$\rightarrow$ cond-identifier-plistp (*lst*, y)

Event: Disable superset-preserves-cond-identifier-plistp.

Definition:
nonempty-cond-identifier-plistp (*lst*, *cond-list*)
$=$ (cond-identifier-plistp (*lst*, *cond-list*) $\land$ (*lst* $\neq$ **nil**))

Definition:
ok-condition (*exp*, *cond-list*)
$=$ ((*exp* $=$ `'routineerror`)
$\lor$ ((ok-mg-namep (*exp*) $\lor$ (*exp* $=$ `'leave`))
$\land$ (*exp* $\in$ *cond-list*)))

Theorem: ok-condition-litatom

```
ok-condition (exp, cond-list) → litatom (exp)
```

Event: Disable ok-condition-litatom.

```
;; Notice that I'm not allowing references to individual array elements, so
;; this doesn't change from the previous version.
```

Definition:

```
ok-actual-params-list (lst, alist)
=   if lst ≃ nil then lst = nil
    else defined-identifierp (car (lst), alist)
         ∧  ok-actual-params-list (cdr (lst), alist) endif
```

```
;; The formal is of the form (name kind typeref);  kind is in {var-mg const-mg}.

;; There is a match between actual and formal if the actual is a literal of the type of the
;; formal or is an identifier of exactly the same type as the formal.
;; This will have to be relaxed for subtypes when I add them.
```

Definition:

```
ok-identifier-actual (actual, formal, alist)
=   (identifierp (actual)
     ∧  (get-m-type (actual, alist) = formal-type (formal)))
```

Definition:

```
data-params-match (actual, formal, alist)
=   ok-identifier-actual (actual, formal, alist)
```

Definition:

```
data-param-lists-match (actuals, formals, alist)
=   if (actuals ≃ nil) ∨ (formals ≃ nil)
    then (formals = nil) ∧ (actuals = nil)
    else data-params-match (car (actuals), car (formals), alist)
         ∧  data-param-lists-match (cdr (actuals),
                                     cdr (formals),
                                     alist) endif
```

Theorem: data-param-lists-match-in-length

```
data-param-lists-match (actuals, formals, alist)
→   (length (formals) = length (actuals))
```

Event: Disable data-param-lists-match-in-length.

Definition:
cond-params-match $(\mathit{cond\text{-}actuals},\ \mathit{conds})$
$=\quad (\mathrm{length}\,(\mathit{cond\text{-}actuals}) = \mathrm{length}\,(\mathit{conds}))$

Theorem: list-count-decreases

$$\begin{aligned}
&((x = \mathrm{car}\,(\mathit{stmt})) \land (x \neq 0))\\
\rightarrow\quad &(((\mathrm{count}\,(\mathrm{cadr}\,(\mathit{stmt})) < \mathrm{count}\,(\mathit{stmt})) = \mathbf{t})\\
&\quad\land\quad ((\mathrm{count}\,(\mathrm{caddr}\,(\mathit{stmt})) < \mathrm{count}\,(\mathit{stmt})) = \mathbf{t})\\
&\quad\land\quad ((\mathrm{count}\,(\mathrm{cadddr}\,(\mathit{stmt})) < \mathrm{count}\,(\mathit{stmt})) = \mathbf{t})\\
&\quad\land\quad ((\mathrm{count}\,(\mathrm{caddddr}\,(\mathit{stmt})) < \mathrm{count}\,(\mathit{stmt})) = \mathbf{t}))
\end{aligned}$$

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;                         PREDEFINED PROCEDURES                      ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Notice that I define the semantics, etc. of each of the predefined routines
;; individually.  This allows me to loosen the restrictions on user-defined
;; procedures.  It also, allows me to dispense with much of the overhead of
;; user defined routines.  The only condition that any of the predefined's can
;; return is 'routineerror.  Consequently, I can do away with the cond translation
;; mechanism required of user-defined routines.  These simply set 'routineerror
;; themselves.   Also, I can eliminate the aliasing requirement for these since
;; I guarantee by coding the aliasing does not cause any problem.

;; This approach does seem to add an additional burden with respect to the
;; amount of things which must be proved.  However, since the predefined's
;; are not defined recursively, this is not particularly bad.

;; THE 'GENERIC' OPERATIONS
;; The operations of assignment and EQ work on any of the simple types.  For
;; each of them, we allow the variant where one of the args is a literal
;; of the appropriate type.  This is not strictly necessary except insofar
;; as it makes the subset more realistic and usable.

;; (mg-simple-variable-assignment x y)
;; x := y  where source must be a variable of the same type as the destination.
```

Definition:
ok-mg-simple-variable-assignment-args $(\mathit{args},\ \mathit{alist})$

$$\begin{aligned}
=\quad &(\text{length-plistp}\,(\mathit{args},\ 2)\\
&\quad\land\quad \text{simple-identifierp}\,(\mathrm{car}\,(\mathit{args}),\ \mathit{alist})\\
&\quad\land\quad \text{simple-identifierp}\,(\mathrm{cadr}\,(\mathit{args}),\ \mathit{alist})
\end{aligned}$$

```
     ∧  (cadr (assoc (car (args), alist))
           =  cadr (assoc (cadr (args), alist))))
```

```
;; (mg-simple-constant-assignment x c)
;; x := c where c must be a literal of the same type as the destination
```

DEFINITION:

```
ok-mg-simple-constant-assignment-args (args, alist)
=  (length-plistp (args, 2)
     ∧  simple-identifierp (car (args), alist)
     ∧  simple-typed-literalp (cadr (args), cadr (assoc (car (args), alist))))
```

```
;; (mg-simple-variable-eq b x y)
;; b := (x = y) where both x and y are variables of the same type.
```

DEFINITION:

```
ok-mg-simple-variable-eq-args (args, alist)
=  (length-plistp (args, 3)
     ∧  boolean-identifierp (car (args), alist)
     ∧  simple-identifierp (cadr (args), alist)
     ∧  simple-identifierp (caddr (args), alist)
     ∧  (cadr (assoc (cadr (args), alist))
           =  cadr (assoc (caddr (args), alist))))
```

```
;; (mg-simple-constant-eq b x c)
;; b := (x = c) where x is a variable and c a literal of compatible type.
```

DEFINITION:

```
ok-mg-simple-constant-eq-args (args, alist)
=  (length-plistp (args, 3)
     ∧  boolean-identifierp (car (args), alist)
     ∧  simple-identifierp (cadr (args), alist)
     ∧  simple-typed-literalp (caddr (args),
                               cadr (assoc (cadr (args), alist))))
```

```
;; THE INTEGER OPERATIONS

;; (mg-integer-le b x y)
;; b := (x le y) where both x and y are integer variables
```

DEFINITION:

ok-mg-integer-le-args (*args*, *alist*)
= (length-plistp (*args*, 3)
∧ boolean-identifierp (car (*args*), *alist*)
∧ int-identifierp (cadr (*args*), *alist*)
∧ int-identifierp (caddr (*args*), *alist*))

```
;; (mg-integer-unary-minus (args alist))
;; z := -x
```

Definition:
ok-mg-integer-unary-minus-args (*args*, *alist*)
= (length-plistp (*args*, 2)
∧ int-identifierp (car (*args*), *alist*)
∧ int-identifierp (cadr (*args*), *alist*))

```
;; (mg-integer-add (x y z))
;; x := y + z  >> Notice that this is a change from the previous version.
```

Definition:
ok-mg-integer-add-args (*args*, *alist*)
= (length-plistp (*args*, 3)
∧ int-identifierp (car (*args*), *alist*)
∧ int-identifierp (cadr (*args*), *alist*)
∧ int-identifierp (caddr (*args*), *alist*))

```
;; (mg-integer-subtract (x y z))
;; x := y - z
```

Definition:
ok-mg-integer-subtract-args (*args*, *alist*)
= (length-plistp (*args*, 3)
∧ int-identifierp (car (*args*), *alist*)
∧ int-identifierp (cadr (*args*), *alist*)
∧ int-identifierp (caddr (*args*), *alist*))

```
;; BOOLEAN OPERATIONS

;; (mg-boolean-or (b c d))
;; b := c or d   -- both disjuncts must be boolean identifiers
```

Definition:

```
ok-mg-boolean-or-args (args, alist)
=   (length-plistp (args, 3)
       ∧   boolean-identifierp (car (args), alist)
       ∧   boolean-identifierp (cadr (args), alist)
       ∧   boolean-identifierp (caddr (args), alist))
```

```
;; (mg-boolean-and (b c d))
;; b := c and d
```

DEFINITION:

```
ok-mg-boolean-and-args (args, alist)
=   (length-plistp (args, 3)
       ∧   boolean-identifierp (car (args), alist)
       ∧   boolean-identifierp (cadr (args), alist)
       ∧   boolean-identifierp (caddr (args), alist))
```

```
;; (mg-boolean-not (b c))
;; b := not c
```

DEFINITION:

```
ok-mg-boolean-not-args (args, alist)
=   (length-plistp (args, 2)
       ∧   boolean-identifierp (car (args), alist)
       ∧   boolean-identifierp (cadr (args), alist))
```

```
;; ARRAY OPERATIONS

;; (mg-index-array (z A i size))  >> Notice the change in the order of the args
;; z := A[i]
;; It is necessary for that last to be passed to do bounds-checking.  That information
;; is not available to the translator otherwise.  It is a special argument which is a
;; numberp rather than an MG literal.  It is expected that the preprocessor
;; will actually supply this argument, not
;; the mg programmer.  Thus the prefix form might have args (z A i 24) where 24 is the
;; size of A.
```

DEFINITION:

```
ok-mg-index-array-args (args, alist)
=   (length-plistp (args, 4)
       ∧   array-identifierp (cadr (args), alist)
       ∧   int-identifierp (caddr (args), alist)
       ∧   simple-typed-identifierp (car (args),
                                  array-elemtype (cadr (assoc (cadr (args),
                                                               alist))),
                                  alist)
       ∧  (cadddr (args) = array-length (cadr (assoc (cadr (args), alist))))
       ∧  (cadddr (args) < MAXINT))
```

```
;; (mg-array-element-assignment A i value size)
;; A[i] := value
;; Here i must be an integer variable and value a variable of
;; the appropriate element-type.
```

DEFINITION:

```
ok-mg-array-element-assignment-args (args, alist)
=   (length-plistp (args, 4)
     ∧  array-identifierp (car (args), alist)
     ∧  int-identifierp (cadr (args), alist)
     ∧  (cadddr (args) = array-length (cadr (assoc (car (args), alist))))
     ∧  (cadddr (args) < MAXINT)
     ∧  simple-typed-identifierp (caddr (args),
                                 array-elemtype (cadr (assoc (car (args),
                                                              alist))),
                                 alist))
```

DEFINITION:

```
ok-predefined-proc-args (name, args, alist)
=   case on name:
    case = mg-simple-variable-assignment
    then ok-mg-simple-variable-assignment-args (args, alist)
    case = mg-simple-constant-assignment
      then ok-mg-simple-constant-assignment-args (args, alist)
    case = mg-simple-variable-eq
      then ok-mg-simple-variable-eq-args (args, alist)
    case = mg-simple-constant-eq
      then ok-mg-simple-constant-eq-args (args, alist)
    case = mg-integer-le
      then ok-mg-integer-le-args (args, alist)
    case = mg-integer-unary-minus
      then ok-mg-integer-unary-minus-args (args, alist)
    case = mg-integer-add
      then ok-mg-integer-add-args (args, alist)
    case = mg-integer-subtract
      then ok-mg-integer-subtract-args (args, alist)
    case = mg-boolean-or
      then ok-mg-boolean-or-args (args, alist)
    case = mg-boolean-and
     then ok-mg-boolean-and-args (args, alist)
    case = mg-boolean-not
     then ok-mg-boolean-not-args (args, alist)
    case = mg-index-array
     then ok-mg-index-array-args (args, alist)
    case = mg-array-element-assignment
     then ok-mg-array-element-assignment-args (args, alist)
    otherwise f endcase
```

Event: Disable ok-predefined-proc-args.

```
;; A predefined proc-call is of the form (predefined-proc-call-mg name actuals)
;; where name is one of the legal predefineds and the actuals are legitimate arguments
;; for that predefined procedure according to the definitions above.
```

Definition:

```
ok-predefined-proc-call (stmt, alist)
=   (length-plistp (stmt, 3)
     ∧  predefined-procp (call-name (stmt))
     ∧  ok-predefined-proc-args (call-name (stmt),
                                 call-actuals (stmt),
                                 alist))
```

Event: Disable ok-predefined-proc-call.

Definition:

```
ok-proc-call (stmt, r-cond-list, alist, proc-list)
=   (length-plistp (stmt, 4)
     ∧  identifierp (call-name (stmt))
     ∧  user-defined-procp (call-name (stmt), proc-list)
     ∧  ok-actual-params-list (call-actuals (stmt), alist)
     ∧  no-duplicates (call-actuals (stmt))
     ∧  data-param-lists-match (call-actuals (stmt),
                                def-formals (fetch-called-def (stmt,
                                                               proc-list)),
                                alist)
     ∧  cond-identifier-plistp (call-conds (stmt), r-cond-list)
     ∧  cond-params-match (call-conds (stmt),
                           def-conds (fetch-called-def (stmt, proc-list))))
```

Event: Disable ok-proc-call.

DEFINITION:

```
ok-mg-statement (stmt, r-cond-list, alist, proc-list)
=   case on car (stmt):
    case = no-op-mg
    then cdr (stmt) = nil
    case = signal-mg
     then length-plistp (stmt, 2)
          ∧   ok-condition (signalled-condition (stmt), r-cond-list)
    case = prog2-mg
     then length-plistp (stmt, 3)
          ∧   ok-mg-statement (prog2-left-branch (stmt),
                               r-cond-list,
                               alist,
                               proc-list)
          ∧   ok-mg-statement (prog2-right-branch (stmt),
                               r-cond-list,
                               alist,
                               proc-list)
    case = loop-mg
     then length-plistp (stmt, 2)
          ∧   ok-mg-statement (loop-body (stmt),
                               cons ('leave, r-cond-list),
                               alist,
                               proc-list)
    case = if-mg
     then length-plistp (stmt, 4)
          ∧   boolean-identifierp (if-condition (stmt), alist)
          ∧   ok-mg-statement (if-true-branch (stmt),
                               r-cond-list,
                               alist,
                               proc-list)
          ∧   ok-mg-statement (if-false-branch (stmt),
                               r-cond-list,
                               alist,
                               proc-list)
    case = begin-mg
     then length-plistp (stmt, 4)
          ∧   ok-mg-statement (begin-body (stmt),
                               append (when-labels (stmt), r-cond-list),
                               alist,
                               proc-list)
          ∧   nonempty-cond-identifier-plistp (when-labels (stmt),
                                               r-cond-list)
          ∧   ok-mg-statement (when-handler (stmt),
                                        r-cond-list,
                                        alist,
                                        proc-list)
    case = proc-call-mg
     then ok-proc-call (stmt, r-cond-list, alist, proc-list)
    case = predefined-proc-call-mg
     then ok-predefined-proc-call (stmt, alist)
    otherwise f endcase
```

EVENT: Disable signalled-condition.

EVENT: Disable prog2-left-branch.

EVENT: Disable prog2-right-branch.

EVENT: Disable loop-body.

EVENT: Disable if-condition.

EVENT: Disable if-true-branch.

EVENT: Disable if-false-branch.

EVENT: Disable begin-body.

EVENT: Disable when-handler.

EVENT: Disable when-labels.

EVENT: Disable call-name.

EVENT: Disable call-actuals.

EVENT: Disable call-conds.

THEOREM: ok-signal-expansion
ok-mg-statement (cons (`'signal-mg`, *args*), *r-cond-list*, *alist*, *proc-list*)

```
=   (length-plistp (cons ('signal-mg, args), 2)
      ∧   ok-condition (signalled-condition (cons ('signal-mg, args)),
                        r-cond-list))
```

Theorem: ok-prog2-statement

```
((car (stmt) = 'prog2-mg)
 ∧   ok-mg-statement (stmt, r-cond-list, alist, proc-list))
→   (ok-mg-statement (prog2-left-branch (stmt), r-cond-list, alist, proc-list)
      ∧   ok-mg-statement (prog2-right-branch (stmt),
                           r-cond-list,
                           alist,
                           proc-list))
```

Theorem: ok-loop-statement

```
((car (stmt) = 'loop-mg)
 ∧   ok-mg-statement (stmt, r-cond-list, alist, proc-list))
→   ok-mg-statement (loop-body (stmt),
                     cons ('leave, r-cond-list),
                     alist,
                     proc-list)
```

Theorem: ok-if-statement

```
((car (stmt) = 'if-mg)
 ∧   ok-mg-statement (stmt, r-cond-list, alist, proc-list))
→   (ok-mg-statement (if-true-branch (stmt), r-cond-list, alist, proc-list)
      ∧   ok-mg-statement (if-false-branch (stmt),
                           r-cond-list,
                           alist,
                           proc-list))
```

Theorem: ok-begin-expansion

```
ok-mg-statement (cons ('begin-mg, args), r-cond-list, alist, proc-list)
=   (length-plistp (cons ('begin-mg, args), 4)
      ∧   ok-mg-statement (begin-body (cons ('begin-mg, args)),
                           append (when-labels (cons ('begin-mg, args)),
                                   r-cond-list),
                           alist,
                           proc-list)
      ∧   nonempty-cond-identifier-plistp (when-labels (cons ('begin-mg,
                                                              args)),
                                           r-cond-list)
      ∧   ok-mg-statement (when-handler (cons ('begin-mg, args)),
                           r-cond-list,
                           alist,
                           proc-list))
```

THEOREM: ok-begin-statement

```
((car (stmt) = 'begin-mg)
 ∧  ok-mg-statement (stmt, r-cond-list, alist, proc-list))
→   (ok-mg-statement (begin-body (stmt),
                      append (when-labels (stmt), r-cond-list),
                      alist,
                      proc-list)
      ∧  ok-mg-statement (when-handler (stmt),
                          r-cond-list,
                          alist,
                          proc-list))
```

THEOREM: ok-proc-call-expansion

```
ok-mg-statement (cons ('proc-call-mg, args), r-cond-list, alist, proc-list)
=  ok-proc-call (cons ('proc-call-mg, args), r-cond-list, alist, proc-list)
```

EVENT: Disable ok-mg-statement.

THEOREM: signalled-condition-not-normal

```
(('signal-mg = car (stmt))
 ∧  ok-mg-statement (stmt, r-cond-list, alist, proc-list))
→   (signalled-condition (stmt) ≠ 'normal)
```

EVENT: Disable signalled-condition-not-normal.

```
;; member of the data formal list is of the form
;;   (name typeref)
```

DEFINITION:

```
ok-mg-formal-data-param (exp)
=  (length-plistp (exp, 2)
     ∧  ok-mg-namep (car (exp))
     ∧  mg-type-refp (formal-type (exp)))
```

EVENT: Disable ok-mg-formal-data-param.

DEFINITION:

```
ok-mg-formal-data-params-plistp (lst)
=  if lst ≃ nil then lst = nil
   else ok-mg-formal-data-param (car (lst))
          ∧  ok-mg-formal-data-params-plistp (cdr (lst)) endif
```

```
;; A local decl is of the form (name typeref initial-value).
;; Notice that this obviates the need to compute initial values.
```

Definition:

```
ok-mg-local-data-decl (exp)
=   (length-plistp (exp, 3)
      ∧  ok-mg-namep (car (exp))
      ∧  mg-type-refp (formal-type (exp))
      ∧  ok-mg-valuep (formal-initial-value (exp), formal-type (exp)))
```

Definition:

```
ok-mg-local-data-plistp (lst)
=   if lst ≃ nil then lst = nil
    else ok-mg-local-data-decl (car (lst))
          ∧  ok-mg-local-data-plistp (cdr (lst)) endif
```

```
;; The legal conditions which can signalled are those which appear in the
;; the formal or in the local conds.
```

Definition:

```
make-cond-list (def) = append (def-conds (def), def-cond-locals (def))
```

```
;; This takes a list of formal data params or local var decls and
;; makes a name-alist.

;; >>> This is a useless definition.  It does nothing
```

Definition:

```
make-alist-from-formals (lst)
=   if lst ≃ nil then nil
    else cons (list (name (car (lst)), formal-type (car (lst))),
               make-alist-from-formals (cdr (lst))) endif
```

```
;; This takes the formals and locals and makes a name-alist.

;; >>> Why not just concatenate them.
```

Definition:

```
make-name-alist (def)
=   append (make-alist-from-formals (def-formals (def)),
            make-alist-from-formals (def-locals (def)))
```

```
;; Given a list of formal-data-params or local-data-decls this lists
;; off the names.  This is necessary to check that all local names are
;; unique.
```

Definition:

```
collect-local-names (def)
=   append (listcars (def-formals (def)), listcars (def-locals (def)))
```

Definition:

```
ok-mg-def (def, proc-list)
=   (length-plistp (def, 6)
       ∧   ok-mg-namep (def-name (def))
       ∧   ok-mg-formal-data-params-plistp (def-formals (def))
       ∧   identifier-plistp (def-conds (def))
       ∧   ok-mg-local-data-plistp (def-locals (def))
       ∧   identifier-plistp (def-cond-locals (def))
       ∧   no-duplicates (collect-local-names (def))
       ∧   ((length (def-conds (def)) + length (def-cond-locals (def)))
             <   (((exp (2, MG-WORD-SIZE) − 1) − 1) − 1))
       ∧   ok-mg-statement (def-body (def),
                             make-cond-list (def),
                             make-name-alist (def),
                             proc-list))
```

Theorem: make-cond-list-legal-length

```
 ok-mg-def (def, proc-list)
 →   ((length (make-cond-list (def))
          <   (((exp (2, MG-WORD-SIZE) − 1) − 1) − 1))
        =   t)
```

```
;; (name formals locals conds local-conds body)
```

Definition:

```
ok-mg-def-plistp1 (lst1, lst2)
=   if lst1 ≃ nil then lst1 = nil
    else ok-mg-def (car (lst1), lst2)
           ∧   ok-mg-def-plistp1 (cdr (lst1), lst2) endif
```

Definition:

```
ok-mg-def-plistp (proc-list) = ok-mg-def-plistp1 (proc-list, proc-list)
```

Event: Disable ok-mg-def.

Theorem: assoc-def-ok1
(ok-mg-def-plistp1 (*proc-list1*, *proc-list2*) ∧ definedp (*name*, *proc-list1*))
→ ok-mg-def (assoc (*name*, *proc-list1*), *proc-list2*)

Event: Disable assoc-def-ok1.

Theorem: called-def-ok
((`'proc-call-mg` = car (*stmt*))
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ ok-mg-def-plistp (*proc-list*))
→ ok-mg-def (fetch-called-def (*stmt*, *proc-list*), *proc-list*)

Event: Disable fetch-called-def.

Theorem: called-def-formals-ok
((`'proc-call-mg` = car (*stmt*))
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ ok-mg-def-plistp (*proc-list*))
→ (ok-mg-formal-data-params-plistp (def-formals (fetch-called-def (*stmt*,
proc-list)))
∧ ok-mg-local-data-plistp (def-locals (fetch-called-def (*stmt*,
proc-list)))
∧ data-param-lists-match (call-actuals (*stmt*),
def-formals (fetch-called-def (*stmt*,
proc-list)),
name-alist)
∧ no-duplicates (listcars (def-formals (fetch-called-def (*stmt*,
proc-list))))
∧ plistp (make-cond-list (fetch-called-def (*stmt*, *proc-list*)))
∧ no-duplicates (append (listcars (def-formals (fetch-called-def (*stmt*,
proc-list))),
listcars (def-locals (fetch-called-def (*stmt*,
proc-list)))))
∧ all-cars-unique (def-formals (fetch-called-def (*stmt*, *proc-list*)))
∧ all-cars-unique (def-locals (fetch-called-def (*stmt*, *proc-list*))))

Theorem: ok-locals-plistp
ok-mg-local-data-plistp (x) → plistp (x)

Event: Disable ok-locals-plistp.

Theorem: locals-plistp
((`'proc-call-mg` = car (*stmt*))

$\land$ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
$\land$ ok-mg-def-plistp (*proc-list*))
$\rightarrow$ plistp (def-locals (fetch-called-def (*stmt*, *proc-list*)))

EVENT: Disable mg-name-alist-elementp.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;                      MG INTERPRETER FUNCTIONS                      ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The following set of functions defines the interpreter for Micro-Gypsy.
;; A state is defined to be an ordered pair consisting of a variable a-list and a
;; global condition indicator.
;; The meaning of a statement in an environment (state) is the environment
;; which results from executing that statement.  Thus, the semantics is very
;; much an operational semantics.

;; The condition component of the state is simply a litatom which is 'normal
;; in the initial state.  The variable a-list is of the form
;; ((v1 (value1 type1)) ... (vn (valuen typen))) for each of the entities known in
;; the current scope.  Notice that the recognizer alist is not required though I
;; will need to record on the var-alist whether a variable is a const or var
;; param when I add procedure calls.
```

EVENT: Add the shell *mg-state*, with recognizer function symbol *mg-statep* and 3 accessors: *cc*, with type restriction (none-of) and default value false; *mg-alist*, with type restriction (none-of) and default value false; *mg-psw*, with type restriction (none-of) and default value false.

DEFINITION:
resource-errorp (*mg-state*) = (mg-psw (*mg-state*) $\neq$ `'run`)

DEFINITION:
signal-system-error (*mg-state*, *error*)
= mg-state (cc (*mg-state*), mg-alist (*mg-state*), *error*)

DEFINITION: normal (*mg-state*) = (cc (*mg-state*) = `'normal`)

DEFINITION: m-value (x) = caddr (x)

DEFINITION:
get-m-value (*name*, *alist*) = m-value (assoc (*name*, *alist*))

Definition:

$$\begin{aligned}
&\text{mg-alist-elementp}\,(x)\\
= \quad &(\text{length-plistp}\,(x,\ \mathtt{3})\\
&\quad \land \quad \text{ok-mg-namep}\,(\text{car}\,(x))\\
&\quad \land \quad \text{mg-type-refp}\,(\text{m-type}\,(x))\\
&\quad \land \quad \text{ok-mg-valuep}\,(\text{m-value}\,(x),\ \text{m-type}\,(x)))
\end{aligned}$$

Theorem: new-value-mg-alist-elementp

$$\begin{aligned}
&(\text{mg-alist-elementp}\,(x) \land \text{ok-mg-valuep}\,(\mathit{value},\ \text{cadr}\,(x)))\\
\rightarrow \quad &\text{mg-alist-elementp}\,(\text{cons}\,(\text{car}\,(x),\ \text{cons}\,(\text{cadr}\,(x),\ \text{cons}\,(\mathit{value},\ \text{cdddr}\,(x)))))
\end{aligned}$$

Event: Disable mg-alist-elementp.

Definition:

$$\begin{aligned}
&\text{mg-alistp}\,(\mathit{lst})\\
= \quad &\textbf{if } \mathit{lst} \simeq \textbf{nil} \ \textbf{ then } \mathit{lst} = \textbf{nil}\\
&\textbf{else } \text{mg-alist-elementp}\,(\text{car}\,(\mathit{lst})) \land \text{mg-alistp}\,(\text{cdr}\,(\mathit{lst}))\ \textbf{endif}
\end{aligned}$$

Theorem: mg-alistp-cdr

$(\text{listp}\,(x) \land \text{mg-alistp}\,(x)) \rightarrow \text{mg-alistp}\,(\text{cdr}\,(x))$

Theorem: mg-alistp-cons

$\text{mg-alistp}\,(\text{cons}\,(x,\ \text{cons}\,(y,\ z))) \rightarrow \text{mg-alistp}\,(\text{cons}\,(x,\ z))$

Theorem: mg-alist-member-mg-alist-elementp

$(\text{mg-alistp}\,(\mathit{mg\text{-}alist}) \land (x \in \mathit{mg\text{-}alist})) \rightarrow \text{mg-alist-elementp}\,(x)$

Theorem: mg-alistp-distributes

$\text{mg-alistp}\,(\text{append}\,(x,\ y)) \rightarrow \text{mg-alistp}\,(y)$

Theorem: mg-alistp-distributes2

$(\text{mg-alistp}\,(\text{append}\,(x,\ y)) \land \text{plistp}\,(x)) \rightarrow \text{mg-alistp}\,(x)$

Event: Disable mg-alistp-distributes2.

Theorem: mg-alist-mg-name-alistp

$\text{mg-alistp}\,(\mathit{lst}) \rightarrow \text{mg-name-alistp}\,(\mathit{lst})$

Theorem: mg-alistp-plistp

$\text{mg-alistp}\,(\mathit{alist}) \rightarrow \text{plistp}\,(\mathit{alist})$

Event: Disable mg-alistp-plistp.

Theorem: mg-alist-elements-have-ok-values

$$\begin{aligned}
&(\text{mg-alistp}\,(\mathit{alist}) \land \text{definedp}\,(x,\ \mathit{alist}))\\
\rightarrow \quad &\text{ok-mg-valuep}\,(\text{caddr}\,(\text{assoc}\,(x,\ \mathit{alist})),\ \text{cadr}\,(\text{assoc}\,(x,\ \mathit{alist})))
\end{aligned}$$

THEOREM: restrict-preserves-mg-alistp
mg-alistp (*alist*) → mg-alistp (restrict (*alist*, *names*))

DEFINITION:
ok-cc (*c*, *cond-list*)
= (litatom (*c*)
 ∧ ((*c* ∈ `'(normal routineerror)`) ∨ (*c* ∈ *cond-list*)))

THEOREM: mg-alistp-implies-mg-statep
mg-alistp (mg-alist (*mg-state*)) → mg-statep (*mg-state*)

DEFINITION:
ok-mg-statep (*mg-state*, *cond-list*)
= (ok-cc (cc (*mg-state*), *cond-list*) ∧ mg-alistp (mg-alist (*mg-state*)))

THEOREM: ok-mg-statep-alist-plistp
ok-mg-statep (*mg-state*, *cond-list*) → plistp (mg-alist (*mg-state*))

THEOREM: cons-preserves-ok-mg-statep
ok-mg-statep (*mg-state*, *cond-list*)
→ ok-mg-statep (*mg-state*, cons (*x*, *cond-list*))

EVENT: Disable cons-preserves-ok-mg-statep.

DEFINITION:
set-condition (*mg-state*, *condition-name*)
= mg-state (*condition-name*, mg-alist (*mg-state*), mg-psw (*mg-state*))

THEOREM: cc-set-condition
cc (set-condition (*mg-state*, *cond*)) = *cond*

THEOREM: mg-alist-set-condition
mg-alist (set-condition (*mg-state*, *cond*)) = mg-alist (*mg-state*)

THEOREM: ok-mg-statep-mg-alist-mg-alistp
ok-mg-statep (*mg-state*, *r-cond-list*) → mg-alistp (mg-alist (*mg-state*))

DEFINITION:
remove-leave (*mg-state*)
= **if** cc (*mg-state*) = `'leave` **then** set-condition (*mg-state*, `'normal`)
else *mg-state* **endif**

DEFINITION:
mg-expression-falsep (*exp*, *mg-state*)
= (get-m-value (*exp*, mg-alist (*mg-state*))
 = `'(boolean-mg false-mg)`)

```
;; This assumes that the formals and actuals are both given as lists of
;; bare cond names.  This will have to change when the identifier lists
;; are reinstated.
;; Given condition lists (formal1 formal2 ... formaln) and
;; (actual1 actual2 ... actualn), if cond is formali then return actuali,
;; else 'routineerror.
```

DEFINITION:

```
convert-condition1 (cond, formals, actuals)
=   if formals $\simeq$ nil then 'routineerror
    elseif cond = car (formals) then car (actuals)
    else convert-condition1 (cond, cdr (formals), cdr (actuals)) endif
```

DEFINITION:

```
convert-condition (cond, formals, actuals)
=   if cond $\in$ '(normal routineerror) then cond
    else convert-condition1 (cond, formals, actuals) endif
```

THEOREM: convert-condition-non-member
(*cond* $\notin$ *formals*)
$\rightarrow$ (convert-condition1 (*cond*, *formals*, *actuals*) = `'routineerror`)

DEFINITION:

```
set-alist-value (name, val, alist)
=   if alist $\simeq$ nil then nil
    elseif car (car (alist)) = name
    then cons (cons (name,
                     cons (m-type (car (alist)), cons (val, cdddr (car (alist))))),
               cdr (alist))
    else cons (car (alist), set-alist-value (name, val, cdr (alist))) endif
```

THEOREM: set-alist-value-preserves-definedp
definedp (*v*, *alist*) $\rightarrow$ definedp (*v*, set-alist-value (*x*, *y*, *alist*))

EVENT: Disable set-alist-value-preserves-definedp.

THEOREM: set-alist-value-preserves-ok-actual-params-list
ok-actual-params-list (*actuals*, *alist*)
$\rightarrow$ ok-actual-params-list (*actuals*, set-alist-value (*x*, *y*, *alist*))

EVENT: Disable set-alist-value-preserves-ok-actual-params-list.

THEOREM: set-alist-value-preserves-cadr-assoc
mg-alistp (*alist*)
$\rightarrow$ (cadr (assoc (*v*, set-alist-value (*x*, *y*, *alist*))) = cadr (assoc (*v*, *alist*)))

Event: Disable set-alist-value-preserves-cadr-assoc.

Theorem: set-alist-value-preserves-data-param-lists-match
(mg-alistp (*alist*) ∧ data-param-lists-match (*actuals*, *formals*, *alist*))
→ data-param-lists-match (*actuals*, *formals*, set-alist-value (*x*, *y*, *alist*))

Event: Disable set-alist-value-preserves-data-param-lists-match.

Theorem: set-alist-value-preserves-listcars
listcars (set-alist-value (*x*, *y*, *alist*)) = listcars (*alist*)

Theorem: set-alist-value-preserves-all-cars-unique
all-cars-unique (*alist*) → all-cars-unique (set-alist-value (*x*, *y*, *alist*))

Event: Disable set-alist-value-preserves-all-cars-unique.

Theorem: set-alist-value-preserves-signatures-match
plistp (*alist*) → signatures-match (*alist*, set-alist-value (*x*, *y*, *alist*))

Event: Disable set-alist-value-preserves-signatures-match.

Definition:

```
copy-out-params (formals, actuals, new-var-alist, old-var-alist)
=   if formals ≃ nil then old-var-alist
    else copy-out-params (cdr (formals),
                          cdr (actuals),
                          new-var-alist,
                          set-alist-value (car (actuals),
                                           caddr (assoc (caar (formals),
                                                         new-var-alist)),
                                           old-var-alist)) endif
```

Definition:

```
map-call-effects (new-state, def, stmt, old-state)
=   mg-state (convert-condition (cc (new-state),
                                 def-conds (def),
                                 call-conds (stmt)),
              copy-out-params (def-formals (def),
                               call-actuals (stmt),
                               mg-alist (new-state),
                               mg-alist (old-state)),
              mg-psw (new-state))
```

Event: Disable map-call-effects.

Theorem: map-call-effects-preserves-normal

```
normal (new-state)
→   (cc (map-call-effects (new-state, def, stmt, old-state)) = 'normal)
```

Theorem: map-call-effects-preserves-routineerror

```
(cc (new-state) = 'routineerror)
→   (cc (map-call-effects (new-state, def, stmt, old-state))
      =  'routineerror)
```

```
;; This creates the part of the var alist for the call corresponding to the
;; formals.  The formal has form (name kind type) and the actual is either
;; a literal or an identifierp.
```

Definition:

```
make-call-param-alist (formals, actuals, mg-alist)
=   if formals ≃ nil then nil
    else cons (list (caar (formals),
                     cadar (formals),
                     caddr (assoc (car (actuals), mg-alist))),
               make-call-param-alist (cdr (formals),
                                      cdr (actuals),
                                      mg-alist)) endif
```

Theorem: make-call-param-alist-plistp

```
plistp (make-call-param-alist (formals, actuals, alist))
```

Theorem: make-call-param-alist-preserves-listcars

```
listcars (make-call-param-alist (formals, actuals, mg-alist))
=   listcars (formals)
```

Definition:

```
make-call-var-alist (mg-alist, stmt, def)
=   append (make-call-param-alist (def-formals (def),
                                   call-actuals (stmt),
                                   mg-alist),
            def-locals (def))
```

```
;; This doesn't really need the hypothesis.
```

Theorem: plistp-make-call-var-alist

```
plistp (def-locals (def)) → plistp (make-call-var-alist (state, stmt, def))
```

Definition:
make-call-environment (*mg-state*, *stmt*, *def*)
= mg-state ('`normal`,
make-call-var-alist (mg-alist (*mg-state*), *stmt*, *def*),
mg-psw (*mg-state*))

Event: Disable make-call-environment.

Theorem: make-call-environment-decomposition
(cc (make-call-environment (*mg-state*, *stmt*, *def*)) = '`normal`)
∧ (mg-alist (make-call-environment (*mg-state*, *stmt*, *def*))
= make-call-var-alist (mg-alist (*mg-state*), *stmt*, *def*))
∧ (mg-psw (make-call-environment (*mg-state*, *stmt*, *def*))
= mg-psw (*mg-state*))

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;             SEMANTICS FOR THE PREDEFINED PROCEDURES                ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; These are structurally identical to the Piton version merely
;; altered to change 't and 'f to 'true-mg and 'false-mg.
```

Definition:
mg-bool (*x*)
= tag ('`boolean-mg`,
if *x* **then** '`true-mg`
else '`false-mg` **endif**)

Definition:
mg-or-bool (*x*, *y*)
= **if** *x* = '`false-mg` **then** *y*
else '`true-mg` **endif**

Definition:
mg-and-bool (*x*, *y*)
= **if** *x* = '`false-mg` **then** '`false-mg`
else *y* **endif**

Definition:
mg-not-bool (*x*)
= **if** *x* = '`false-mg` **then** '`true-mg`
else '`false-mg` **endif**

DEFINITION:
fetch-array-element (*a*, *i*, *alist*) = get (*i*, caddr (assoc (*a*, *alist*)))

```
;; This returns the array with the substitution, not the resulting
;; alist.
```

DEFINITION:
put-array-element (*a*, *i*, *val*, *alist*) = put (*val*, *i*, caddr (assoc (*a*, *alist*)))

```
;; x := y  -- y is a variable
```

DEFINITION:

```
mg-meaning-mg-simple-variable-assignment (stmt, mg-state)
=   mg-state ('normal,
              set-alist-value (car (call-actuals (stmt)),
                               get-m-value (cadr (call-actuals (stmt)),
                                            mg-alist (mg-state)),
                               mg-alist (mg-state)),
              mg-psw (mg-state))
```

```
;; x := c  -- c is a constant
```

DEFINITION:

```
mg-meaning-mg-simple-constant-assignment (stmt, mg-state)
=   mg-state ('normal,
              set-alist-value (car (call-actuals (stmt)),
                               cadr (call-actuals (stmt)),
                               mg-alist (mg-state)),
              mg-psw (mg-state))
```

```
;; b := (x = y)
```

DEFINITION:

```
mg-meaning-mg-simple-variable-eq (stmt, mg-state)
=   mg-state ('normal,
              set-alist-value (car (call-actuals (stmt)),
                               mg-bool (untag (get-m-value (cadr (call-actuals (stmt)),
                                                            mg-alist (mg-state)))
                                        =   untag (get-m-value (caddr (call-actuals (stmt)),
                                                                mg-alist (mg-state)))),
                               mg-alist (mg-state)),
              mg-psw (mg-state))
```

```
;; b := (x = c)
```

DEFINITION:

```
mg-meaning-mg-simple-constant-eq (stmt, mg-state)
=   mg-state ('normal,
              set-alist-value (car (call-actuals (stmt)),
                               mg-bool (untag (get-m-value (cadr (call-actuals (stmt)),
                                                            mg-alist (mg-state)))
                                        =   untag (caddr (call-actuals (stmt)))),
                               mg-alist (mg-state)),
              mg-psw (mg-state))
```

```
;; b := (x le y)  -- Here x and y are integer variables
```

DEFINITION:

```
mg-meaning-mg-integer-le (stmt, mg-state)
=   mg-state ('normal,
              set-alist-value (car (call-actuals (stmt)),
                               mg-bool (ileq (untag (get-m-value (cadr (call-actuals (stmt)),
                                                                  mg-alist (mg-state))),
                                              untag (get-m-value (caddr (call-actuals (stmt)),
                                                                  mg-alist (mg-state))))),
                               mg-alist (mg-state)),
              mg-psw (mg-state))
```

```
;; x := -y
```

DEFINITION:

```
mg-meaning-mg-integer-unary-minus (stmt, mg-state)
=   let value  be  inegate (untag (get-m-value (cadr (call-actuals (stmt)),
                                                mg-alist (mg-state))))
    in
    if small-integerp (value, MG-WORD-SIZE)
    then mg-state ('normal,
                   set-alist-value (car (call-actuals (stmt)),
                                    tag ('int-mg, value),
                                    mg-alist (mg-state)),
                   mg-psw (mg-state))
    else set-condition (mg-state, 'routineerror) endif endlet
```

```
;; x := y + z
```

Definition:

```
mg-meaning-mg-integer-add (stmt, mg-state)
=   let sum  be  iplus (untag (get-m-value (cadr (call-actuals (stmt)),
                                            mg-alist (mg-state))),
                        untag (get-m-value (caddr (call-actuals (stmt)),
                                            mg-alist (mg-state))))
    in
    if small-integerp (sum, MG-WORD-SIZE)
    then mg-state ('normal,
                   set-alist-value (car (call-actuals (stmt)),
                                    tag ('int-mg, sum),
                                    mg-alist (mg-state)),
                   mg-psw (mg-state))
    else set-condition (mg-state, 'routineerror) endif endlet
```

```
;; x := y - z
```

Definition:

```
mg-meaning-mg-integer-subtract (stmt, mg-state)
=   let diff  be  idifference (untag (get-m-value (cadr (call-actuals (stmt)),
                                                   mg-alist (mg-state))),
                               untag (get-m-value (caddr (call-actuals (stmt)),
                                                   mg-alist (mg-state))))
    in
    if small-integerp (diff, MG-WORD-SIZE)
    then mg-state ('normal,
                   set-alist-value (car (call-actuals (stmt)),
                                    tag ('int-mg, diff),
                                    mg-alist (mg-state)),
                   mg-psw (mg-state))
    else set-condition (mg-state, 'routineerror) endif endlet
```

```
;; b := b1 or b2
```

Definition:

```
mg-meaning-mg-boolean-or (stmt, mg-state)
=   mg-state ('normal,
              set-alist-value (car (call-actuals (stmt)),
                               tag ('boolean-mg,
                                    mg-or-bool (untag (get-m-value (cadr (call-actuals (stmt)),
                                                                    mg-alist (mg-state))),
                                                untag (get-m-value (caddr (call-actuals (stmt)),
                                                                    mg-alist (mg-state))))),
                        mg-alist (mg-state)),
            mg-psw (mg-state))
```

```
;; b := b1 and b2
```

Definition:

```
mg-meaning-mg-boolean-and (stmt, mg-state)
=   mg-state ('normal,
            set-alist-value (car (call-actuals (stmt)),
                        tag ('boolean-mg,
                            mg-and-bool (untag (get-m-value (cadr (call-actuals (stmt)),
                                                            mg-alist (mg-state))),
                                        untag (get-m-value (caddr (call-actuals (stmt)),
                                                            mg-alist (mg-state))))),
                        mg-alist (mg-state)),
            mg-psw (mg-state))
```

```
;; b := not b1
```

Definition:

```
mg-meaning-mg-boolean-not (stmt, mg-state)
=   mg-state ('normal,
            set-alist-value (car (call-actuals (stmt)),
                        tag ('boolean-mg,
                            mg-not-bool (untag (get-m-value (cadr (call-actuals (stmt)),
                                                            mg-alist (mg-state))))),
                        mg-alist (mg-state)),
            mg-psw (mg-state))
```

```
;; THE ARRAY OPERATIONS

;; z := A[i] for A of size
;; The call is (predefined-proc-call-mg mg-index-array (z A i size))
```

Definition:

```
mg-meaning-mg-index-array (stmt, mg-state)
=   let index be untag (get-m-value (caddr (call-actuals (stmt)),
                                    mg-alist (mg-state)))
    in
    if (index ∈ N)
      ∧  (index < array-length (get-m-type (cadr (call-actuals (stmt)),
                                           mg-alist (mg-state))))
    then mg-state ('normal,
                   set-alist-value (car (call-actuals (stmt)),
                                    fetch-array-element (cadr (call-actuals (stmt)),
                                                         index,
                                                         mg-alist (mg-state)),
                                    mg-alist (mg-state)),
                   mg-psw (mg-state))
    else set-condition (mg-state, 'routineerror) endif endlet
```

```
;; A[i] := v   -- The actual argument list is (A i v size) where size is the
;;      array-length of the type of A.  Here i and v are both variables.
```

DEFINITION:

```
mg-meaning-mg-array-element-assignment (stmt, mg-state)
=   let index  be  untag (get-m-value (cadr (call-actuals (stmt)),
                                       mg-alist (mg-state))),
        val  be  get-m-value (caddr (call-actuals (stmt)),
                              mg-alist (mg-state))
    in
    if ($index \in \mathbf{N}$)
       $\wedge$  ($index <$ array-length (get-m-type (car (call-actuals (stmt)),
                                                  mg-alist (mg-state))))
    then mg-state ('normal,
                   set-alist-value (car (call-actuals (stmt)),
                                    put-array-element (car (call-actuals (stmt)),
                                                       index,
                                                       val,
                                                       mg-alist (mg-state)),
                                    mg-alist (mg-state)),
                   mg-psw (mg-state))
    else set-condition (mg-state, 'routineerror) endif endlet
```

DEFINITION:

```
mg-meaning-predefined-proc-call (stmt, mg-state)
=   case on call-name (stmt):
    case = mg-simple-variable-assignment
    then mg-meaning-mg-simple-variable-assignment (stmt, mg-state)
    case = mg-simple-constant-assignment
     then mg-meaning-mg-simple-constant-assignment (stmt, mg-state)
    case = mg-simple-variable-eq
     then mg-meaning-mg-simple-variable-eq (stmt, mg-state)
    case = mg-simple-constant-eq
     then mg-meaning-mg-simple-constant-eq (stmt, mg-state)
    case = mg-integer-le
```

```
     then mg-meaning-mg-integer-le (stmt, mg-state)
    case = mg-integer-unary-minus
     then mg-meaning-mg-integer-unary-minus (stmt, mg-state)
    case = mg-integer-add
     then mg-meaning-mg-integer-add (stmt, mg-state)
    case = mg-integer-subtract
     then mg-meaning-mg-integer-subtract (stmt, mg-state)
    case = mg-boolean-or
     then mg-meaning-mg-boolean-or (stmt, mg-state)
    case = mg-boolean-and
     then mg-meaning-mg-boolean-and (stmt, mg-state)
    case = mg-boolean-not
     then mg-meaning-mg-boolean-not (stmt, mg-state)
    case = mg-index-array
     then mg-meaning-mg-index-array (stmt, mg-state)
    case = mg-array-element-assignment
     then mg-meaning-mg-array-element-assignment (stmt, mg-state)
    otherwise mg-state endcase
```

Event: Disable mg-meaning-predefined-proc-call.

Definition:

```
mg-meaning (stmt, proc-list, mg-state, n)
=   if $n \simeq 0$ then signal-system-error (mg-state, 'timed-out)
    elseif ¬ normal (mg-state) then mg-state
    else case on car (stmt):
         case = no-op-mg
         then mg-state
         case = signal-mg
          then set-condition (mg-state, signalled-condition (stmt))
         case = prog2-mg
          then mg-meaning (prog2-right-branch (stmt),
                           proc-list,
                           mg-meaning (prog2-left-branch (stmt),
                                       proc-list,
                                       mg-state,
                                       $n - 1$),
                           $n - 1$)
         case = loop-mg
          then remove-leave (mg-meaning (stmt,
                                         proc-list,
                                         mg-meaning (loop-body (stmt),
                                                     proc-list,
                                                     mg-state,
                                                $n - 1$),
                                   $n - 1$))
case = if-mg
  then if mg-expression-falsep (if-condition (stmt), mg-state)
        then mg-meaning (if-false-branch (stmt),
                         proc-list,
                         mg-state,
                         $n - 1$)
        else mg-meaning (if-true-branch (stmt),
                         proc-list,
                         mg-state,
                         $n - 1$) endif
case = begin-mg
  then if cc (mg-meaning (begin-body (stmt),
                          proc-list,
                          mg-state,
                          $n - 1$))
           ∈  when-labels (stmt)
        then mg-meaning (when-handler (stmt),
                         proc-list,
                         set-condition (mg-meaning (begin-body (stmt),
                                                    proc-list,
                                                    mg-state,
                                                    $n - 1$),
                                        'normal),
                         $n - 1$)
        else mg-meaning (begin-body (stmt),
                         proc-list,
                         mg-state,
                         $n - 1$) endif
case = proc-call-mg
  then map-call-effects (mg-meaning (def-body (fetch-called-def (stmt,
                                                                 proc-list)),
                                     proc-list,
                                     make-call-environment (mg-state,
                                                            stmt,
                                                            fetch-called-def (stmt,
                                                                              proc-list)),
                                     $n - 1$),
                         fetch-called-def (stmt, proc-list),
                         stmt,
                         mg-state)
case = predefined-proc-call-mg
  then mg-meaning-predefined-proc-call (stmt, mg-state)
```

```
        otherwise mg-state endcase endif
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;                         RESOURCE ERRORS                            ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; This is the version of mg-meaning with resource errors.  It should be a
;; theorem that in the absence of resource-error, it behaves exactly as
;; mg-meaning.

;; The resource descriptor is a pair <temp-stk-size, ctrl-stk-size> where
;; the two components are numberps which characterize the number of free
;; slots on the top of the stacks.  We cause a resource error if there is
;; not enough space to continue without stack overflow.  If the resource
;; requirements exceed the available space, the cc is set to 'resource-error.

;; This computes the amount of space required for the storage of the locals.  It is the
;; number of simple variables plus the sum of the lengths of the arrays.
```

Definition:

```
data-length (locals)
=   if locals ≃ nil then 0
    elseif simple-mg-type-refp (cadr (car (locals)))
    then 1 + data-length (cdr (locals))
    else array-length (cadr (car (locals)))
         +   data-length (cdr (locals)) endif
```

Theorem: data-length-not-zerop
(ok-mg-local-data-plistp (*locals*) $\land$ listp (*locals*))
$\rightarrow$ (data-length (*locals*) $\not\simeq$ 0)

Event: Disable data-length-not-zerop.

Theorem: data-length-not-zerop2
(ok-mg-local-data-plistp (*locals*) $\land$ listp (*locals*))
$\rightarrow$ ((data-length (*locals*) $\in \mathbf{N}$) $\land$ (data-length (*locals*) $\neq$ 0))

Definition:

```
predefined-proc-call-temp-stk-requirement (name)
=   case on name:
    case = mg-simple-variable-assignment
    then 2
    case = mg-simple-constant-assignment
     then 2
    case = mg-simple-variable-eq
     then 3
    case = mg-simple-constant-eq
     then 3
    case = mg-integer-le
     then 3
    case = mg-integer-unary-minus
     then 2
    case = mg-integer-add
     then 3
    case = mg-integer-subtract
     then 3
    case = mg-boolean-or
     then 3
    case = mg-boolean-and
     then 3
    case = mg-boolean-not
     then 2
    case = mg-index-array
     then 4
    case = mg-array-element-assignment
     then 4
    otherwise 0 endcase
```

```
;; The number associated with each predefined procedure
;; represents the number of formals plus locals in the
;; Piton implementation.  This is required because the
;; ctrl-stk requirements for the call p-frame is
;; (plus 2
;;       (length (formal-vars def))
;;       (length (temp-var-dcls def)))
```

Definition:

predefined-proc-call-bindings-count (*name*)

```
=   case on name:
    case = mg-simple-variable-assignment
    then 2
    case = mg-simple-constant-assignment
     then 2
    case = mg-simple-variable-eq
    then 3
  case = mg-simple-constant-eq
    then 3
  case = mg-integer-le
    then 3
  case = mg-integer-unary-minus
    then 4
  case = mg-integer-add
    then 4
  case = mg-integer-subtract
    then 4
  case = mg-boolean-or
    then 3
  case = mg-boolean-and
    then 3
  case = mg-boolean-not
    then 3
  case = mg-index-array
    then 5
  case = mg-array-element-assignment
    then 5
  otherwise 0 endcase
```

DEFINITION:
predefined-proc-call-p-frame-size (*name*)
= (1 + (1 + predefined-proc-call-bindings-count (*name*)))

```
;; I'm implemententing the resources-available as a pair
;; <t-size c-size> of numberps.
```

DEFINITION: t-size (x) = car (x)

DEFINITION: c-size (x) = cadr (x)

```
;; An interesting fact is that the requirements for execution of a statement are not
;; dependent on the state.  (Except in the case of begin where a when-label is
;; signalled.)  This makes the computation of the resource requirements independent of
;; mg-meaning and may allow a much cleaner treatment.
```

DEFINITION:
temp-stk-requirements (*stmt*, *proc-list*)

```
= case on car (stmt):
  case = no-op-mg
```

```
    then 0
    case = signal-mg
     then 1
    case = prog2-mg
     then 0
    case = loop-mg
     then 1
    case = if-mg
     then 1
    case = begin-mg
     then 1
    case = proc-call-mg
     then max (data-length (def-locals (fetch-called-def (stmt, proc-list)))
                  +    length (def-locals (fetch-called-def (stmt, proc-list)))
                  +    length (call-actuals (stmt)),
                  1)
    case = predefined-proc-call-mg
     then predefined-proc-call-temp-stk-requirement (call-name (stmt))
    otherwise 0 endcase
```

Definition:

```
ctrl-stk-requirements (stmt, proc-list)
=   case on car (stmt):
    case = no-op-mg
    then 0
    case = signal-mg
     then 0
    case = prog2-mg
     then 0
    case = loop-mg
     then 0
    case = if-mg
     then 0
    case = begin-mg
     then 0
    case = proc-call-mg
     then 2
             +    length (def-locals (fetch-called-def (stmt, proc-list)))
             +    length (def-formals (fetch-called-def (stmt, proc-list)))
    case = predefined-proc-call-mg
     then predefined-proc-call-p-frame-size (call-name (stmt))
    otherwise 0 endcase
```

Event: Disable temp-stk-requirements.

EVENT: Disable ctrl-stk-requirements.

```
;; Resources are inadequate if I can't perform the current operation without
;; running out of space.  This can alternatively be phrased as follows.
;; In this version, the size-pair contains the
;; <current temp-stk length, current ctrl-stk length>
```

DEFINITION:

```
resources-inadequatep (stmt, proc-list, size-pair)
=   ((temp-stk-requirements (stmt, proc-list)
       $\not<$  (MG-MAX-TEMP-STK-SIZE $-$ t-size (size-pair)))
      $\vee$  (ctrl-stk-requirements (stmt, proc-list)
           $\not<$  (MG-MAX-CTRL-STK-SIZE $-$ c-size (size-pair))))
```

EVENT: Disable resources-inadequatep.

DEFINITION:

```
mg-meaning-r (stmt, proc-list, mg-state, n, sizes)
=   if $n \simeq 0$ then signal-system-error (mg-state, 'timed-out)
    elseif $\neg$ normal (mg-state) then mg-state
    elseif resources-inadequatep (stmt, proc-list, sizes)
    then signal-system-error (mg-state, 'resource-error)
    else case on car (stmt):
         case = no-op-mg
         then mg-state
         case = signal-mg
          then set-condition (mg-state, signalled-condition (stmt))
         case = prog2-mg
          then mg-meaning-r (prog2-right-branch (stmt),
                             proc-list,
                             mg-meaning-r (prog2-left-branch (stmt),
                                           proc-list,
                                           mg-state,
                                           $n - 1$,
                                           sizes),
                             $n - 1$,
                             sizes)
         case = loop-mg
          then remove-leave (mg-meaning-r (stmt,
                                           proc-list,
                                           mg-meaning-r (loop-body (stmt),
                                                         proc-list,
                                                    mg-state,
                                                    $n - 1$,
                                                    sizes),
                                     $n - 1$,
                                     sizes))
case = if-mg
 then if mg-expression-falsep (if-condition (stmt), mg-state)
       then mg-meaning-r (if-false-branch (stmt),
                          proc-list,
                          mg-state,
                          $n - 1$,
                          sizes)
       else mg-meaning-r (if-true-branch (stmt),
                          proc-list,
                          mg-state,
                          $n - 1$,
                          sizes) endif
case = begin-mg
 then if cc (mg-meaning-r (begin-body (stmt),
                           proc-list,
                           mg-state,
                           $n - 1$,
                           sizes))
          ∈   when-labels (stmt)
       then mg-meaning-r (when-handler (stmt),
                          proc-list,
                          set-condition (mg-meaning-r (begin-body (stmt),
                                                       proc-list,
                                                       mg-state,
                                                       $n - 1$,
                                                       sizes),
                                         'normal),
                          $n - 1$,
                          sizes)
       else mg-meaning-r (begin-body (stmt),
                          proc-list,
                          mg-state,
                          $n - 1$,
                          sizes) endif
case = proc-call-mg
 then map-call-effects (mg-meaning-r (def-body (fetch-called-def (stmt,
                                                                   proc-list)),
                                      proc-list,
                                      make-call-environment (mg-state,
                                                                        stmt,
                                                                        fetch-called-def (stmt,
                                                                                          proc-list)),
                                                      n − 1,
                                                      list (t-size (sizes)
                                                               +    data-length (def-locals (fetch-called-def (stmt,
                                                                                                                proc-list))),
                                                               c-size (sizes)
                                                               +    (2
                                                                     +    length (def-locals (fetch-called-def (stmt,
                                                                                                                proc-list)))
                                                                     +    length (def-formals (fetch-called-def (stmt,
                                                                                                                  proc-list)))
                                            fetch-called-def (stmt, proc-list),
                                            stmt,
                                            mg-state)
                         case = predefined-proc-call-mg
                          then mg-meaning-predefined-proc-call (stmt, mg-state)
                         otherwise mg-state endcase endif
```

THEOREM: map-call-effects-preserves-resource-errorp

```
 resource-errorp (new-state)
 →    resource-errorp (map-call-effects (new-state, def, stmt, old-state))
```

THEOREM: map-call-effects-preserves-mg-psw

```
 mg-psw (map-call-effects (new-state, def, stmt, old-state))
 =    mg-psw (new-state)
```

THEOREM: mg-meaning-predefined-proc-call-preserves-resource-error

```
 mg-psw (mg-meaning-predefined-proc-call (stmt, mg-state)) = mg-psw (mg-state)
```

THEOREM: resource-errors-propogate

```
 resource-errorp (mg-state)
 →    resource-errorp (mg-meaning-r (stmt, proc-list, mg-state, n, sizes))
```

THEOREM: resource-errors-propogate2

```
 (mg-psw (mg-state) ≠ 'run)
 →    (mg-psw (mg-meaning-r (stmt, proc-list, mg-state, n, sizes)) ≠ 'run)
```

```
;; This lemma shows that in the absence of resource errors, the two interpreters
;; are equivalent.
```

THEOREM: mg-meaning-equivalence

```
 (¬ resource-errorp (mg-meaning-r (stmt, proc-list, mg-state, n, sizes)))
 →    (mg-meaning-r (stmt, proc-list, mg-state, n, sizes)
        =    mg-meaning (stmt, proc-list, mg-state, n))
```

Theorem: lessp-preserves-difference-lessp
$((y < (a - \textit{t-size1})) \land (\textit{t-size1} \not< \textit{t-size2}))$
$\rightarrow \quad ((y < (a - \textit{t-size2})) = \mathbf{t})$

Theorem: map-call-effects-preserves-resource-errorp2
resource-errorp (map-call-effects (*new-state*, *stmt*, *def*, *old-state*))
$=$ resource-errorp (*new-state*)

```
;; KEY POINT, t-size and c-size are the amount used, not the amount left
```

Theorem: more-resources-preserves-resources-adequatep2
((t-size (*sizes1*) $\not<$ t-size (*sizes2*))
$\land$ (c-size (*sizes1*) $\not<$ c-size (*sizes2*))
$\land$ ($\neg$ resources-inadequatep (*stmt*, *proc-list*, *sizes1*)))
$\rightarrow$ (resources-inadequatep (*stmt*, *proc-list*, *sizes2*) $=$ **f**)

Event: Disable mg-meaning-equivalence.

Definition:

```
meaning-induction-hint0 (stmt, proc-list, mg-state, n, sizes1, sizes2)
=   if $n \simeq 0$ then t
    elseif resources-inadequatep (stmt, proc-list, sizes1) then t
    elseif $\neg$ normal (mg-state) then t
    elseif 'no-op-mg = car (stmt) then t
    elseif 'signal-mg = car (stmt) then t
    elseif 'prog2-mg = car (stmt)
    then meaning-induction-hint0 (prog2-left-branch (stmt),
                                  proc-list,
                                  mg-state,
                                  $n - 1$,
                                  sizes1,
                                  sizes2)
         $\land$  meaning-induction-hint0 (prog2-right-branch (stmt),
                                     proc-list,
                                     mg-meaning-r (prog2-left-branch (stmt),
                                                   proc-list,
                                                   mg-state,
                                                   $n - 1$,
                                                   sizes1),
                                     $n - 1$,
                                     sizes1,
                                     sizes2)
    elseif 'loop-mg = car (stmt)
```

```
then meaning-induction-hint0 (loop-body (stmt),
                              proc-list,
                              mg-state,
                              n - 1,
                              sizes1,
                              sizes2)
     ∧  meaning-induction-hint0 (stmt,
                                 proc-list,
                                 mg-meaning-r (loop-body (stmt),
                                               proc-list,
                                               mg-state,
                                               n - 1,
                                               sizes1),
                                 n - 1,
                                 sizes1,
                                 sizes2)
elseif 'if-mg = car (stmt)
then meaning-induction-hint0 (if-false-branch (stmt),
                              proc-list,
                              mg-state,
                              n - 1,
                              sizes1,
                              sizes2)
     ∧  meaning-induction-hint0 (if-true-branch (stmt),
                                 proc-list,
                                 mg-state,
                                 n - 1,
                                 sizes1,
                                 sizes2)
elseif 'begin-mg = car (stmt)
then meaning-induction-hint0 (begin-body (stmt),
                              proc-list,
                              mg-state,
                              n - 1,
                              sizes1,
                              sizes2)
     ∧  meaning-induction-hint0 (when-handler (stmt),
                                 proc-list,
                                 set-condition (mg-meaning-r (begin-body (stmt),
                                                              proc-list,
                                                              mg-state,
                                                              n - 1,
                                                              sizes1),
                                                'normal),
                                 n − 1,
                                 sizes1,
                                 sizes2)
elseif 'proc-call-mg = car (stmt)
then meaning-induction-hint0 (def-body (fetch-called-def (stmt, proc-list)),
                              proc-list,
                              make-call-environment (mg-state,
                                                     stmt,
                                                     fetch-called-def (stmt,
                                                                       proc-list)),
                              n − 1,
                              list (t-size (sizes1)
                                 +   data-length (def-locals (fetch-called-def (stmt,
                                                                               proc-list))),
                                 c-size (sizes1)
                                 +   (2
                                      +   length (def-locals (fetch-called-def (stmt,
                                                                                proc-list)))
                                      +   length (def-formals (fetch-called-def (stmt,
                                                                                 proc-list))))),
                              list (t-size (sizes2)
                                 +   data-length (def-locals (fetch-called-def (stmt,
                                                                               proc-list))),
                                 c-size (sizes2)
                                 +   (2
                                      +   length (def-locals (fetch-called-def (stmt,
                                                                                proc-list)))
                                      +   length (def-formals (fetch-called-def (stmt,
                                                                                 proc-list))))))
elseif 'predefined-proc-call-mg = car (stmt) then t
else f endif
```

THEOREM: mg-meaning-equivalence3

```
((¬ resource-errorp (mg-meaning-r (stmt, proc-list, mg-state, n, sizes1)))
 ∧  (t-size (sizes1) ≮ t-size (sizes2))
 ∧  (c-size (sizes1) ≮ c-size (sizes2)))
→  (mg-meaning-r (stmt, proc-list, mg-state, n, sizes1)
     =  mg-meaning-r (stmt, proc-list, mg-state, n, sizes2))
```

EVENT: Disable mg-meaning-equivalence3.

THEOREM: mg-meaning-equivalence4

```
((¬ resource-errorp (mg-meaning-r (stmt, proc-list, mg-state, n, sizes1)))
 ∧  (t-size (sizes1) ≮ t-size (sizes2))
```

∧ (c-size (*sizes1*) ≮ c-size (*sizes2*)))
→ (mg-meaning-r (*stmt*, *proc-list*, *mg-state*, *n*, *sizes2*)
= mg-meaning-r (*stmt*, *proc-list*, *mg-state*, *n*, *sizes1*))

Event: Disable mg-meaning-equivalence4.

Theorem: more-resources-preserves-not-resource-errorp0
((*t-size1* ≮ *t-size2*)
∧ (*c-size1* ≮ *c-size2*)
∧ resource-errorp (mg-meaning-r (*stmt*,
proc-list,
mg-state,
n,
list (*t-size2*, *c-size2*))))
→ resource-errorp (mg-meaning-r (*stmt*,
proc-list,
mg-state,
n,
list (*t-size1*, *c-size1*)))

Theorem: more-resources-preserves-not-resource-errorp
((*t-size1* ≮ *t-size2*)
∧ (*c-size1* ≮ *c-size2*)
∧ (¬ resource-errorp (mg-meaning-r (*stmt*,
proc-list,
mg-state,
n,
list (*t-size1*, *c-size1*)))))
→ (¬ resource-errorp (mg-meaning-r (*stmt*,
proc-list,
mg-state,
n,
list (*t-size2*, *c-size2*))))

Theorem: mg-meaning-equivalence2
((¬ resource-errorp (mg-meaning-r (*stmt*,
proc-list,
mg-state,
n,
list (*t-size1*, *c-size1*))))
∧ (*t-size1* ≮ *t-size2*)
∧ (*c-size1* ≮ *c-size2*))
→ (mg-meaning-r (*stmt*, *proc-list*, *mg-state*, *n*, list (*t-size2*, *c-size2*))
= mg-meaning (*stmt*, *proc-list*, *mg-state*, *n*))

THEOREM: mg-meaning-mg-meaning-r-resource-error-equivalence

```
(¬ resource-errorp (mg-meaning-r (stmt, proc-list, mg-state, n, sizes)))
→   (¬ resource-errorp (mg-meaning (stmt, proc-list, mg-state, n)))
```

THEOREM: zerop-n-mg-meaning

```
(n ≃ 0)
→   (mg-meaning (stmt, proc-list, mg-state, n)
        =   signal-system-error (mg-state, 'timed-out))
```

THEOREM: not-normal-mg-meaning

```
((n ≄ 0) ∧ (¬ normal (mg-state)))
→   (mg-meaning (stmt, proc-list, mg-state, n) = mg-state)
```

THEOREM: proc-call-meaning-2

```
(car (stmt) = 'proc-call-mg)
→   (mg-meaning (stmt, proc-list, mg-state, n)
        =   if n ≃ 0 then signal-system-error (mg-state, 'timed-out)
            elseif ¬ normal (mg-state)  then mg-state
            else map-call-effects (mg-meaning (def-body (fetch-called-def (stmt,
                                                                          proc-list)),
                                                proc-list,
                                                make-call-environment (mg-state,
                                                                       stmt,
                                                                       fetch-called-def (stmt,
                                                                                         proc-list)),
                                                n − 1),
                                  fetch-called-def (stmt, proc-list),
                                  stmt,
                                  mg-state) endif)
```

```
;; The versions for mg-meaning-r
```

THEOREM: zerop-n-mg-meaning-r

```
(n ≃ 0)
→   (mg-meaning-r (stmt, proc-list, mg-state, n, sizes)
        =   signal-system-error (mg-state, 'timed-out))
```

THEOREM: not-normal-mg-meaning-r

```
((n ≄ 0) ∧ (¬ normal (mg-state)))
→   (mg-meaning-r (stmt, proc-list, mg-state, n, sizes) = mg-state)
```

THEOREM: resources-inadequatep-mg-meaning-r

```
((n ≄ 0)
 ∧   normal (mg-state)
```

$\wedge$ resources-inadequatep (*stmt*, *proc-list*, *sizes*))
$\rightarrow$ (mg-meaning-r (*stmt*, *proc-list*, *mg-state*, *n*, *sizes*)
  = signal-system-error (*mg-state*, `'resource-error`))

Event: Disable mg-meaning-r.

Theorem: call-cond-lists-lengths-match
((`'proc-call-mg` = car (*stmt*))
$\wedge$ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*))
$\rightarrow$ (length (def-conds (fetch-called-def (*stmt*, *proc-list*)))
  = length (call-conds (*stmt*)))

Event: Disable call-cond-lists-lengths-match.

Theorem: set-alist-value-preserves-mg-alistp
(mg-alistp (*alist*) $\wedge$ ok-mg-valuep (*val*, cadr (assoc (*name*, *alist*))))
$\rightarrow$ mg-alistp (set-alist-value (*name*, *val*, *alist*))

Theorem: mg-alistps-append
(mg-alistp (*lst1*) $\wedge$ mg-alistp (*lst2*)) $\rightarrow$ mg-alistp (append (*lst1*, *lst2*))

```
;; The recognizer has a structure called the name-alist of the following
;; form:
;;        (... (namei typei other-stuff) ...)
;; This allows the identification of the types of variables.  It should be
;; the case that the values of the variables in the meaning alist correspond
;; to their types on the name-alist.

;; This says that a variable on the name-alist has the same name and type as
;; on the variable alist.  I need to know this to guarantee that the checks for
;; legality are visible in the execution world.

;; The only time a name-alist is ever created in the recognizer is from the formal
;; and locals lists.  There is really no reason why the same structure couldn't
;; be adhered to in the interpreter.  That is, the order of the variables could
;; be maintained.  >> Where would the initial values of the variables come from
;; in that case?
;; I could have an initial alist which serves both as the name-alist and var-alist
;; for the execution of the stmt.  That is, the initial alist only has to be an
;; assignment of values to the vars which is consistent with the types.

;; Notice that this checks a very strong correspondence between the two alists.
;; Each element agrees in name and type.  The idea here is that the name and var
;; alists are really identical except that some values have been added on the
;; var alists.

;; Notice that this correspondence really defines an equivalence relation.  I
;; don't know that I'll need the full power of this.
```

THEOREM: signatures-match-preserves-get-m-type
signatures-match ($alist1$, $alist2$)
$\rightarrow$ (cadr (assoc (x, $alist1$)) = cadr (assoc (x, $alist2$)))

EVENT: Disable signatures-match-preserves-get-m-type.

THEOREM: signatures-match-preserves-definedp
signatures-match ($alist1$, $alist2$)
$\rightarrow$ (definedp (x, $alist1$) = definedp (x, $alist2$))

THEOREM: signatures-match-preserves-ok-actual-params-list
(signatures-match ($alist1$, $alist2$) $\wedge$ ok-actual-params-list (lst, $alist1$))
$\rightarrow$ ok-actual-params-list (lst, $alist2$)

THEOREM: set-alist-value-preserves-plistp
plistp (lst) $\rightarrow$ plistp (set-alist-value ($name$, val, lst))

THEOREM: signatures-match-preserves-boolean-identifierp
(signatures-match ($alist1$, $alist2$) $\wedge$ boolean-identifierp (b, $alist1$))
$\rightarrow$ boolean-identifierp (b, $alist2$)

EVENT: Disable signatures-match-preserves-boolean-identifierp.

THEOREM: signatures-match-preserves-int-identifierp
(signatures-match ($alist1$, $alist2$) $\wedge$ int-identifierp (x, $alist1$))
$\rightarrow$ int-identifierp (x, $alist2$)

EVENT: Disable signatures-match-preserves-int-identifierp.

THEOREM: signatures-match-preserves-character-identifierp
(signatures-match ($alist1$, $alist2$) $\wedge$ character-identifierp (x, $alist1$))
$\rightarrow$ character-identifierp (x, $alist2$)

EVENT: Disable signatures-match-preserves-character-identifierp.

THEOREM: signatures-match-preserves-array-identifierp
(signatures-match ($alist1$, $alist2$) $\wedge$ array-identifierp (x, $alist1$))
$\rightarrow$ array-identifierp (x, $alist2$)

Event: Disable signatures-match-preserves-array-identifierp.

Theorem: signatures-match-preserves-simple-identifierp
(signatures-match (*alist1*, *alist2*) ∧ simple-identifierp (*x*, *alist1*))
→ simple-identifierp (*x*, *alist2*)

Theorem: signatures-match-preserves-simple-typed-identifierp
(signatures-match (*alist1*, *alist2*) ∧ simple-typed-identifierp (*x*, *type*, *alist1*))
→ simple-typed-identifierp (*x*, *type*, *alist2*)

Theorem: signatures-match-preserves-data-param-lists-match
(signatures-match (*alist1*, *alist2*)
∧ data-param-lists-match (*actuals*, *formals*, *alist1*))
→ data-param-lists-match (*actuals*, *formals*, *alist2*)

Theorem: signatures-match-preserves-ok-predefined-proc-args
(signatures-match (*alist1*, *alist2*)
∧ ok-predefined-proc-args (*name*, *actuals*, *alist1*))
→ ok-predefined-proc-args (*name*, *actuals*, *alist2*)

Theorem: signatures-match-preserves-ok-predefined-proc-call
(signatures-match (*alist1*, *alist2*) ∧ ok-predefined-proc-call (*stmt*, *alist1*))
→ ok-predefined-proc-call (*stmt*, *alist2*)

Theorem: signatures-match-preserves-ok-mg-statement
(signatures-match (*alist1*, *alist2*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *alist1*, *proc-list*))
→ ok-mg-statement (*stmt*, *r-cond-list*, *alist2*, *proc-list*)

Theorem: mg-meaning-predefined-proc-call-preserves-signatures-match
plistp (mg-alist (*mg-state*))
→ signatures-match (mg-alist (*mg-state*),
mg-alist (mg-meaning-predefined-proc-call (*stmt*,
mg-state)))

Theorem: copy-out-params-preserves-signatures-match
plistp (*v*) → signatures-match (*v*, copy-out-params (*x*, *y*, *z*, *v*))

Theorem: signatures-match-preserves-plistp
(plistp (*x*) ∧ signatures-match (*x*, *y*)) → plistp (*y*)

Theorem: mg-meaning-preserves-signatures-match
plistp (mg-alist (*mg-state*))
→ signatures-match (mg-alist (*mg-state*),
mg-alist (mg-meaning (*stmt*, *proc-list*, *mg-state*, *n*)))

Theorem: cadr-litatom-implies-definedp
(cadr (assoc (x, *alist*)) $\neq$ `0`) → definedp (x, *alist*)

Event: Disable cadr-litatom-implies-definedp.

Theorem: call-param-alist-mg-alistp
(mg-alistp (*mg-alist*)
$\land$ ok-mg-formal-data-params-plistp (*formals*)
$\land$ data-param-lists-match (*actuals*, *formals*, *name-alist*)
$\land$ signatures-match (*mg-alist*, *name-alist*))
→ mg-alistp (make-call-param-alist (*formals*, *actuals*, *mg-alist*))

Theorem: call-locals-alist-mg-alistp
ok-mg-local-data-plistp (*locals-list*) → mg-alistp (*locals-list*)

Theorem: make-call-var-alist-mg-alistp
((`'proc-call-mg` = car (*stmt*))
$\land$ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
$\land$ ok-mg-def-plistp (*proc-list*)
$\land$ mg-alistp (*mg-alist*)
$\land$ signatures-match (*mg-alist*, *name-alist*))
→ mg-alistp (make-call-var-alist (*mg-alist*,
stmt,
fetch-called-def (*stmt*, *proc-list*)))

Theorem: ok-mg-statep-preserved-call-case
((`'proc-call-mg` = car (*stmt*))
$\land$ ok-mg-statep (*mg-state*, *r-cond-list*)
$\land$ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
$\land$ signatures-match (mg-alist (*mg-state*), *name-alist*)
$\land$ ok-mg-def-plistp (*proc-list*))
→ ok-mg-statep (make-call-environment (*mg-state*,
stmt,
fetch-called-def (*stmt*, *proc-list*)),
make-cond-list (fetch-called-def (*stmt*, *proc-list*)))

Theorem: call-formal-signatures-match
signatures-match (make-alist-from-formals (*formals*),
make-call-param-alist (*formals*, *actuals*, *mg-alist*))

Theorem: call-formal-signatures-match2
signatures-match (make-call-param-alist (*formals*, *actuals*, *mg-alist*),
make-alist-from-formals (*formals*))

Theorem: call-local-signatures-match
plistp (*locals*) → signatures-match (make-alist-from-formals (*locals*), *locals*)

THEOREM: call-local-signatures-match2
signatures-match (*locals*, make-alist-from-formals (*locals*))

THEOREM: call-signatures-match1
plistp (def-locals (*def*))
→ signatures-match (make-name-alist (*def*),
make-call-var-alist (*mg-alist*, *stmt*, *def*))

THEOREM: call-signatures-match2
signatures-match (make-call-var-alist (*mg-alist*, *stmt*, *def*), make-name-alist (*def*))

THEOREM: call-signatures-match3
signatures-match (mg-alist (make-call-environment (*mg-state*,
stmt,
fetch-called-def (*stmt*,
proc-list))),
make-name-alist (fetch-called-def (*stmt*, *proc-list*)))

DEFINITION:
formal-types-preserved (*formals*, *alist*)

```
=   if formals ≃ nil then t
    else definedp (caar (formals), alist)
         ∧  (cadar (formals)
              =  cadr (assoc (caar (formals), alist)))
         ∧  formal-types-preserved (cdr (formals), alist) endif
```

THEOREM: formal-types-preserved-append
formal-types-preserved (*formals*, *lst1*)
→ formal-types-preserved (*formals*, append (*lst1*, *lst2*))

THEOREM: formal-types-unaffected-by-extra-binding
(car (x) $\notin$ listcars (y))
→ (formal-types-preserved (y, cons (x, z)) = formal-types-preserved (y, z))

THEOREM: formal-types-preserved-in-call-param-alist
(ok-mg-formal-data-params-plistp (*formals*)
∧ no-duplicates (listcars (*formals*)))
→ formal-types-preserved (*formals*,
make-call-param-alist (*formals*, *actuals*, *mg-alist*))

THEOREM: formal-types-preserved-in-call-environment
((`'proc-call-mg` = car (*stmt*))
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ ok-mg-def-plistp (*proc-list*)
∧ mg-alistp (mg-alist (*mg-state*)))

```
 →   formal-types-preserved (def-formals (fetch-called-def (stmt, proc-list)),
                             mg-alist (make-call-environment (mg-state,
                                                              stmt,
                                                              fetch-called-def (stmt,
                                                                                proc-list))))
```

```
Theorem: copy-out-params-preserves-mg-alistp
 (mg-alistp (old-alist)
  ∧   mg-alistp (new-alist)
  ∧   formal-types-preserved (formals, new-alist)
  ∧   data-param-lists-match (actuals, formals, name-alist)
  ∧   signatures-match (old-alist, name-alist)
  ∧   ok-mg-formal-data-params-plistp (formals))
 →   mg-alistp (copy-out-params (formals, actuals, new-alist, old-alist))
```

```
Theorem: formal-types-preserved-in-matching-signatures
 (mg-name-alistp (old-alist)
  ∧   formal-types-preserved (formals, old-alist)
  ∧   signatures-match (old-alist, new-alist))
 →   formal-types-preserved (formals, new-alist)
```

```
;; This is the case needed for map-call-effects-preserves-ok-state.
```

```
Theorem: map-call-effects-preserves-mg-alistp
 (('proc-call-mg = car (stmt))
  ∧   mg-alistp (mg-alist (mg-state))
  ∧   ok-mg-statement (stmt, r-cond-list, name-alist, proc-list)
  ∧   signatures-match (mg-alist (mg-state), name-alist)
  ∧   ok-mg-def-plistp (proc-list)
  ∧   ok-mg-def (fetch-called-def (stmt, proc-list), proc-list)
  ∧   mg-alistp (mg-alist (mg-meaning (def-body (fetch-called-def (stmt,
                                                                   proc-list)),
                                       proc-list,
                                       make-call-environment (mg-state,
                                                              stmt,
                                                              fetch-called-def (stmt,
                                                                                proc-list)),
                                       n − 1))))
 →   mg-alistp (copy-out-params (def-formals (fetch-called-def (stmt, proc-list)),
                                 call-actuals (stmt),
                                 mg-alist (mg-meaning (def-body (fetch-called-def (stmt,
                                                                                   proc-list)),
                                                       proc-list,
                                                       make-call-environment (mg-state,
                                                                                    stmt,
                                                                                    fetch-called-def (stmt,
                                                                                                      proc-list)),
                                                                n − 1)),
                                              mg-alist (mg-state)))
```

Theorem: convert-condition1-membership

```
 (length (def-conds) = length (call-conds))
 →    (convert-condition1 (cc, def-conds, call-conds)
        ∈    cons ('routineerror, call-conds))
```

Theorem: cond-identifier-plistp-preserves-membership

```
 ((cc ∈ lst1)
  ∧    cond-identifier-plistp (lst1, lst2)
  ∧    (cc ≠ 'routineerror))
 →    (cc ∈ lst2)
```

Event: Disable cond-identifier-plistp-preserves-membership.

Theorem: cons-preserves-membership

```
 ((x ∈ cons (y, z)) ∧ (x ≠ y)) → (x ∈ z)
```

Event: Disable cons-preserves-membership.

Theorem: cond-identifier-conversion-litatom

```
 ((length (def-conds) = length (call-conds))
  ∧    cond-identifier-plistp (call-conds, cond-list))
 →    litatom (convert-condition1 (cc, def-conds, call-conds))
```

Event: Disable cond-identifier-conversion-litatom.

Theorem: map-call-effects-preserves-ok-state

```
 (('proc-call-mg = car (stmt))
  ∧    ok-mg-statep (mg-state, r-cond-list)
  ∧    ok-mg-statement (stmt, r-cond-list, name-alist, proc-list)
  ∧    signatures-match (mg-alist (mg-state), name-alist)
  ∧    ok-mg-def-plistp (proc-list)
  ∧    ok-mg-statep (mg-meaning (def-body (fetch-called-def (stmt, proc-list)),
                                 proc-list,
                                 make-call-environment (mg-state,
                                                        stmt,
                                                        fetch-called-def (stmt,
                                                                          proc-list)),
                                  n - 1),
                    make-cond-list (fetch-called-def (stmt, proc-list))))
→    ok-mg-statep (map-call-effects (mg-meaning (def-body (fetch-called-def (stmt,
                                                                            proc-list)),
                                                 proc-list,
                                                 make-call-environment (mg-state,
                                                                        stmt,
                                                                        fetch-called-def (stmt,
                                                                                          proc-list)),
                                                 n - 1),
                                     fetch-called-def (stmt, proc-list),
                                     stmt,
                                     mg-state),
                    r-cond-list)
```

Theorem: simple-typed-literalp-ok-valuep
simple-typed-literalp (*exp*, *type*) → ok-mg-valuep (*exp*, *type*)

Event: Enable ok-predefined-proc-call.

Event: Enable ok-predefined-proc-args.

Event: Enable ok-mg-statement.

Event: Enable simple-identifierp-implies-definedp.

Event: Enable int-identifierp-implies-definedp.

Event: Enable boolean-identifierp-implies-definedp.

Event: Enable character-identifierp-implies-definedp.

Event: Enable boolean-identifierp.

Event: Enable character-identifierp.

Event: Enable int-identifierp.

Event: Disable mg-bool.

Event: Disable mg-or-bool.

Event: Disable mg-and-bool.

Event: Disable mg-not-bool.

Theorem: tag-length-plistp
length-plistp (tag (x, y), `2`)

Theorem: car-tag
(car (tag (x, y)) = x) ∧ (cdr (tag (x, y)) = list (y))

Theorem: simple-typed-literalp-boolean-literals
simple-typed-literalp (tag (`'boolean-mg`, `'true-mg`), `'boolean-mg`)
∧ simple-typed-literalp (tag (`'boolean-mg`, `'false-mg`),
`'boolean-mg`)

Theorem: simple-typed-literalp-boolean-mg-bool
simple-typed-literalp (mg-bool (x), `'boolean-mg`)

Theorem: simple-typed-literalp-boolean-mg-bool-not
simple-typed-literalp (tag (`'boolean-mg`, mg-not-bool (x)), `'boolean-mg`)

Theorem: ok-mg-valuep-int-mg
ok-mg-valuep (tag (`'int-mg`, x), `'int-mg`)
= small-integerp (x, mg-word-size)

Theorem: boolean-literalp-tag-untag
boolean-literalp (x) → boolean-literalp (tag (`'boolean-mg`, untag (x)))

Theorem: boolean-identifier-boolean-literal-value
(mg-alistp (*mg-alist*) ∧ (cadr (assoc (x, *mg-alist*)) = `'boolean-mg`))
→ boolean-literalp (caddr (assoc (x, *mg-alist*)))

Theorem: boolean-identifier-boolean-literalp
(boolean-identifierp (x, *name-alist*)
∧ mg-alistp (*mg-alist*)
∧ signatures-match (*mg-alist*, *name-alist*))
→ boolean-literalp (caddr (assoc (x, *mg-alist*)))

Theorem: mg-meaning-mg-simple-variable-assignment-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-simple-variable-assignment`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)

∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

THEOREM: mg-meaning-mg-simple-constant-assignment-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-simple-constant-assignment`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

THEOREM: mg-meaning-mg-simple-variable-eq-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-simple-variable-eq`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

THEOREM: mg-meaning-mg-simple-constant-eq-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-simple-constant-eq`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

THEOREM: mg-meaning-mg-integer-le-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-integer-le`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

THEOREM: mg-meaning-mg-integer-unary-minus-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-integer-unary-minus`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)

∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

Theorem: mg-meaning-mg-integer-add-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-integer-add`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

Theorem: mg-meaning-mg-integer-subtract-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-integer-subtract`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

Theorem: mg-meaning-mg-boolean-or-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-boolean-or`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

Theorem: mg-meaning-mg-boolean-and-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-boolean-and`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

Theorem: mg-meaning-mg-boolean-not-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-boolean-not`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)

∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

Theorem: simple-typed-literal-list-elements
(simple-typed-literal-plistp (*lst*, *type*) ∧ (*i* < length (*lst*)))
→ simple-typed-literalp (get (*i*, *lst*), *type*)

Theorem: index-array-mg-alist-elementp
(mg-alistp (*mg-alist*)
∧ array-identifierp (*a*, *mg-alist*)
∧ (*i* < array-length (cadr (assoc (*a*, *mg-alist*)))))
→ simple-typed-literalp (get (*i*, caddr (assoc (*a*, *mg-alist*))),
array-elemtype (cadr (assoc (*a*, *mg-alist*))))

Theorem: put-preserves-simple-typed-literal-plistp
(simple-typed-literal-plistp (*lst*, *type*)
∧ (*i* < length (*lst*))
∧ simple-typed-literalp (*val*, *type*))
→ simple-typed-literal-plistp (put (*val*, *i*, *lst*), *type*)

Theorem: simple-type-literal-plistp-ok-valuep
(array-mg-type-refp (*type*)
∧ simple-typed-literal-plistp (*exp*, array-elemtype (*type*))
∧ (length (*exp*) = array-length (*type*)))
→ ok-mg-valuep (*exp*, *type*)

Theorem: mg-meaning-mg-index-array-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
∧ (call-name (*stmt*) = `'mg-index-array`)
∧ ok-mg-statep (*mg-state*, *r-cond-list*)
∧ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
∧ signatures-match (mg-alist (*mg-state*), *name-alist*))
→ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

Theorem: array-identifiers-have-array-types
(array-identifierp (*x*, *mg-alist*) ∧ mg-alistp (*mg-alist*))
→ array-mg-type-refp (cadr (assoc (*x*, *mg-alist*)))

Theorem: array-identifiers-have-array-types2
(array-identifierp (*a*, *mg-alist*) ∧ mg-alistp (*mg-alist*))
→ simple-typed-literal-plistp (caddr (assoc (*a*, *mg-alist*)),
array-elemtype (cadr (assoc (*a*, *mg-alist*))))

THEOREM: array-identifier-lengths-match
(array-identifierp (*a*, *mg-alist*) $\wedge$ mg-alistp (*mg-alist*))
$\rightarrow$ (length (caddr (assoc (*a*, *mg-alist*)))
  = array-length (cadr (assoc (*a*, *mg-alist*))))

THEOREM: simple-typed-identifier-has-simple-typed-literal-value
(mg-alistp (*mg-alist*) $\wedge$ simple-typed-identifierp (*x*, *type*, *mg-alist*))
$\rightarrow$ simple-typed-literalp (caddr (assoc (*x*, *mg-alist*)), *type*)

THEOREM: mg-meaning-mg-array-element-assignment-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
$\wedge$ (call-name (*stmt*) = `'mg-array-element-assignment`)
$\wedge$ ok-mg-statep (*mg-state*, *r-cond-list*)
$\wedge$ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
$\wedge$ signatures-match (mg-alist (*mg-state*), *name-alist*))
$\rightarrow$ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

THEOREM: mg-meaning-predefined-proc-call-preserves-ok-mg-statep
((car (*stmt*) = `'predefined-proc-call-mg`)
$\wedge$ ok-mg-statep (*mg-state*, *r-cond-list*)
$\wedge$ ok-mg-statement (*stmt*, *r-cond-list*, *name-alist*, *proc-list*)
$\wedge$ signatures-match (mg-alist (*mg-state*), *name-alist*))
$\rightarrow$ ok-mg-statep (mg-meaning-predefined-proc-call (*stmt*, *mg-state*),
r-cond-list)

THEOREM: set-condition-normal-preserves-ok-mg-statep
ok-mg-statep (*state*, *cond-list1*)
$\rightarrow$ ok-mg-statep (set-condition (*state*, `'normal`), *cond-list2*)

THEOREM: append-conditions-preserves-ok-mg-statep
(ok-mg-statep (*state*, append (*lst*, *lst2*)) $\wedge$ (cc (*state*) $\notin$ *lst*))
$\rightarrow$ ok-mg-statep (*state*, *lst2*)

DEFINITION:
meaning-induction-hint (*stmt*, *proc-list*, *mg-state*, *n*, *name-alist*, *r-cond-list*)

```
=  if n ≃ 0 then t
   elseif ¬ normal (mg-state) then t
   elseif 'no-op-mg = car (stmt) then t
   elseif 'signal-mg = car (stmt) then t
   elseif 'prog2-mg = car (stmt)
   then meaning-induction-hint (prog2-left-branch (stmt),
                                proc-list,
                                mg-state,
                                n - 1,
                        name-alist,
                        r-cond-list)
    ∧   meaning-induction-hint (prog2-right-branch (stmt),
                                proc-list,
                                mg-meaning (prog2-left-branch (stmt),
                                            proc-list,
                                            mg-state,
                                            $n - 1$),
                                $n - 1$,
                                name-alist,
                                r-cond-list)
elseif 'loop-mg = car (stmt)
then meaning-induction-hint (loop-body (stmt),
                             proc-list,
                             mg-state,
                             $n - 1$,
                             name-alist,
                             cons ('leave, r-cond-list))
    ∧   meaning-induction-hint (stmt,
                                proc-list,
                                mg-meaning (loop-body (stmt),
                                            proc-list,
                                            mg-state,
                                            $n - 1$),
                                $n - 1$,
                                name-alist,
                                r-cond-list)
elseif 'if-mg = car (stmt)
then meaning-induction-hint (if-false-branch (stmt),
                             proc-list,
                             mg-state,
                             $n - 1$,
                             name-alist,
                             r-cond-list)
    ∧   meaning-induction-hint (if-true-branch (stmt),
                                proc-list,
                                mg-state,
                                $n - 1$,
                                name-alist,
                                r-cond-list)
elseif 'begin-mg = car (stmt)
then meaning-induction-hint (begin-body (stmt),
                             proc-list,
                             mg-state,
                                      n − 1,
                                      name-alist,
                                      append (when-labels (stmt), r-cond-list))
           ∧    meaning-induction-hint (when-handler (stmt),
                                           proc-list,
                                           set-condition (mg-meaning (begin-body (stmt),
                                                                      proc-list,
                                                                      mg-state,
                                                                      n − 1),
                                                          'normal),
                                           n − 1,
                                           name-alist,
                                           r-cond-list)
     elseif 'proc-call-mg = car (stmt)
     then meaning-induction-hint (def-body (fetch-called-def (stmt, proc-list)),
                                      proc-list,
                                      make-call-environment (mg-state,
                                                             stmt,
                                                             fetch-called-def (stmt,
                                                                               proc-list)),
                                      n − 1,
                                      make-name-alist (fetch-called-def (stmt,
                                                                         proc-list)),
                                      make-cond-list (fetch-called-def (stmt,
                                                                        proc-list)))
     elseif 'predefined-proc-call-mg = car (stmt)  then t
     else f endif
```

Event: Disable ok-predefined-proc-call.

Event: Disable ok-predefined-proc-args.

Event: Disable ok-mg-statement.

Event: Disable simple-identifierp-implies-definedp.

Event: Disable int-identifierp-implies-definedp.

Event: Disable boolean-identifierp-implies-definedp.

Event: Disable character-identifierp-implies-definedp.

Event: Disable boolean-identifierp.

Event: Disable character-identifierp.

Event: Disable int-identifierp.

Event: Disable signatures-match-preserves-ok-mg-statement.

Event: Disable signatures-match-preserves-ok-predefined-proc-call.

Event: Disable signatures-match-preserves-ok-predefined-proc-args.

Event: Disable signatures-match-preserves-simple-identifierp.

Event: Disable not-member-listcars-not-assoc.

Event: Disable ok-mg-statep.

Event: Disable call-locals-alist-mg-alistp.

Theorem: removing-condition-preserves-ok-mg-statep

$$\begin{array}{ll} \multicolumn{2}{l}{(\text{ok-mg-statep}\,(\mathit{state},\ \text{cons}\,(x,\ \mathit{lst})) \land (\text{cc}\,(\mathit{state}) \neq x))} \\ \rightarrow & \text{ok-mg-statep}\,(\mathit{state},\ \mathit{lst}) \end{array}$$

Event: Disable removing-condition-preserves-ok-mg-statep.

Theorem: adding-condition-preserves-ok-mg-statep

$$\text{ok-mg-statep}\,(\mathit{state},\ \mathit{lst2}) \rightarrow \text{ok-mg-statep}\,(\mathit{state},\ \text{append}\,(\mathit{lst},\ \mathit{lst2}))$$

Event: Disable adding-condition-preserves-ok-mg-statep.

Theorem: mg-meaning-preserves-ok-mg-statep

$$\begin{array}{ll} \multicolumn{2}{l}{(\text{ok-mg-statep}\,(\mathit{mg\text{-}state},\ \mathit{r\text{-}cond\text{-}list})} \\ \land & \text{ok-mg-statement}\,(\mathit{stmt},\ \mathit{r\text{-}cond\text{-}list},\ \mathit{name\text{-}alist},\ \mathit{proc\text{-}list}) \\ \land & \text{signatures-match}\,(\text{mg-alist}\,(\mathit{mg\text{-}state}),\ \mathit{name\text{-}alist}) \\ \land & \text{ok-mg-def-plistp}\,(\mathit{proc\text{-}list})) \\ \rightarrow & \text{ok-mg-statep}\,(\text{mg-meaning}\,(\mathit{stmt},\ \mathit{proc\text{-}list},\ \mathit{mg\text{-}state},\ n),\ \mathit{r\text{-}cond\text{-}list}) \end{array}$$

Event: Disable mg-meaning-preserves-ok-mg-statep.

Theorem: mg-meaning-preserves-ok-cc

$$\begin{array}{ll}
\multicolumn{2}{l}{(\text{ok-mg-statep}\,(\textit{mg-state},\ \textit{r-cond-list})} \\
\ \wedge & \text{ok-mg-statement}\,(\textit{stmt},\ \textit{r-cond-list},\ \textit{name-alist},\ \textit{proc-list}) \\
\ \wedge & \text{signatures-match}\,(\text{mg-alist}\,(\textit{mg-state}),\ \textit{name-alist}) \\
\ \wedge & \text{ok-mg-def-plistp}\,(\textit{proc-list})) \\
\rightarrow & \text{ok-cc}\,(\text{cc}\,(\text{mg-meaning}\,(\textit{stmt},\ \textit{proc-list},\ \textit{mg-state},\ \textit{n})),\ \textit{r-cond-list})
\end{array}$$

Event: Disable mg-meaning-preserves-ok-cc.

Theorem: mg-meaning-preserves-mg-alistp

$$\begin{array}{ll}
\multicolumn{2}{l}{(\text{ok-mg-statep}\,(\textit{mg-state},\ \textit{r-cond-list})} \\
\ \wedge & \text{ok-mg-statement}\,(\textit{stmt},\ \textit{r-cond-list},\ \textit{name-alist},\ \textit{proc-list}) \\
\ \wedge & \text{signatures-match}\,(\text{mg-alist}\,(\textit{mg-state}),\ \textit{name-alist}) \\
\ \wedge & \text{ok-mg-def-plistp}\,(\textit{proc-list})) \\
\rightarrow & \text{mg-alistp}\,(\text{mg-alist}\,(\text{mg-meaning}\,(\textit{stmt},\ \textit{proc-list},\ \textit{mg-state},\ \textit{n})))
\end{array}$$

Event: Disable mg-meaning-preserves-mg-alistp.

Theorem: mg-meaning-condition-member-cond-list1

$$\begin{array}{ll}
\multicolumn{2}{l}{(\text{ok-mg-statep}\,(\textit{mg-state},\ \textit{r-cond-list})} \\
\ \wedge & \text{ok-mg-statement}\,(\textit{stmt},\ \textit{r-cond-list},\ \textit{name-alist},\ \textit{proc-list}) \\
\ \wedge & \text{ok-mg-def-plistp}\,(\textit{proc-list}) \\
\ \wedge & \text{signatures-match}\,(\text{mg-alist}\,(\textit{mg-state}),\ \textit{name-alist}) \\
\ \wedge & (\neg\ \text{resource-errorp}\,(\text{mg-meaning}\,(\textit{stmt},\ \textit{proc-list},\ \textit{mg-state},\ \textit{n})))) \\
\rightarrow & (\text{cc}\,(\text{mg-meaning}\,(\textit{stmt},\ \textit{proc-list},\ \textit{mg-state},\ \textit{n})) \\
 & \quad \in \quad \text{cons}\,(\texttt{'normal},\ \text{cons}\,(\texttt{'routineerror},\ \textit{r-cond-list})))
\end{array}$$

Event: Disable mg-meaning-condition-member-cond-list1.

Event: Make the library `"c3"`.

Index