```
;; Modified to remove the calls of DO-MUTUAL, which are now commented out.
;; The forms below them in capital letters are presumably the ones that DO-MUTUAL
;; generated.

; ************************************
;  Functions on Integers and Rationals
; ************************************

; integer and rational libraries
```

EVENT: Start with the library "r2".

DEFINITION:
ipower $(x,\ e)$
$=$ **if** $e \simeq 0$ **then** 1
else itimes $(x,$ ipower $(x,\ e-1))$ **endif**

DEFINITION:
rpower $(x,\ e)$
$=$ **if** $e \simeq 0$ **then** rational $(1,\ 1)$
else rtimes $(x,$ rpower $(x,\ e-1))$ **endif**

DEFINITION: rleq $(x,\ y) =$ (rlessp $(x,\ y) \vee$ requal $(x,\ y)$)

```
;; *****************
;;  ASCII characters
;; *****************
```

DEFINITION: ASCII_NUL $= 0$

DEFINITION: ASCII_SOH $= 1$

DEFINITION: ASCII_STX $= 2$

DEFINITION: ASCII_ETX $= 3$

DEFINITION: ASCII_EOT $= 4$

DEFINITION: ASCII_ENQ $= 5$

DEFINITION: ASCII_ACK $= 6$

DEFINITION: ASCII_BEL $= 7$

DEFINITION: ASCII_BS $= 8$

Definition: ascii_ht = 9

Definition: ascii_lf = 10

Definition: ascii_vt = 11

Definition: ascii_ff = 12

Definition: ascii_cr = 13

Definition: ascii_so = 14

Definition: ascii_si = 15

Definition: ascii_dle = 16

Definition: ascii_dc1 = 17

Definition: ascii_dc2 = 18

Definition: ascii_dc3 = 19

Definition: ascii_dc4 = 20

Definition: ascii_nak = 21

Definition: ascii_syn = 22

Definition: ascii_etb = 23

Definition: ascii_can = 24

Definition: ascii_em = 25

Definition: ascii_sub = 26

Definition: ascii_esc = 27

Definition: ascii_fs = 28

Definition: ascii_gs = 29

Definition: ascii_rs = 30

Definition: ascii_us = 31

Definition: ascii_space = 32

Definition: ascii_exclamation_point = 33

Definition: ascii_double_quote = 34

Definition: ascii_number_sign = 35

Definition: ascii_dollar = 36

Definition: ascii_percent = 37

Definition: ascii_and = 38

Definition: ascii_single_quote = 39

Definition: ascii_open_paren = 40

Definition: ascii_close_paren = 41

Definition: ascii_star = 42

Definition: ascii_plus = 43

Definition: ascii_comma = 44

Definition: ascii_dash = 45

Definition: ascii_dot = 46

Definition: ascii_slash = 47

Definition: ascii_0 = 48

Definition: ascii_1 = 49

Definition: ascii_2 = 50

Definition: ascii_3 = 51

Definition: ascii_4 = 52

Definition: ascii_5 = 53

Definition: ascii_6 = 54

Definition: ascii_7 = 55

Definition: ascii_8 = 56

Definition: ascii_9 = 57

Definition: ascii_colon = 58

Definition: ascii_semicolon = 59

Definition: ascii_lt = 60

Definition: ascii_equal = 61

Definition: ascii_gt = 62

Definition: ascii_question = 63

Definition: ascii_at = 64

Definition: ascii_a = 65

Definition: ascii_b = 66

Definition: ascii_c = 67

Definition: ascii_d = 68

Definition: ascii_e = 69

Definition: ascii_f = 70

Definition: ascii_g = 71

Definition: ascii_h = 72

Definition: ascii_i = 73

Definition: ascii_j = 74

Definition: ascii_k = 75

Definition: ascii_l = 76

Definition: ascii_m = 77

Definition: ascii_n = 78

Definition: ascii_o = 79

Definition: ascii_p = 80

Definition: ascii_q = 81

Definition: ascii_r = 82

Definition: ascii_s = 83

Definition: ascii_t = 84

Definition: ascii_u = 85

Definition: ascii_v = 86

Definition: ascii_w = 87

Definition: ascii_x = 88

Definition: ascii_y = 89

Definition: ascii_z = 90

Definition: ascii_open_bracket = 91

Definition: ascii_backslash = 92

Definition: ascii_close_bracket = 93

Definition: ascii_caret = 94

Definition: ascii_underscore = 95

Definition: ascii_back_quote = 96

Definition: ascii_lc_a = 97

Definition: ascii_lc_b = 98

Definition: ascii_lc_c = 99

Definition: ascii_lc_d = 100

Definition: ascii_lc_e = 101

Definition: ascii_lc_f = 102

Definition: ascii_lc_g = 103

Definition: ascii_lc_h = 104

Definition: ascii_lc_i = 105

Definition: ascii_lc_j = 106

Definition: ascii_lc_k = 107

Definition: ascii_lc_l = 108

DEFINITION: ASCII_LC_M = 109

DEFINITION: ASCII_LC_N = 110

DEFINITION: ASCII_LC_O = 111

DEFINITION: ASCII_LC_P = 112

DEFINITION: ASCII_LC_Q = 113

DEFINITION: ASCII_LC_R = 114

DEFINITION: ASCII_LC_S = 115

DEFINITION: ASCII_LC_T = 116

DEFINITION: ASCII_LC_U = 117

DEFINITION: ASCII_LC_V = 118

DEFINITION: ASCII_LC_W = 119

DEFINITION: ASCII_LC_X = 120

DEFINITION: ASCII_LC_Y = 121

DEFINITION: ASCII_LC_Z = 122

DEFINITION: ASCII_OPEN_BRACE = 123

DEFINITION: ASCII_VERTICAL_BAR = 124

DEFINITION: ASCII_CLOSE_BRACE = 125

DEFINITION: ASCII_TILDE = 126

DEFINITION: ASCII_DEL = 127

```
;; ***********
;;  Utilities
;; ***********

; --------------------------------
;  ASCII character (list) utilities
; --------------------------------
```

DEFINITION:
ascii_characterp $(c) = ((c \in \mathbf{N}) \wedge (0 \leq c) \wedge (c \leq 127))$

DEFINITION:
ascii_character_listp (x)
= **if** $x \simeq$ **nil** **then** $x = 0$
else ascii_characterp (car (x))
$\wedge$ ascii_character_listp (cdr (x)) **endif**

DEFINITION:
is_digit (x) = (($x \in \mathbf{N}$) $\wedge$ (ASCII_0 $\leq x$) $\wedge$ ($x \leq$ ASCII_9))

DEFINITION:
is_letter (x)
= (($x \in \mathbf{N}$)
$\wedge$ (((ASCII_A $\leq x$) $\wedge$ ($x \leq$ ASCII_Z))
$\vee$ ((ASCII_LC_A $\leq x$) $\wedge$ ($x \leq$ ASCII_LC_Z))))

DEFINITION:
printable_char_ordp (x)
= (($x \in \mathbf{N}$) $\wedge$ (ASCII_SPACE $\leq x$) $\wedge$ ($x \leq$ ASCII_TILDE))

DEFINITION:
upper_case (c)
= **if** (ASCII_LC_A $\leq c$) $\wedge$ ($c \leq$ ASCII_LC_Z) **then** $c - 32$
else c **endif**

DEFINITION:
uc_list (u)
= **if** $u \simeq$ **nil** **then** u
else cons (upper_case (car (u)), uc_list (cdr (u))) **endif**

```
; ------------------
;  Number Utilities
; ------------------
```

DEFINITION:
number_list2 (hi, lo, nl)
= **if** $hi \simeq 0$ **then** cons (lo, nl)
else number_list2 ($hi - 1$, lo, cons (iplus (hi, lo), nl)) **endif**

DEFINITION:
number_list (lo, hi)
= **if** ileq (lo, hi) **then** number_list2 (idifference (hi, lo), lo, **nil**)
else nil endif

DEFINITION:
number_to_char_list (x)
= **if** $x \in \mathbf{N}$
then let q **be** $x \div 10$,
r **be** cons (((x **mod** 10) + ASCII_0, 0)
in
if $q \simeq 0$ **then** r
else append (number_to_char_list (q), r) **endif endlet**
else nil endif

```
; ----------------
;  List Utilities
; ----------------
```

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

DEFINITION:
length (l)
= **if** $l \simeq$ **nil then** ZERO
else 1 + length (cdr (l)) **endif**

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

DEFINITION:
nth (i, s)
= **if** $s \simeq$ **nil then nil**
elseif $i = 1$ **then** car (s)
else nth ($i - 1$, cdr (s)) **endif**

DEFINITION:
rcar (x)
= **if** $x \simeq$ **nil then nil**
elseif cdr (x) $\simeq$ **nil then** car (x)
else rcar (cdr (x)) **endif**

DEFINITION:
rcdr (x)

= **if** $x \simeq$ **nil then** x
elseif cdr $(x) \simeq$ **nil then** cdr (x)
else cons (car (x), rcdr (cdr (x))) **endif**

THEOREM: lessp_rcdr
listp $(s) \rightarrow$ (count (rcdr (s)) $<$ count (s))

DEFINITION:
rcons (x, y)
= **if** $x \simeq$ **nil then** cons (y, x)
else cons (car (x), rcons (cdr (x), y)) **endif**

THEOREM: rcar_rcons
rcar (rcons (x, y)) $= y$

THEOREM: rcdr_rcons
rcdr (rcons (x, y)) $= x$

DEFINITION:
remove (e, s)
= **if** $s \simeq$ **nil then** s
elseif car $(s) = e$ **then** remove $(e,$ cdr $(s))$
else cons (car (s), remove $(e,$ cdr $(s))$) **endif**

THEOREM: lessp_remove_length
$(e \in s) \rightarrow$ (length (remove (e, s)) $<$ length (s))

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

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

DEFINITION: set_equal (x, y) = (subsetp $(x, y) \wedge$ subsetp (y, x))

```
; ****************
;  Key-Value Maps
; ****************
```

DEFINITION: EMPTY_MAP = **nil**

DEFINITION: map_entry $(k,\ v)$ = cons $(k,\ v)$

DEFINITION: in_map $(m,\ k)$ = assoc $(k,\ m)$

DEFINITION:
add_to_map $(m,\ k,\ v)$
= **if** $m \simeq$ **nil then** list (map_entry $(k,\ v)$)
elseif caar $(m) = k$ **then** cons (map_entry $(k,\ v)$, cdr (m))
else cons (car (m), add_to_map (cdr (m), k, v)) **endif**

DEFINITION:
mapped_value $(m,\ k)$
= **let** v **be** assoc $(k,\ m)$
in
if listp (v) **then** cdr (v)
else nil endif endlet

DEFINITION:
all_matches $(k,\ m)$
= **if** $m \simeq$ **nil then nil**
elseif $k =$ caar (m) **then** cons (cdar (m), all_matches $(k,$ cdr $(m)))$
else all_matches $(k,$ cdr $(m))$ **endif**

DEFINITION:
keys (x)
= **if** $x \simeq$ **nil then nil**
else cons (caar (x), keys (cdr $(x)))$ **endif**

DEFINITION:
key_values (x)
= **if** $x \simeq$ **nil then nil**
else cons (cdar (x), key_values (cdr $(x)))$ **endif**

DEFINITION:
key_value_mapp (m)
= **if** $m \simeq$ **nil then** $m =$ **nil**
else listp (car $(m)) \wedge$ key_value_mapp (cdr $(m))$ **endif**

THEOREM: lessp_keys
(key_value_mapp $(m) \wedge$ listp $(m)) \rightarrow$ (count (keys $(m)) <$ count $(m))$

THEOREM: lessp_key_values
(key_value_mapp $(m) \wedge$ listp $(m)) \rightarrow$ (count (key_values $(m)) <$ count $(m))$

DEFINITION:
pair_list_map $(x,\ y)$
= **if** $x \simeq$ **nil then nil**
else cons (map_entry (car (x), car $(y))$, pair_list_map (cdr (x), cdr $(y)))$ **endif**

```
; *******
;  Trees
; *******
```

EVENT: Add the shell *mk_tree*, with recognizer function symbol *treep* and 2 accessors: *root*, with type restriction (none-of) and default value false; *subtrees*, with type restriction (none-of) and default value false.

```
; ****************************
;  Productions and Parse Trees
; ****************************

; A grammar symbol is a litatom, representing either a terminal or a
; nonterminal in the grammar.  A terminal grammar symbol is called a token.

; (prodn x y) is the representation of "x ::= y" and
; (lhs (prodn x y)) = x & (rhs (prodn x y)) = y.
;
; The left-hand-side (lhs) of a production (prodn) is a nonterminal grammar
; symbol.  The right-hand-side (rhs) of a production is either a grammar
; symbol or a list of grammar symbols.  The grammar symbols in a production
; may be tagged (see below).
```

EVENT: Add the shell *prodn*, with recognizer function symbol *prodnp* and 2 accessors: *lhs*, with type restriction (none-of) and default value false; *rhs*, with type restriction (none-of) and default value false.

```
; (tag grammar_symbol the_tag) constructs a tagged grammar symbol.
; (gsymbol (tag grammar_symbol the_tag)) = grammar_symbol and
; (label (tag grammar_symbol the_tag)) = the_tag.
;
; Grammar symbols in productions are tagged so that semantic functions can be
; converted mechanically to report form, where the tags are thought to clarify
; the presentation.
```

EVENT: Add the shell *tag*, with recognizer function symbol *taggedp* and 2 accessors: *gsymbol*, with type restriction (none-of) and default value false; *label*, with type restriction (none-of) and default value false.

```
; (mk_tree r s) is the representation of a parse tree and
```

```
; (root (mk_tree r s)) = r & (subtrees (mk_tree r s)) = s.
;
; The root of a parse tree is a grammar symbol, representing a terminal or a
; nonterminal.  Grammar symbols in parse trees are not tagged.  If the root
; represents a terminal, the parse tree is a leaf and its subtrees part is a
; representation of the lexeme that matched the terminal grammar symbol
; (token).  Otherwise, the root represents a nonterminal, and the subtrees
; part is a parse tree or a list of parse trees.



; ********************
;   Parse Tree Leaves
; ********************

; A leaf of a parse tree is a parse tree whose root is a terminal grammar
; symbol (token) and whose subtrees part is the list of characters (lexeme)
; that matched the token.  The last cdr of the character list should be zero,
; not nil.
```

DEFINITION:
RESERVED_WORDS
=
```
'(adjoin all and append array assert assume await
  before begin behind binary block buffer case cblock
  centry cexit cobegin cond const decimal difference
  div each element elif else end entry eq exit extends
  fi from function ge give gt hex hold if iff imp input
  in into initially intersect is keep le leave lemma
  loop lt mapomit mapping mod move name ne new normal
  not octal of omit on or otherwise output pending
  procedure prove receive record remove scope send seq
  seqomit sequence set signal some sub then to type
  union unless var when with alias export import
  multiplecond none space string value)
```

DEFINITION:
SPECIAL_SYMBOL_MAP
=
```
list (cons ('and, cons (ASCII_AND, 0)),
      cons ('append, cons (ASCII_AT, 0)),
      cons ('close_paren, cons (ASCII_CLOSE_PAREN, 0)),
      cons ('colon, cons (ASCII_COLON, 0)),
      cons ('colon_equal, cons (ASCII_COLON, cons (ASCII_EQUAL, 0))),
```

cons('colon_gt, cons(ASCII_COLON, cons(ASCII_GT, 0))),
cons('comma, cons(ASCII_COMMA, 0)),
cons('dot, cons(ASCII_DOT, 0)),
cons('dot_dot, cons(ASCII_DOT, cons(ASCII_DOT, 0))),
cons('equal, cons(ASCII_EQUAL, 0)),
cons('gt, cons(ASCII_GT, 0)),
cons('imp, cons(ASCII_DASH, cons(ASCII_GT, 0))),
cons('lt, cons(ASCII_LT, 0)),
cons('lt_colon, cons(ASCII_LT, cons(ASCII_COLON, 0))),
cons('minus, cons(ASCII_DASH, 0)),
cons('open_paren, cons(ASCII_OPEN_PAREN, 0)),
cons('plus, cons(ASCII_PLUS, 0)),
cons('semi_colon, cons(ASCII_SEMICOLON, 0)),
cons('slash, cons(ASCII_SLASH, 0)),
cons('star, cons(ASCII_STAR, 0)),
cons('star_star, cons(ASCII_STAR, cons(ASCII_STAR, 0))))

DEFINITION: special_symbols = keys(special_symbol_map)

DEFINITION:
tokens
= append('(digit_list identifier character_value
string_value entry_value),
reserved_words ∪ special_symbols)

DEFINITION: tokenp(x) = (x ∈ tokens)

DEFINITION:
leafp(x)
= (treep(x) ∧ tokenp(root(x)) ∧ ascii_character_listp(subtrees(x)))

DEFINITION:
lexeme(x)
= **if** leafp(x) **then** subtrees(x)
else nil endif

```
; ========================
;  Recognizers for Lexemes
; ========================

; ----------------------
;  Reserved Word Lexeme
; ----------------------
```

DEFINITION:
reserved_word_lexemep (x) = (pack (uc_list (x))) $\in$ RESERVED_WORDS)

```
; ------------------------
;  Special Symbol Lexeme
; ------------------------
```

DEFINITION: CLOSE_BRACKET_LEXEME = cons (ASCII_CLOSE_BRACKET, 0)

DEFINITION: CLOSE_PAREN_LEXEME = cons (ASCII_CLOSE_PAREN, 0)

DEFINITION: OPEN_BRACKET_LEXEME = cons (ASCII_OPEN_BRACKET, 0)

DEFINITION: OPEN_PAREN_LEXEME = cons (ASCII_OPEN_PAREN, 0)

DEFINITION:
SPECIAL_SYMBOL_LEXEMES
= append (list (OPEN_BRACKET_LEXEME, CLOSE_BRACKET_LEXEME),
key_values (SPECIAL_SYMBOL_MAP))

DEFINITION:
special_symbol_lexemep (x) = ($x \in$ SPECIAL_SYMBOL_LEXEMES)

```
; ------------------------
;  Character Value Lexeme
; ------------------------
```

DEFINITION:
character_value_lexemep (x)
= ((length (x) = 3)
$\wedge$ (car (x) = ASCII_SINGLE_QUOTE)
$\wedge$ printable_char_ordp (cadr (x)))
$\wedge$ (rcar (x) = ASCII_SINGLE_QUOTE)
$\wedge$ (cdddr (x) = 0))

```
; -------------------
;  Digit List Lexeme
; -------------------
```

DEFINITION:
is_hexdigit (d)
= (is_digit (d)
$\vee$ ((ASCII_A $\leq$ upper_case (d)) $\wedge$ (upper_case (d) $\leq$ ASCII_F)))

DEFINITION:
is_hexdigit_list (x)
= **if** $x \simeq$ **nil** **then** $x = 0$
else is_hexdigit (car (x)) $\wedge$ is_hexdigit_list (cdr (x)) **endif**

DEFINITION:
digit_list_lexemep (x)
= (listp (x) $\wedge$ is_digit (car (x)) $\wedge$ is_hexdigit_list (cdr (x)))

```
; ------------------------------------
;  Entry Value and Identifier Lexemes
; ------------------------------------
```

DEFINITION:
identifier_lexeme_form (x)
= **if** $x \simeq$ **nil** **then** $x = 0$
elseif car (x) = ASCII_UNDERSCORE
then listp (cdr (x))
$\wedge$ (cadr (x) $\neq$ ASCII_UNDERSCORE)
$\wedge$ identifier_lexeme_form (cdr (x))
else (is_letter (car (x)) $\vee$ is_digit (car (x)))
$\wedge$ identifier_lexeme_form (cdr (x)) **endif**

DEFINITION:
identifier_lexemep (x)
= **if** listp (x)
then is_letter (car (x))
$\wedge$ identifier_lexeme_form (x)
$\wedge$ (pack (uc_list (x)) $\notin$ RESERVED_WORDS)
else f endif

DEFINITION:
entry_value_lexemep (x)
= (listp (x)
$\wedge$ identifier_lexemep (rcdr (x))
$\wedge$ (rcar (x) = ASCII_SINGLE_QUOTE))

```
; ---------------------
;  String Value Lexeme
; ---------------------
```

DEFINITION:
string_char_listp (s)

```
=   if s ≃ nil then s = 0
    elseif car (s) = ASCII_DOUBLE_QUOTE
    then listp (cdr (s))
           ∧   (cadr (s) = ASCII_DOUBLE_QUOTE)
           ∧   string_char_listp (cddr (s))
    else ascii_characterp (car (s)) ∧ string_char_listp (cdr (s)) endif

DEFINITION:
string_value_lexemep (x)
=   ((length (x) ≥ 2)
     ∧   (car (x) = ASCII_DOUBLE_QUOTE)
     ∧   (rcar (x) = ASCII_DOUBLE_QUOTE)
     ∧   string_char_listp (rcdr (cdr (x))))

; ==================================
;  Recognizers for Parse Tree Leaves
; ==================================


DEFINITION:
reserved_wordp (x)
=   (leafp (x)
     ∧   (root (x) ∈ RESERVED_WORDS)
     ∧   (uc_list (lexeme (x)) = uc_list (unpack (root (x)))))

DEFINITION:
special_symbolp (x)
=   (leafp (x)
     ∧   (root (x) ∈ SPECIAL_SYMBOLS)
     ∧   case on root (x):
         case = open_paren
         then lexeme (x)
                ∈   list (OPEN_PAREN_LEXEME, OPEN_BRACKET_LEXEME)
         case = close_paren
          then lexeme (x)
                 ∈   list (CLOSE_PAREN_LEXEME, CLOSE_BRACKET_LEXEME)
         otherwise lexeme (x)
                     =   mapped_value (SPECIAL_SYMBOL_MAP, root (x)) endcase)

DEFINITION:
character_valuep (x)
=   (leafp (x)
     ∧   (root (x) = 'character_value)
     ∧   character_value_lexemep (lexeme (x)))
```

DEFINITION:
digit_listp (x)
= (leafp (x)
  $\wedge$ (root (x) = 'digit_list)
  $\wedge$ digit_list_lexemep (lexeme (x)))

DEFINITION:
entry_valuep (x)
= (leafp (x)
  $\wedge$ (root (x) = 'entry_value)
  $\wedge$ entry_value_lexemep (lexeme (x)))

DEFINITION:
identifierp (x)
= (leafp (x)
  $\wedge$ (root (x) = 'identifier)
  $\wedge$ identifier_lexemep (lexeme (x)))

DEFINITION:
string_valuep (x)
= (leafp (x)
  $\wedge$ (root (x) = 'string_value)
  $\wedge$ string_value_lexemep (lexeme (x)))

```
; ********************************************
;  Relation between Parse Trees and Productions
; ********************************************

; ==================
;  The Gypsy Grammar
; ==================
```

EVENT: Introduce the function symbol *gypsy_grammar* of 0 arguments.

```
; ==========================================
;  First Rule Used in Derivation of Parse Tree
; ==========================================
```

DEFINITION:
mk_rhs (pt)
= **if** treep (pt) **then** root (pt)
  elseif listp (pt) **then** cons (mk_rhs (car (pt)), mk_rhs (cdr (pt)))
  else pt **endif**

DEFINITION:
mk_rule (pt)
= **if** treep (pt) **then** prodn (root (pt), mk_rhs (subtrees (pt)))
else nil endif

```
; ========================
;  Well-Formed Parse Trees
; ========================
```

DEFINITION:
parse_tree_leafp (pt)
= **if** leafp (pt)
then case on root (pt):
case = *character_value*
then character_valuep (pt)
case = *digit_list*
then digit_listp (pt)
case = *entry_value*
then entry_valuep (pt)
case = *identifier*
then identifierp (pt)
case = *string_value*
then string_valuep (pt)
otherwise reserved_wordp (pt) $\vee$ special_symbolp (pt) **endcase**
else f endif

EVENT: Disable tokenp.

EVENT: Disable parse_tree_leafp.

EVENT: Disable mk_rule.

```
#|
(do-mutual '(

(defn parse_treep (pt)
  (if (treep pt)
      (if (tokenp (root pt))
  (parse_tree_leafp pt)
(and (litatom (root pt))
     (member (mk_rule pt) (Gypsy_grammar))
     (if (treep (subtrees pt))
```

```
 (parse_treep (subtrees pt))
       (parse_tree_listp (subtrees pt)))))
       ; & (correct_precedence_rule_application pt)
    F)
  ( (lessp (count pt)) ))

(defn parse_tree_listp (sts)
  (if (listp sts)
      (and (parse_treep (car sts))
   (parse_tree_listp (cdr sts)))
    (equal sts nil))
  ( (lessp (count sts)) ))

))
|#
```

DEFINITION:
mutual-parse_tree_listp-parse_treep (*mutual-flg*, *pt*, *sts*)
= **if** *mutual-flg* = 'parse_tree_listp
then if listp (*sts*)
then mutual-parse_tree_listp-parse_treep ('parse_treep,
car (*sts*),
t)
∧ mutual-parse_tree_listp-parse_treep ('parse_tree_listp,
t,
cdr (*sts*))
else *sts* = **nil endif**
elseif treep (*pt*)
then if tokenp (root (*pt*)) **then** parse_tree_leafp (*pt*)
else litatom (root (*pt*))
∧ (mk_rule (*pt*) ∈ GYPSY_GRAMMAR)
∧ **if** treep (subtrees (*pt*))
then mutual-parse_tree_listp-parse_treep ('parse_treep,
subtrees (*pt*),
t)
else mutual-parse_tree_listp-parse_treep ('parse_tree_listp,
t,
subtrees (*pt*)) **endif endif**
else f endif

DEFINITION:
parse_tree_listp (*sts*)
= mutual-parse_tree_listp-parse_treep ('parse_tree_listp, **t**, *sts*)

DEFINITION:
parse_treep (pt)
= mutual-parse_tree_listp-parse_treep (**'parse_treep**, pt, **t**)

EVENT: Enable tokenp.

EVENT: Enable parse_tree_leafp.

EVENT: Enable mk_rule.

CONSERVATIVE AXIOM: pt_intro
($\neg$ parse_treep (pt (s, nt))) $\rightarrow$ (pt (s, nt) = **nil**)

Simultaneously, we introduce the new function symbol pt.

```
; ================================
;  Parse Tree/Production Matching
; ================================
```

DEFINITION:
untag (pr)
= **if** prodnp (pr) **then** prodn (untag (lhs (pr)), untag (rhs (pr)))
elseif taggedp (pr) **then** gsymbol (pr)
elseif listp (pr) **then** cons (untag (car (pr)), untag (cdr (pr)))
else pr **endif**

DEFINITION:
rule (pt, pr)
= (parse_treep (pt)
$\wedge$ (mk_rule (pt) = untag (pr))
$\wedge$ (untag (pr) $\in$ GYPSY_GRAMMAR))

```
; ********************
;  Functions on trees
; ********************

; -----------
;  Tree Size
; -----------
```

DEFINITION:
tree_size (x)

```
=   if treep (x) then 1 + tree_size (subtrees (x))
    elseif listp (x) then 1 + (tree_size (car (x)) + tree_size (cdr (x)))
    else 0 endif

; ---------------
;  Tree Equality
; ---------------
```

DEFINITION:
leaf_equal (*t1*, *t2*)
= **if** leafp (*t1*) ∧ leafp (*t2*)
then (root (*t1*) = root (*t2*))
∧ **if** root (*t1*)
∈ '(character_value string_value)
then lexeme (*t1*) = lexeme (*t2*)
else uc_list (lexeme (*t1*)) = uc_list (lexeme (*t2*)) **endif**
else f endif

DEFINITION:
tree_equal (*t1*, *t2*)
= **if** leafp (*t1*) **then** leaf_equal (*t1*, *t2*)
elseif treep (*t1*)
then treep (*t2*)
∧ (root (*t1*) = root (*t2*))
∧ tree_equal (subtrees (*t1*), subtrees (*t2*))
elseif listp (*t1*)
then listp (*t2*)
∧ tree_equal (car (*t1*), car (*t2*))
∧ tree_equal (cdr (*t1*), cdr (*t2*))
else *t1* = *t2* **endif**

```
; ----------
;  Subtrees
; ----------
```

DEFINITION:
list_subtree (*x*, *n*, *i*)
= **if** *x* ≃ **nil then nil**
elseif treep (car (*x*)) ∧ (root (car (*x*)) = *n*)
then if *i* = 1 **then** car (*x*)
else list_subtree (cdr (*x*), *n*, *i* − 1) **endif**
else list_subtree (cdr (*x*), *n*, *i*) **endif**

DEFINITION:
subtree $(x,\ n)$
$=$ **if** treep (x)
then if treep (subtrees (x))
then if root (subtrees (x)) $= n$ **then** subtrees (x)
else nil endif
else list_subtree (subtrees (x), n, 1) **endif**
else nil endif

DEFINITION:
subtree_body $(x,\ n)$
$=$ **if** treep (subtree $(x,\ n)$) **then** subtrees (subtree $(x,\ n)$)
else nil endif

DEFINITION:
subtree_i $(x,\ n,\ i)$
$=$ **if** treep (x) **then** list_subtree (subtrees (x), n, i)
else nil endif

THEOREM: lessp_list_subtree_size
(tree_size (list_subtree $(x,\ n,\ i)$) $\neq$ 0)
$\rightarrow$ (tree_size (list_subtree $(x,\ n,\ i)$) $<$ tree_size (x))

THEOREM: lessp_subtree_size
treep $(x) \rightarrow$ (tree_size (subtree $(x,\ n)$) $<$ tree_size (x))

THEOREM: lessp_subtree_i_size
treep $(x) \rightarrow$ (tree_size (subtree_i $(x,\ n,\ i)$) $<$ tree_size (x))

THEOREM: tree_size_not_zero
treep $(x) \rightarrow$ (tree_size $(x) \neq$ 0)

THEOREM: lessp_subtree_body_size
treep $(x) \rightarrow$ (tree_size (subtree_body $(x,\ n)$) $<$ tree_size (x))

THEOREM: rule_imp_treep
rule $(u,$ prodn $(x,\ y)) \rightarrow$ treep (u)

THEOREM: rule_imp_lessp_subtree_size
rule $(u,$ prodn $(x,\ y)) \rightarrow$ (tree_size (subtree $(u,\ z)$) $<$ tree_size (u))

THEOREM: rule_imp_lessp_subtree_i_size
rule $(u,$ prodn $(x,\ y)) \rightarrow$ (tree_size (subtree_i $(u,\ z,\ i)$) $<$ tree_size (u))

DEFINITION:
subtreep (*t1*, *t2*)
= **if** tree_equal (*t1*, *t2*) **then t**
elseif listp (*t2*) **then** subtreep (*t1*, car (*t2*)) ∨ subtreep (*t1*, cdr (*t2*))
elseif leafp (*t2*) **then f**
elseif treep (*t2*) **then** subtreep (*t1*, subtrees (*t2*))
else f endif

```
; -------------------
;  Tree Substitution
; -------------------
```

DEFINITION:
subst_tree (*t1*, *t2*, *t3*)
= **if** tree_equal (*t2*, *t3*) **then** *t1*
elseif listp (*t3*)
then cons (subst_tree (*t1*, *t2*, car (*t3*)), subst_tree (*t1*, *t2*, cdr (*t3*)))
elseif leafp (*t3*) **then** *t3*
elseif treep (*t3*)
then mk_tree (root (*t3*), subst_tree (*t1*, *t2*, subtrees (*t3*)))
else *t3* **endif**

```
; ******************
;  Tree Constructors
; ******************

; --------
;  Leaves
; --------
```

DEFINITION:
mk_reserved_word (*k*)
= **if** *k* ∈ RESERVED_WORDS **then** mk_tree (*k*, unpack (*k*))
else nil endif

DEFINITION:
special_symbol_lexeme (*x*)
= **if** in_map (SPECIAL_SYMBOL_MAP, *x*)
then mapped_value (SPECIAL_SYMBOL_MAP, *x*)
else nil endif

DEFINITION:
mk_special_symbol (*x*)

```
=   let r be special_symbol_lexeme(x)
    in
    if listp(r) then mk_tree(x, r)
    else nil endif endlet
```

DEFINITION:

```
mk_unary_operator(x)
=   if x ∈ RESERVED_WORDS
    then mk_tree('unary_operator, mk_reserved_word(x))
    else mk_tree('unary_operator, mk_special_symbol(x)) endif
```

DEFINITION:

```
mk_digit_list(e)
=   if digit_listp(e) then e
    elseif e ∈ N then mk_tree('digit_list, number_to_char_list(e))
    else mk_tree('digit_list, e) endif

; Check e?
```

DEFINITION:

```
mk_identifier_lexeme(id)
=   let n be if litatom(id) then unpack(id)
             else id endif
    in
    if n = unpack('nil~) then unpack(nil)
    elseif identifier_lexemep(n) then n
    else nil endif endlet
```

DEFINITION:

```
mk_identifier(id)
=   if root(id) = 'identifier then id
    else let n be mk_identifier_lexeme(id)
         in
         if identifier_lexemep(n)
         then mk_tree('identifier, n)
         else nil endif endlet endif
```

DEFINITION:

```
mk_entry_value_lexeme(id)
=   let n1 be if litatom(id) then unpack(id)
              else id endif
    in
    let n2 be if rcar(n1) = ASCII_SINGLE_QUOTE
              then mk_identifier_lexeme(rcdr(n1))
```

```
            else mk_identifier_lexeme(n1) endif
    in
    if identifier_lexemep(n2)
    then rcons(n2, ASCII_SINGLE_QUOTE)
    else nil endif endlet endlet

DEFINITION:
mk_entry_value(id)
=   if root(id) = 'entry_value then id
    elseif root(id) = 'identifier
    then mk_tree('entry_value, rcons(lexeme(id), ASCII_SINGLE_QUOTE))
    else let n be mk_entry_value_lexeme(id)
         in
         if entry_value_lexemep(n)
         then mk_tree('entry_value, n)
         else nil endif endlet endif

; need:  CHARACTER_VALUE STRING_VALUE


; ------------
;  Non-Leaves
; ------------


DEFINITION:  MK_EMPTY = mk_tree('empty, nil)

; Some (all?) of the following need work.


DEFINITION:
mk_number(e)
=   if e ∈ N then mk_tree('number, mk_digit_list(e))
    else mk_tree('number, e) endif

DEFINITION:
mk_literal_value(e)
=   if e ∈ N then mk_tree('literal_value, mk_number(e))
    else mk_tree('literal_value, e) endif

DEFINITION:
mk_primary_value(e)
=   if e ∈ N then mk_tree('primary_value, mk_literal_value(e))
    elseif litatom(e) then mk_tree('primary_value, mk_identifier(e))
    else mk_tree('primary_value, e) endif
```

DEFINITION:
mk_modified_primary_value (e)
= mk_tree ('`modified_primary_value`, mk_primary_value (e))

DEFINITION:
mk_expression (e)
= **if** ($e \in \mathbf{N}$) $\vee$ litatom (e)
then mk_tree ('`expression`, mk_modified_primary_value (e))
else mk_tree ('`expression`, e) **endif**

DEFINITION:
mk_scalar_const_unit (n, v, q)
= mk_tree ('`constant_declaration`,
list (mk_reserved_word ('`const`),
mk_identifier (v),
mk_special_symbol ('`colon`),
mk_tree ('`type_specification`, n),
mk_special_symbol ('`colon_equal`),
mk_tree ('`constant_body`, mk_expression (q))))

DEFINITION:
mk_named_unit (u, s)
= mk_tree ('`name_declaration`,
list (mk_reserved_word ('`name`),
mk_tree ('`local_aliases`, u),
mk_reserved_word ('`from`),
s))

DEFINITION:
mk_single_formal_data_parameter (a, p, ft)
= mk_tree ('`similar_formal_data_parameters`,
list (mk_tree ('`opt_access_specification`,
mk_reserved_word (a)),
mk_tree ('`identifier_list`, p),
mk_special_symbol ('`colon`),
ft))

DEFINITION:
mk_arg_list (x)
= mk_tree ('`arg_list`,
list (mk_special_symbol ('`open_paren`),
x,
mk_special_symbol ('`close_paren`)))

DEFINITION:

```
mk_actual(n)
=   mk_tree('expression,
            mk_tree('modified_primary_value,
                    mk_tree('primary_value, mk_identifier(n))))
```

DEFINITION:

```
mk_actual_list(as, n)
=   if as = nil then mk_tree('value_list, mk_actual(n))
    else mk_tree('value_list,
                 list(as, mk_special_symbol('comma), mk_actual(n))) endif
```

DEFINITION:

```
namelist_to_actuals(ns, as)
=   if ns ≃ nil
    then if as = nil then as
         else mk_arg_list(as) endif
    else namelist_to_actuals(cdr(ns), mk_actual_list(as, car(ns))) endif
```

DEFINITION:

```
mk_component_selectors(x) = mk_tree('component_selectors, x)
```

DEFINITION:

```
mk_value_modifiers(x)
=   if root(x) = 'arg_list
    then mk_tree('value_modifiers, mk_component_selectors(x))
    else mk_tree('value_modifiers, x) endif
```

DEFINITION:

```
MK_TRUE_EXPRESSION
=   mk_tree('expression,
            mk_tree('modified_primary_value,
                    mk_primary_value('true)))
```

DEFINITION:

```
mk_identifier_list(is, i)
=   if is = nil then mk_tree('identifier_list, i)
    else mk_tree('identifier_list,
                 list(is, mk_special_symbol('comma), i)) endif
```

DEFINITION:

```
mk_bound_expression(qn, ts, be)
=   mk_tree('bound_expression,
            list(qn,
                 mk_special_symbol('colon),
                 ts,
                 mk_special_symbol('comma),
                 be))
```

DEFINITION:
mk_quantified_expression (*qf*, *qn*, *ts*, *be*)
= mk_tree (`'expression`,
list (**if** litatom (*qf*) **then** mk_reserved_word (*qf*)
else *qf* **endif**,
mk_bound_expression (*qn*, *ts*, *be*)))

```
;; ********
;;  errors
;; ********
```

DEFINITION: mk_error (*y*) = mk_tree (`'error*`, *y*)

DEFINITION: mk_error_decl (*y*) = mk_error (*y*)

DEFINITION: errorp (*x*) = (treep (*x*) ∧ (root (*x*) = `'error*`))

DEFINITION:
error_msg (*x*)
= **if** errorp (*x*) **then** subtrees (*x*)
else `'(no error message)` **endif**

DEFINITION:
actual_formal_type_error (*a*, *ft*)
= mk_error (list (`'actual`,
`'parameter`,
a,
`'is`,
`'not`,
`'in`,
`'formal`,
`'type`,
ft))

DEFINITION:
adjoin_args_error (*v*)
= mk_error (list (`'cannot`,
`'adjoin`,
`'to`,
v,
`'because`,
`'it`,
`'is`,
`'not`,
`'a`,
`'set`))

DEFINITION:
alias_id_error (i, sn)
= mk_error_decl (cons (sn,
append (`'(is not the home scope of unit)`,
list (i))))

DEFINITION:
append_args_error ($v1$, $v2$)
= mk_error (list (`'append`,
`'is`,
`'not`,
`'defined`,
`'on`,
$v1$,
`'and`,
$v2$,
`'because`,
`'they`,
`'are`,
`'not`,
`'sequences`))

DEFINITION:
array_index_error (id)
= mk_error (list (`'array`,
`'index`,
`'type`,
id,
`'is`,
`'not`,
`'a`,
`'nonrational`,
`'simple`,
`'type`))

DEFINITION:
bad_string_error (s)
= mk_error (list (s, `'is`, `'not`, `'a`, `'well_formed`, `'string`))

DEFINITION:
bad_value_modifiers_error (x)
= mk_error (list (`'ill_formed`, `'value_modifiers`, x))

DEFINITION:
character_error (x) = mk_error (list (x, `'is`, `'not`, `'a`, `'character`))

DEFINITION:
COLON_GT_ARGS_ERROR
= mk_error ('(can colon_gt only to sequences))

DEFINITION:
component_assign_error (*x*)
= mk_error (list ('components,
'of,
'value,
x,
'cannot,
'be,
'assigned))

DEFINITION:
condition_params_error (*e*)
= mk_error (list ('expression,
e,
'should,
'not,
'have,
'actual,
'condition,
'parameters))

DEFINITION:
difference_args_error (*v1*, *v2*)
= mk_error (list ('difference,
'is,
'not,
'defined,
'on,
v1,
'and,
v2,
'because,
'they,
'are,
'neither,
'mappings,
'nor,
'sets))

DEFINITION:
DOMAIN_ARG_ERROR
= mk_error ('(domain is defined only on mappings))

DEFINITION:
duplicate_field_names_error (d)
= mk_error (list (d, 'has, 'duplicate, 'field, 'names))

DEFINITION:
duplicate_param_names_error (n)
= mk_error (list ('duplicate, 'formal, 'parameter, 'names, n))

DEFINITION:
each_id_type_error (e, c)
= mk_error (list ('the,
'bound,
'identifier,
'type,
'in,
e,
'of,
'scope,
c,
'is,
'not,
'a,
'bounded,
'index,
'type))

DEFINITION:
empty_seq_error (fn, s)
= mk_error (list (fn, 'of, 'an, 'empty, 'sequence, s))

DEFINITION:
empty_type_error (td) = mk_error (list ('type, td, 'is, 'empty))

DEFINITION:
entry_not_true_error (fn, sn)
= mk_error (list ('the,
'entry,
'spec,
'is,
'not,
'true,
'for,
'function,
fn,
'in,

```
                'scope,
                sn))
```

DEFINITION:

```
field_name_reserved_error (fds)
=   mk_error (list ('reserved,
                'identifier,
                'cannot,
                'be,
                'used,
                'as,
                'a,
                'record,
                'field,
                'name,
                'in,
                fds))
```

DEFINITION:

```
FIRST_ARG_ERROR
=   mk_error ('(first is defined only on sequences))
```

DEFINITION:

```
function_access_error (fp)
=   mk_error (list ('function,
                'formal,
                'parameter,
                'cannot,
                'be,
                'var,
                fp))
```

DEFINITION:

```
if_test_not_boolean_error (e, sn)
=   mk_error (list ('the,
                'if,
                'test,
                'of,
                'expression,
                e,
                'in,
                'scope,
                sn,
                'is,
                'not,
```

```
                'a,
                'boolean,
                'valued,
                'expression))

DEFINITION:
in_arg_error (v)
=   mk_error (list ('in,
                'is,
                'not,
                'defined,
                'on,
                v,
                'because,
                'it,
                'is,
                'neither,
                'a,
                'sequence,
                'nor,
                'a,
                'set))

DEFINITION:
indeterminate_fn_result_error (fn, sn)
=   mk_error (list ('function,
                fn,
                'of,
                'scope,
                sn,
                'returned,
                'an,
                'indeterminate,
                'value))

DEFINITION:
intersect_args_error (v1, v2)
=   mk_error (list ('intersect,
                'is,
                'not,
                'defined,
                'on,
                v1,
                'and,
                v2,
```

```
                'because,
                'they,
                'are,
                'neither,
                'mappings,
                'nor,
                'sets))
```

DEFINITION:
LAST_ARG_ERROR
= mk_error ('(last is defined only on sequences))

DEFINITION:
lower_pred_error (td)
= mk_error (list ('pred, 'of, 'lower, 'of, 'type, td))

DEFINITION:
LT_COLON_ARGS_ERROR
= mk_error ('(can lt_colon only to sequences))

DEFINITION:
many_post_conditions_error (u, c)
= mk_error (list (u,
                'has,
                'more,
                'than,
                'one,
                'exit,
                'specification,
                'for,
                'condition,
                c))

DEFINITION:
many_scope_error (sn)
= mk_error_decl (append ('(there are several scopes named),
                         list (sn)))

DEFINITION:
many_unit_error (i, sn)
= mk_error_decl (cons ('scope,
                       cons (sn,
                             append ('(has several units named),
                                     list (i)))))

DEFINITION:
mapping_merge_error (*v1*, *v2*)
= mk_error (list (`'mappings`,
v1,
`'and`,
v2,
`'have`,
`'elements`,
`'with`,
`'the`,
`'same`,
`'selector`,
`'and`,
`'different`,
`'component`,
`'values`))

DEFINITION:
mapping_selector_type_error (*sd*)
= mk_error (list (`'mapping`,
`'selector`,
`'type`,
sd,
`'is`,
`'not`,
`'an`,
`'equality`,
`'type`))

DEFINITION:
MAX_ARG_ERROR
= mk_error (`'(max is defined only on simple types)`)

DEFINITION:
MIN_ARG_ERROR
= mk_error (`'(min is defined only on simple types)`)

DEFINITION:
N_TOO_SMALL = mk_error (list (`'n`, `'is`, `'too`, `'small`))

DEFINITION:
name_already_in_use_error (*n*)
= mk_error (list (`'the`, `'name`, *n*, `'is`, `'already`, `'in`, `'use`))

DEFINITION:
NEGATIVE_EXPONENT_ERROR = mk_error (`'(negative exponent)`)

DEFINITION:
no_function_defn_error (*fn*, *sn*)
= mk_error (list (*fn*, 'in, 'scope, *sn*, 'has, 'no, 'definition))

DEFINITION:
no_scope_error (*sn*)
= mk_error_decl (append ('(there is no scope), list (*sn*)))

DEFINITION:
no_such_component_error (*s*, *i*)
= mk_error (list (*s*, 'has, 'no, 'component, *i*))

DEFINITION:
no_such_field_error (*r*, *n*) = mk_error (list (*r*, 'has, 'no, 'field, *n*))

DEFINITION:
no_unit_error (*i*, *sn*)
= mk_error_decl (list ('unit, *i*, 'is, 'not, 'in, 'scope, *sn*))

DEFINITION:
non_simple_subrange_type_error (*td*)
= mk_error (list ('subrange, 'of, 'non-simple, 'type, *td*))

DEFINITION:
NONFIRST_ARG_ERROR
= mk_error ('(nonfirst is defined only on sequences))

DEFINITION:
NONLAST_ARG_ERROR
= mk_error ('(nonlast is defined only on sequences))

DEFINITION:
not_array_error (*x*) = mk_error (list (*x*, 'is, 'not, 'an, 'array))

DEFINITION:
not_binary_op_error (*op*)
= mk_error (list (*op*, 'is, 'not, 'a, 'binary, 'operator))

DEFINITION:
not_defined_on_type_error (*op*, *td*)
= mk_error (list (*op*, 'is, 'not, 'defined, 'on, 'type, *td*))

DEFINITION:
not_equality_type_error (*td*)
= mk_error (list (*td*, 'is, 'not, 'an, 'equality, 'type))

DEFINITION:
not_expression_error (e)
= mk_error (list (e, 'is, 'not, 'an, 'expression))

DEFINITION:
not_function_or_const_error (fn, sn)
= mk_error (list (fn,
'of,
'scope,
sn,
'is,
'not,
'a,
'function,
'or,
'constant))

DEFINITION:
not_in_set_error (e, s) = mk_error (list (e, 'is, 'not, 'in, 'set, s))

DEFINITION:
not_in_type_error (v, td) = mk_error (list (v, 'is, 'not, 'in, 'type, td))

DEFINITION:
not_mapping_error (x) = mk_error (list (x, 'is, 'not, 'a, 'mapping))

DEFINITION:
not_mapping_type_error (td)
= mk_error (list (td, 'is, 'not, 'a, 'mapping, 'type))

DEFINITION:
not_range_error (r, sn)
= mk_error (list (r, 'of, 'scope, sn, 'is, 'not, 'a, 'range))

DEFINITION:
not_record_error (x) = mk_error (list (x, 'is, 'not, 'a, 'record))

DEFINITION:
not_record_fields_error (s, sn)
= mk_error (list (s,
'of,
'scope,
sn,
'is,
'not,

```
                'a,
                'parse,
                'tree,
                'for,
                'record,
                'field,
                'names))
```

DEFINITION:
not_selectable_error (x)
= mk_error (list ('components, 'of, x, 'cannot, 'be, 'selected))

DEFINITION:
not_sequence_error (s) = mk_error (list (s, 'is, 'not, 'a, 'sequence))

DEFINITION:
not_sequence_type_error (td)
= mk_error (list (td, 'is, 'not, 'a, 'sequence, 'type))

DEFINITION:
not_set_type_error (td) = mk_error (list (td, 'is, 'not, 'a, 'set, 'type))

DEFINITION:
not_type_descriptor_error (td)
= mk_error (list (td, 'is, 'not, 'a, 'type, 'descriptor))

DEFINITION:
not_type_error (tt, sn)
= mk_error (list (tt, 'of, 'scope, sn, 'is, 'not, 'a, 'type))

DEFINITION:
not_unary_op_error (op)
= mk_error (list (op, 'is, 'not, 'a, 'unary, 'operator))

DEFINITION:
null_undefined_error (td)
= mk_error (list ('null, 'is, 'not, 'defined, 'on, 'type, td))

DEFINITION:
number_error (d, b)
= mk_error (list (d, 'is, 'not, 'a, 'number, 'in, 'base, b))

DEFINITION:
omit_args_error (v)
= mk_error (list ('cannot,
                'omit,

```
                'from,
                v,
                'because,
                'it,
                'is,
                'not,
                'a,
                'set))
```

DEFINITION:
opt_default_value_error (d, sn)
= mk_error (list (d,
```
                'in,
                'scope,
                sn,
                'is,
                'not,
                'an,
                'opt_default_initial_value_expression))
```

DEFINITION:
opt_size_limit_error (r, sn)
= mk_error (list (r,
```
                'of,
                'scope,
                sn,
                'is,
                'not,
                'an,
                'opt_size_limit_restriction))
```

DEFINITION:
ORD_ARG_ERROR
= mk_error (`'(ord is defined only on scalar types)`)

DEFINITION:
param_reserved_error (n)
= mk_error (list (`'reserved`,
```
                'identifier,
                n,
                'cannot,
                'be,
                'used,
                'as,
                'a,
```

```
                'formal,
                'parameter,
                'name))
```

DEFINITION:

```
pending_default_value_error (td, sn)
=   mk_error (list ('cannot,
                'compute,
                'default,
                'value,
                'for,
                'pending,
                'type,
                td,
                'in,
                'scope,
                sn))
```

DEFINITION:

```
pending_in_type_error (td)
=   mk_error (list ('in_type, 'of, 'pending, 'type, td))
```

DEFINITION:

```
pending_type_value_set_error (td)
=   mk_error (list ('cannot,
                'compute,
                'value,
                'set,
                'of,
                'pending,
                'type,
                td))
```

DEFINITION:

```
PRED_ARG_ERROR
=   mk_error ('(pred is defined only on scalar types))
```

DEFINITION:

```
RANGE_ARG_ERROR
=   mk_error ('(range is defined only on mappings))
```

DEFINITION:

```
range_limits_error (lo, hi)
=   mk_error (list (lo,
                'and,
```

hi,
'are,
'not,
'of,
'the,
'same,
'non_rational,
'simple,
'type))

DEFINITION:
rational_value_set_error (*td*)
= mk_error (list ('cannot,
'compute,
'the,
'value,
'set,
'of,
'rational,
'type,
td))

DEFINITION:
scale_int_arg_error (*i*, *tn*)
= mk_error (list (*i*, 'cannot, 'be, 'scaled, 'in, 'type, *tn*))

DEFINITION:
scale_type_arg_error (*tn*)
= mk_error (list ('scale,
'is,
'not,
'defined,
'on,
'type,
tn,
'because,
'it,
'is,
'not,
'a,
'scalar,
'type))

DEFINITION:
scope_id_error (*sn*)

```
=   mk_error_decl (append ('(scope name),
                           cons (sn,
                                 '(cannot be used as a unit name))))
```

DEFINITION:

```
scope_reserved_error (sn)
=   mk_error_decl (list ('reserved,
                         'identifier,
                         sn,
                         'cannot,
                         'be,
                         'used,
                         'as,
                         'a,
                         'scope,
                         'name))
```

DEFINITION:

```
SIZE_ARG_ERROR
=   mk_error ('(size is defined only on mappings sequences
                sets))
```

DEFINITION:

```
size_limit_error (d, sn)
=   mk_error (list ('size,
                    'limit,
                    d,
                    'of,
                    'scope,
                    sn,
                    'is,
                    'not,
                    'a,
                    'non-negative,
                    'integer,
                    'expression))
```

DEFINITION:

```
sub_args_error (v1, v2)
=   mk_error (list ('sub,
                    'is,
                    'not,
                    'defined,
                    'on,
                    v1,
```

```
                'and,
                v2,
                'because,
                'they,
                'are,
                'not,
                'mappings,
                'or,
                'sequences,
                'or,
                'sets))
```

DEFINITION:
SUCC_ARG_ERROR
= mk_error ('(succ is defined only on scalar types))

DEFINITION:
type_defn_cycle_error (*tn*, *sn*)
= mk_error (list ('cycle,
                'in,
                'type,
                'definition,
                'on,
                tn,
                'in,
                'scope,
                sn))

DEFINITION:
type_error (*td*, *sn*)
= mk_error (list ('error, 'in, 'type, *td*, 'of, 'scope, *sn*))

DEFINITION:
UNBOUNDED_SEQUENCE_VALUE_SET_ERROR
= mk_error (list ('cannot,
                'compute,
                'the,
                'value,
                'set,
                'of,
                'a,
                'sequence,
                'without,
                'a,
                'size,

```
                'limit,
                'restriction))
```

DEFINITION:
unbounded_type_error (*td*)
= mk_error (list (`'type`, *td*, `'is`, `'not`, `'bounded`))

DEFINITION:
unbounded_value_set_error (*td*)
```
= mk_error (list ('cannot,
                  'compute,
                  'the,
                  'value,
                  'set,
                  'of,
                  'unbounded,
                  'type,
                  td))
```

DEFINITION:
union_arg_error (*v1*, *v2*)
```
= mk_error (list ('union,
                  'defined,
                  'only,
                  'on,
                  'mappings,
                  'and,
                  'sets,
                  v1,
                  v2))
```

DEFINITION:
unit_reserved_error (*u*)
```
= mk_error_decl (list ('reserved,
                       'identifier,
                       u,
                       'cannot,
                       'be,
                       'used,
                       'as,
                       'a,
                       'unit,
                       'name))
```

DEFINITION:
unknown_name_error (*n*) = mk_error (list (`'the`, `'name`, *n*, `'is`, `'unknown`))

DEFINITION:
upper_succ_error (td)
= mk_error (list (`'succ`, `'of`, `'upper`, `'of`, `'type`, td))

DEFINITION:
upper_undefined_error (td)
= mk_error (list (`'upper`, `'is`, `'not`, `'defined`, `'on`, `'type`, td))

DEFINITION:
ZERO_DIVIDE_ERROR = mk_error (`'(divide by zero)`)

DEFINITION:
ZERO_TO_THE_ZERO_POWER_ERROR
= mk_error (`'(zero to the zero power)`)

```
; ************************************
;  Semantic functions - tree extraction
; ************************************
```

EVENT: Disable rule.

EVENT: Disable subtree.

EVENT: Disable subtree_body.

EVENT: Disable subtree_i.

DEFINITION:
gname (u)
= **if** identifierp (u)
  then let n **be** uc_list (lexeme (u))
    in
    if pack (n) = **nil then** `'nil~`
    else pack (n) **endif endlet**
  else nil endif

DEFINITION:
entry_name (e)
= **if** entry_valuep (e)
  then let n **be** uc_list (lexeme (e))
    in
    if pack (rcdr (n)) = **nil**

```
        then pack (rcons (rcons (rcdr (n), ASCII_TILDE), rcar (n)))
        else pack (n) endif endlet
    else nil endif

DEFINITION:
access (a)
=   if rule (a,
            prodn (tag ('similar_formal_data_parameters, 'd),
                list (tag ('opt_access_specification, 'a),
                    tag ('identifier_list, 'is),
                    'colon,
                    tag ('type_specification, 'ft))))
    then access (subtree (a, 'opt_access_specification))
    elseif rule (a,
                prodn (tag ('internal_data_or_condition_objects,
                        'iv),
                    list (tag ('access_specification, 'a),
                        tag ('identifier_list, 'is),
                        'colon,
                        tag ('type_specification, 'ts),
                        tag ('opt_internal_initial_value, 'v),
                        'semi_colon)))
    then access (subtree (a, 'access_specification))
    elseif rule (a,
                prodn (tag ('opt_access_specification, 'a), 'empty))
    then 'const
    elseif rule (a,
                prodn (tag ('opt_access_specification, 'a),
                    tag ('access_specification, 'a2)))
    then access (subtree (a, 'access_specification))
    elseif rule (a, prodn (tag ('access_specification, 'a), 'var))
    then 'var
    elseif rule (a, prodn (tag ('access_specification, 'a), 'const))
    then 'const
    else nil endif

DEFINITION:
arg_list (e)
=   if rule (e,
            prodn (tag ('expression, 'e),
                tag ('modified_primary_value, 'm)))
    then arg_list (subtree (e, 'modified_primary_value))
    elseif rule (e,
                prodn (tag ('modified_primary_value, 'm),
```

```
                    list (tag ('modified_primary_value, 'm2),
                        tag ('value_modifiers, 'vm))))
    then arg_list (subtree (e, 'value_modifiers))
    elseif rule (e,
            prodn (tag ('modified_primary_value, 'm),
                    list (tag ('modified_primary_value, 'm2),
                        tag ('actual_condition_parameters, 'cp))))
    then arg_list (subtree (e, 'modified_primary_value))
    elseif rule (e,
            prodn (tag ('value_modifiers, 'm),
                    tag ('component_selectors, 's)))
    then arg_list (subtree (e, 'component_selectors))
    elseif rule (e,
            prodn (tag ('component_selectors, 's),
                    tag ('arg_list, 'as)))
    then subtree (e, 'arg_list)
    elseif rule (e,
            prodn (tag ('arg_list, 'as),
                    list ('open_paren,
                        tag ('value_list, 'vs),
                        'close_paren))) then e
    else nil endif
```

DEFINITION:

```
arg_listp (x)
=   if rule (x,
            prodn (tag ('value_modifiers, 'm),
                    tag ('component_selectors, 's)))
    then arg_listp (subtree (x, 'component_selectors))
    elseif rule (x,
            prodn (tag ('component_selectors, 's),
                    tag ('arg_list, 'as))) then t
    elseif rule (x,
            prodn (tag ('arg_list, 'as),
                    list ('open_paren,
                        tag ('value_list, 'vs),
                        'close_paren))) then t
    else nil endif
```

DEFINITION:

```
bound_boolean_expression (e)
=   if rule (e,
            prodn (tag ('expression, 'e),
                    list ('all, tag ('bound_expression, 'b))))
```

```
    then bound_boolean_expression (subtree (e, 'bound_expression))
    elseif rule (e,
                prodn (tag ('expression, 'e),
                       list ('some, tag ('bound_expression, 'b))))
    then bound_boolean_expression (subtree (e, 'bound_expression))
    elseif rule (e,
                prodn (tag ('bound_expression, 'b),
                       list (tag ('identifier_list, 'q),
                             'colon,
                             tag ('type_specification, 's),
                             'comma,
                             tag ('expression, 'e))))
    then subtree (e, 'expression)
    else nil endif

DEFINITION:
bound_id (e)
=   if rule (e,
            prodn (tag ('expression, 'e),
                   list ('all, tag ('bound_expression, 'b))))
    then bound_id (subtree (e, 'bound_expression))
    elseif rule (e,
                prodn (tag ('expression, 'e),
                       list ('some, tag ('bound_expression, 'b))))
    then bound_id (subtree (e, 'bound_expression))
    elseif rule (e,
                prodn (tag ('bound_expression, 'b),
                       list (tag ('identifier_list, 'q),
                             'colon,
                             tag ('type_specification, 's),
                             'comma,
                             tag ('expression, 'e))))
    then bound_id (subtree (e, 'identifier_list))
    elseif rule (e,
                prodn (tag ('identifier_list, 'is),
                       list (tag ('identifier_list, 'is2),
                             'comma,
                             tag ('identifier, 'i))))
    then bound_id (subtree (e, 'identifier_list))
    elseif rule (e,
                prodn (tag ('identifier_list, 'is),
                       tag ('identifier, 'i)))
    then bound_id (subtree (e, 'identifier))
    elseif rule (e,
```

```
            prodn (tag ('opt_each_clause, 'e),
                   list ('each,
                         tag ('identifier, 'i),
                         'colon,
                         tag ('type_specification, 'ts),
                         'comma)))
    then bound_id (subtree (e, 'identifier))
    elseif identifierp (e) then gname (e)
    else nil endif

DEFINITION:
bound_id_type (e)
=   if rule (e,
            prodn (tag ('expression, 'e),
                   list ('all, tag ('bound_expression, 'b))))
    then bound_id_type (subtree (e, 'bound_expression))
    elseif rule (e,
            prodn (tag ('expression, 'e),
                   list ('some, tag ('bound_expression, 'b))))
    then bound_id_type (subtree (e, 'bound_expression))
    elseif rule (e,
            prodn (tag ('bound_expression, 'b),
                   list (tag ('identifier_list, 'q),
                         'colon,
                         tag ('type_specification, 's),
                         'comma,
                         tag ('expression, 'e))))
    then subtree (e, 'type_specification)
    elseif rule (e,
            prodn (tag ('opt_each_clause, 'e),
                   list ('each,
                         tag ('identifier, 'i),
                         'colon,
                         tag ('type_specification, 'ts),
                         'comma)))
    then subtree (e, 'type_specification)
    else nil endif

DEFINITION:
case_exit_list2 (ls, e, c)
=   if rule (ls,
            prodn (tag ('case_exit_labels, 'ls),
                   list (tag ('case_exit_labels, 'ls2),
                         'comma,
```

```
                    tag('exit_label', 'l'))))
    then append(case_exit_list2(subtree(ls, 'case_exit_labels), e, c),
                case_exit_list2(subtree(ls, 'exit_label), e, c))
    elseif rule(ls,
            prodn(tag('case_exit_labels, 'ls),
                    tag('exit_label, 'l')))
    then case_exit_list2(subtree(ls, 'exit_label), e, c)
    elseif rule(ls, prodn(tag('exit_label, 'l), tag('identifier, 'n')))
    then case_exit_list2(subtree(ls, 'identifier), e, c)
    elseif rule(ls, prodn(tag('exit_label, 'l), 'normal))
    then if c = 'normal then list(e)
         else nil endif
    elseif identifierp(ls)
    then if gname(ls) = c then list(e)
         else nil endif
    else nil endif

DEFINITION:
case_exit_list(u, c)
=   if rule(u,
            prodn(tag('procedure_declaration, 'd),
                    list('procedure,
                        tag('identifier, 'pn),
                        tag('external_data_objects, 'a),
                        tag('opt_external_conditions, 'c),
                        'equal,
                        tag('procedure_body, 'b'))))
    then case_exit_list(subtree(u, 'procedure_body), c)
    elseif rule(u,
            prodn(tag('function_declaration, 'd),
                    list('function,
                        tag('identifier, 'fn),
                        tag('opt_external_data_objects, 'a),
                        'colon,
                        tag('type_specification, 'rt),
                        tag('opt_external_conditions, 'c),
                        'equal,
                        tag('procedure_body, 'b'))))
    then case_exit_list(subtree(u, 'procedure_body), c)
    elseif rule(u, prodn(tag('procedure_body, 'b), 'pending))
    then nil
    elseif rule(u,
            prodn(tag('procedure_body, 'b),
                    list('begin,
```

```
                    tag('external_operational_specification,
                        'es),
                    tag('opt_internal_environment, 'iv),
                    tag('opt_keep_specification, 'k),
                    tag('opt_internal_statements, 'st),
                    'end)))
then case_exit_list (subtree (u,
                              'external_operational_specification),
                     c)
elseif rule (u,
             prodn (tag ('external_operational_specification,
                         's),
                    list (tag ('opt_entry_specification, 'e),
                          tag ('opt_exit_specification, 'x))))
then case_exit_list (subtree (u, 'opt_exit_specification), c)
elseif rule (u, prodn (tag ('opt_exit_specification, 'e), 'empty))
then nil
elseif rule (u,
             prodn (tag ('opt_exit_specification, 'e),
                    list ('exit,
                          tag ('non_validated_specification_expression,
                              'se),
                          'semi_colon)))
then if c = 'normal
     then list (subtree (u,
                         'non_validated_specification_expression))
     else nil endif
elseif rule (u,
             prodn (tag ('opt_exit_specification, 'e),
                    list ('exit,
                          tag ('conditional_exit_specification,
                              'c),
                          'semi_colon)))
then case_exit_list (subtree (u,
                              'conditional_exit_specification),
                     c)
elseif rule (u,
             prodn (tag ('conditional_exit_specification, 'c),
                    list ('case,
                          'open_paren,
                          tag ('case_exit_body, 'e),
                          'close_paren)))
then case_exit_list (subtree (u, 'case_exit_body), c)
elseif rule (u,
```

```
            prodn (tag ('case_exit_body, 'b), tag ('case_exit, 'c)))
    then case_exit_list (subtree (u, 'case_exit), c)
    elseif rule (u,
            prodn (tag ('case_exit_body, 'b),
                list (tag ('case_exit_body, 'b2),
                    'semi_colon,
                    tag ('case_exit, 'c))))
    then append (case_exit_list (subtree (u, 'case_exit_body), c),
            case_exit_list (subtree (u, 'case_exit), c))
    elseif rule (u,
            prodn (tag ('case_exit, 'ce),
                list ('is,
                    tag ('case_exit_labels, 'l),
                    'colon,
                    tag ('non_validated_specification_expression,
                        'e))))
    then case_exit_list2 (subtree (u, 'case_exit_labels),
                subtree (u,
                        'non_validated_specification_expression),
                c)
    else nil endif
```

DEFINITION:

```
cdr_quantified_names (e)
=   if rule (e,
        prodn (tag ('expression, 'e),
            list ('all, tag ('bound_expression, 'b))))
    ∨   rule (e,
            prodn (tag ('expression, 'e),
                list ('some, tag ('bound_expression, 'b))))
    then cdr_quantified_names (subtree (e, 'bound_expression))
    elseif rule (e,
            prodn (tag ('bound_expression, 'b),
                list (tag ('identifier_list, 'q),
                    'colon,
                    tag ('type_specification, 's),
                    'comma,
                    tag ('expression, 'e))))
    then cdr_quantified_names (subtree (e, 'identifier_list))
    elseif rule (e,
            prodn (tag ('identifier_list, 'is),
                list (tag ('identifier_list, 'is2),
                    'comma,
                    tag ('identifier, 'i))))
```

```
    then mk_identifier_list (cdr_quantified_names (subtree (e,
                                                            'identifier_list')),
                             subtree (e, 'identifier'))
    else nil endif

DEFINITION:
quantifier (e)
=   if rule (e,
            prodn (tag ('expression, 'e),
                   list ('all, tag ('bound_expression, 'b))))
    then 'all
    elseif rule (e,
                 prodn (tag ('expression, 'e),
                        list ('some, tag ('bound_expression, 'b))))
    then 'some
    else nil endif

DEFINITION:
cdr_quantified_exp (e)
=   let qn be cdr_quantified_names (e)
    in
    if qn = nil then bound_boolean_expression (e)
    else mk_quantified_expression (quantifier (e),
                                   qn,
                                   bound_id_type (e),
                                   bound_boolean_expression (e)) endif endlet

DEFINITION:
constant_value_exp (u)
=   if rule (u,
            prodn (tag ('constant_declaration, 'd),
                   list ('const,
                         tag ('identifier, 'cn),
                         'colon,
                         tag ('type_specification, 'rt),
                         'colon_equal,
                         tag ('constant_body, 'b))))
    then constant_value_exp (subtree (u, 'constant_body))
    elseif rule (u, prodn (tag ('constant_body, 'b), 'pending))
    then nil
    elseif rule (u,
                 prodn (tag ('constant_body, 'b), tag ('expression, 'p)))
    then subtree (u, 'expression)
    else nil endif
```

```
DEFINITION:
scalar_const_units (n, u, q)
=  if rule (u,
          prodn (tag ('identifier_list, 'is), tag ('identifier, 'i)))
   then rcons (nil,
           mk_scalar_const_unit (n, gname (subtree (u, 'identifier)), q))
   elseif rule (u,
           prodn (tag ('identifier_list, 'is),
                  list (tag ('identifier_list, 'is2),
                        'comma,
                        tag ('identifier, 'i))))
   then rcons (scalar_const_units (n,
                                   subtree (u, 'identifier_list),
                                   q - 1),
           mk_scalar_const_unit (n, gname (subtree (u, 'identifier)), q))
   else nil endif

DEFINITION:
scalar_value_list (u)
=  if rule (u,
          prodn (tag ('type_declaration, 'd),
                 list ('type,
                       tag ('identifier, 'tn),
                       'equal,
                       tag ('type_definition, 'd2))))
   then scalar_value_list (subtree (u, 'type_definition))
   elseif rule (u,
           prodn (tag ('type_definition, 'd),
                  tag ('scalar_type, 's)))
   then scalar_value_list (subtree (u, 'scalar_type))
   elseif rule (u,
           prodn (tag ('scalar_type, 's),
                  list ('open_paren,
                        tag ('identifier_list, 'is),
                        'close_paren)))
   then scalar_value_list (subtree (u, 'identifier_list))
   elseif rule (u,
           prodn (tag ('identifier_list, 'is),
                  tag ('identifier, 'i)))
   then rcons (nil, gname (subtree (u, 'identifier)))
   elseif rule (u,
           prodn (tag ('identifier_list, 'is),
                  list (tag ('identifier_list, 'is2),
                        'comma,
```

```
                    tag('identifier, 'i))))
    then rcons(scalar_value_list(subtree(u, 'identifier_list)),
               gname(subtree(u, 'identifier)))
    else nil endif

DEFINITION:
derived_units(n, u)
=   if rule(u,
            prodn(tag('type_definition, 'd), tag('scalar_type, 's)))
    then derived_units(n, subtree(u, 'scalar_type))
    elseif rule(u,
                prodn(tag('scalar_type, 's),
                      list('open_paren,
                           tag('identifier_list, 'is),
                           'close_paren)))
    then scalar_const_units(n,
                            subtree(u, 'identifier_list),
                            length(scalar_value_list(u)) - 1)
    else nil endif

DEFINITION:
dparam_name(d)
=   if rule(d,
            prodn(tag('similar_formal_data_parameters, 'd),
                  list(tag('opt_access_specification, 'a),
                       tag('identifier_list, 'is),
                       'colon,
                       tag('type_specification, 'ft))))
    then dparam_name(subtree(d, 'identifier_list))
    elseif rule(d,
                prodn(tag('identifier_list, 'is),
                      tag('identifier, 'i)))
    then dparam_name(subtree(d, 'identifier))
    elseif identifierp(d) then gname(d)
    else nil endif

DEFINITION:
dparam_name_list(fs)
=   if fs ≃ nil then nil
    else cons(dparam_name(car(fs)), dparam_name_list(cdr(fs))) endif

DEFINITION:
each_clausep(e)
=   if rule(e,
            prodn(tag('opt_each_clause, 'e),
```

```
                list ('each,
                    tag ('identifier, 'i),
                    'colon,
                    tag ('type_specification, 'ts),
                    'comma))) then t
    elseif rule (e, prodn (tag ('opt_each_clause, 'e), 'empty)) then f
    else f endif
```

DEFINITION:

```
expression_from_spec (e)
=   if rule (e,
          prodn (tag ('specification_expression, 'se),
              tag ('validated_specification_expression,
                  'se2)))
      ∨   rule (e,
              prodn (tag ('specification_expression, 'se),
                  list ('open_paren,
                      tag ('validated_specification_expression,
                          'se2),
                      'close_paren)))
    then expression_from_spec (subtree (e,
                                'validated_specification_expression))
    elseif rule (e,
              prodn (tag ('validated_specification_expression,
                      'se),
                  list (tag ('non_validated_specification_expression,
                          'se2),
                      'otherwise,
                      tag ('identifier, 'i))))
    then expression_from_spec (subtree (e,
                                'non_validated_specification_expression))
    elseif rule (e,
              prodn (tag ('specification_expression, 'se),
                  tag ('non_validated_specification_expression,
                      'se2)))
    then expression_from_spec (subtree (e,
                                'non_validated_specification_expression))
    elseif rule (e,
              prodn (tag ('non_validated_specification_expression,
                      'se),
                  list ('open_paren,
                      tag ('proof_directive, 'd),
                      tag ('expression, 'e),
                      'close_paren)))
```

```
        ∨   rule (e,
                prodn (tag ('non_validated_specification_expression,
                            'se),
                       list (tag ('proof_directive, 'd),
                             tag ('expression, 'e))))
        ∨   rule (e,
                prodn (tag ('non_validated_specification_expression,
                            'se),
                       tag ('expression, 'e)))
    then subtree (e, 'expression)
    else nil endif

DEFINITION:
object_namep (m)
=   if rule (m,
            prodn (tag ('expression, 'e),
                   tag ('modified_primary_value, 'm)))
    then object_namep (subtree (m, 'modified_primary_value))
    elseif rule (m,
                prodn (tag ('modified_primary_value, 'm),
                       tag ('primary_value, 'p)))
    then object_namep (subtree (m, 'primary_value))
    elseif rule (m,
                prodn (tag ('primary_value, 'p), tag ('identifier, 'i)))
    then t
    else f endif

DEFINITION:
fn_call_formp (m)
=   if rule (m,
            prodn (tag ('expression, 'e),
                   tag ('modified_primary_value, 'm)))
    then fn_call_formp (subtree (m, 'modified_primary_value))
    elseif rule (m,
                prodn (tag ('modified_primary_value, 'm),
                       tag ('primary_value, 'p))) then object_namep (m)
    elseif rule (m,
                prodn (tag ('modified_primary_value, 'm),
                       list (tag ('modified_primary_value, 'm2),
                             tag ('value_modifiers, 'vm))))
    then object_namep (subtree (m, 'modified_primary_value))
         ∧   arg_listp (subtree (m, 'value_modifiers))
    elseif rule (m,
                prodn (tag ('modified_primary_value, 'm),
```

```
                    list (tag ('modified_primary_value, 'm2),
                        tag ('actual_condition_parameters, 'cp))))
    then fn_call_formp (subtree (m, 'modified_primary_value))
    else f endif

DEFINITION:
foreign_name (u)
=   if rule (u,
            prodn (tag ('name_declaration, 'd),
                list ('name,
                    tag ('local_aliases, 'a),
                    'from,
                    tag ('identifier, 'fs))))
    then foreign_name (subtree (u, 'local_aliases))
    elseif rule (u,
                prodn (tag ('local_aliases, 'a),
                    tag ('local_renaming, 'r)))
    then foreign_name (subtree (u, 'local_renaming))
    elseif rule (u,
                prodn (tag ('local_aliases, 'a),
                    list (tag ('local_aliases, 'a2),
                        'comma,
                        tag ('local_renaming, 'r))))
    then foreign_name (subtree (u, 'local_renaming))
    elseif rule (u,
                prodn (tag ('local_renaming, 'r),
                    tag ('identifier, 'fn)))
    then foreign_name (subtree (u, 'identifier))
    elseif rule (u,
                prodn (tag ('local_renaming, 'r),
                    list (tag ('identifier, 'ln),
                        'equal,
                        tag ('identifier, 'fn))))
    then foreign_name (subtree_i (u, 'identifier, 2))
    elseif identifierp (u)  then gname (u)
    else nil endif

DEFINITION:
foreign_scope_name (u)
=   if rule (u,
            prodn (tag ('name_declaration, 'd),
                list ('name,
                    tag ('local_aliases, 'a),
                    'from,
```

```
                    tag('identifier, 'fs))))
    then foreign_scope_name(subtree(u, 'identifier))
    elseif identifierp(u) then gname(u)
    else nil endif
```

DEFINITION:

```
full_dargs(a, p, ft)
=   if rule(p,
            prodn(tag('identifier_list, 'is), tag('identifier, 'i)))
    then rcons(nil,
               mk_single_formal_data_parameter(a,
                                               subtree(p, 'identifier),
                                               ft))
    elseif rule(p,
                prodn(tag('identifier_list, 'is),
                      list(tag('identifier_list, 'is2),
                           'comma,
                           tag('identifier, 'i))))
    then rcons(full_dargs(a, subtree(p, 'identifier_list), ft),
               mk_single_formal_data_parameter(a,
                                               subtree(p, 'identifier),
                                               ft))
    else nil endif
```

DEFINITION:

```
formal_dargs(d)
=   if rule(d,
            prodn(tag('procedure_declaration, 'd),
                  list('procedure,
                       tag('identifier, 'pn),
                       tag('external_data_objects, 'a),
                       tag('opt_external_conditions, 'c),
                       'equal,
                       tag('procedure_body, 'b))))
    then formal_dargs(subtree(d, 'external_data_objects))
    elseif rule(d,
                prodn(tag('function_declaration, 'd),
                      list('function,
                           tag('identifier, 'fn),
                           tag('opt_external_data_objects, 'a),
                           'colon,
                           tag('type_specification, 'rt),
                           tag('opt_external_conditions, 'c),
                           'equal,
```

```
                    tag('procedure_body, 'b))))
then formal_dargs(subtree(d, 'opt_external_data_objects))
elseif rule(d,
        prodn(tag('opt_external_data_objects, 'd),
              'empty)) then nil
elseif rule(d,
        prodn(tag('opt_external_data_objects, 'd),
              tag('external_data_objects, 'd2)))
then formal_dargs(subtree(d, 'external_data_objects))
elseif rule(d,
        prodn(tag('external_data_objects, 'd),
              list('open_paren,
                   tag('external_data_objects_list, 'd2),
                   'close_paren)))
then formal_dargs(subtree(d, 'external_data_objects_list))
elseif rule(d,
        prodn(tag('external_data_objects_list, 'd),
              tag('similar_formal_data_parameters, 'd2)))
then formal_dargs(subtree(d, 'similar_formal_data_parameters))
elseif rule(d,
        prodn(tag('external_data_objects_list, 'd),
              list(tag('external_data_objects_list, 'd2),
                   'semi_colon,
                   tag('similar_formal_data_parameters,
                       'd3))))
then append(formal_dargs(subtree(d,
                                 'external_data_objects_list)),
            formal_dargs(subtree(d,
                                 'similar_formal_data_parameters)))
elseif rule(d,
        prodn(tag('similar_formal_data_parameters, 'd),
              list(tag('opt_access_specification, 'a),
                   tag('identifier_list, 'is),
                   'colon,
                   tag('type_specification, 'ft))))
then full_dargs(access(subtree(d, 'opt_access_specification)),
                subtree(d, 'identifier_list),
                subtree(d, 'type_specification))
elseif rule(d,
        prodn(tag('constant_declaration, 'd),
              list('const,
                   tag('identifier, 'cn),
                   'colon,
                   tag('type_specification, 'rt),
```

```
                    'colon_equal,
                    tag('constant_body, 'b)))) then nil
    else nil endif

DEFINITION:
formal_type(d)
=   if rule(d,
            prodn(tag('similar_formal_data_parameters, 'd),
                list(tag('opt_access_specification, 'a),
                    tag('identifier_list, 'is),
                    'colon,
                    tag('type_specification, 'ft))))
    then subtree(d, 'type_specification)
    else nil endif

DEFINITION:
ibase(b)
=   if rule(b, prodn(tag('base, 'b), 'binary)) then 2
    elseif rule(b, prodn(tag('base, 'b), 'octal)) then 8
    elseif rule(b, prodn(tag('base, 'b), 'decimal)) then 10
    elseif rule(b, prodn(tag('base, 'b), 'hex)) then 16
    else nil endif

DEFINITION:
if_else_exp(e)
=   if rule(e,
            prodn(tag('if_expression, 'i),
                list('if,
                    tag('expression, 'b),
                    'then,
                    tag('expression, 'p),
                    tag('if_expression_else_part, 'e))))
    then if_else_exp(subtree(e, 'if_expression_else_part))
    elseif rule(e,
                prodn(tag('if_expression_else_part, 'e),
                    list('else, tag('expression, 'p), 'fi)))
    then subtree(e, 'expression)
    elseif rule(e,
                prodn(tag('if_expression_else_part, 'e),
                    list('elif,
                        tag('expression, 'b),
                        'then,
                        tag('expression, 'p),
                        tag('if_expression_else_part, 'e2))))
    then mk_tree('if_expression,
```

```
                cons (mk_reserved_word ('if), cdr (subtrees (e))))
    else nil endif

DEFINITION:
kind (u)
=   if rule (u,
            prodn (tag ('unit_declaration, 'd),
                tag ('type_declaration, 'd2)))
    then kind (subtree (u, 'type_declaration))
    elseif rule (u,
            prodn (tag ('type_declaration, 'd),
                list ('type,
                    tag ('identifier, 'tn),
                    'equal,
                    tag ('type_definition, 'd2)))) then 'type
    elseif rule (u,
            prodn (tag ('unit_declaration, 'd),
                tag ('procedure_declaration, 'd2)))
    then kind (subtree (u, 'procedure_declaration))
    elseif rule (u,
            prodn (tag ('procedure_declaration, 'd),
                list ('procedure,
                    tag ('identifier, 'pn),
                    tag ('external_data_objects, 'a),
                    tag ('opt_external_conditions, 'c),
                    'equal,
                    tag ('procedure_body, 'b))))
    then 'procedure
    elseif rule (u,
            prodn (tag ('unit_declaration, 'd),
                tag ('function_declaration, 'd2)))
    then kind (subtree (u, 'function_declaration))
    elseif rule (u,
            prodn (tag ('function_declaration, 'd),
                list ('function,
                    tag ('identifier, 'fn),
                    tag ('opt_external_data_objects, 'a),
                    'colon,
                    tag ('type_specification, 'rt),
                    tag ('opt_external_conditions, 'c),
                    'equal,
                    tag ('procedure_body, 'b))))
    then 'function
    elseif rule (u,
```

```
            prodn (tag ('unit_declaration, 'd),
                tag ('constant_declaration, 'd2)))
    then kind (subtree (u, 'constant_declaration))
    elseif rule (u,
            prodn (tag ('constant_declaration, 'd),
                list ('const,
                    tag ('identifier, 'cn),
                    'colon,
                    tag ('type_specification, 'rt),
                    'colon_equal,
                    tag ('constant_body, 'b)))) then 'constant
    elseif rule (u,
            prodn (tag ('unit_declaration, 'd),
                tag ('lemma_declaration, 'd2)))
    then kind (subtree (u, 'lemma_declaration))
    elseif rule (u,
            prodn (tag ('lemma_declaration, 'd),
                list ('lemma,
                    tag ('identifier, 'ln),
                    tag ('opt_external_data_objects, 'a),
                    'equal,
                    tag ('non_validated_specification_expression,
                        'b)))) then 'lemma
    elseif rule (u,
            prodn (tag ('name_declaration, 'd),
                list ('name,
                    tag ('local_aliases, 'a),
                    'from,
                    tag ('identifier, 'fs)))) then 'name
    elseif errorp (u) then 'error
    else nil endif

DEFINITION:
unit_name (u)
=   if rule (u,
        prodn (tag ('type_declaration, 'd),
            list ('type,
                tag ('identifier, 'tn),
                'equal,
                tag ('type_definition, 'd2))))
    then unit_name (subtree (u, 'identifier))
    elseif rule (u,
            prodn (tag ('procedure_declaration, 'd),
                list ('procedure,
```

```
                    tag('identifier, 'pn),
                    tag('external_data_objects, 'a),
                    tag('opt_external_conditions, 'c),
                    'equal,
                    tag('procedure_body, 'b))))
  then unit_name(subtree(u, 'identifier))
  elseif rule(u,
          prodn(tag('function_declaration, 'd),
                list('function,
                    tag('identifier, 'fn),
                    tag('opt_external_data_objects, 'a),
                    'colon,
                    tag('type_specification, 'rt),
                    tag('opt_external_conditions, 'c),
                    'equal,
                    tag('procedure_body, 'b))))
  then unit_name(subtree(u, 'identifier))
  elseif rule(u,
          prodn(tag('constant_declaration, 'd),
                list('const,
                    tag('identifier, 'cn),
                    'colon,
                    tag('type_specification, 'rt),
                    'colon_equal,
                    tag('constant_body, 'b))))
  then unit_name(subtree(u, 'identifier))
  elseif rule(u,
          prodn(tag('lemma_declaration, 'd),
                list('lemma,
                    tag('identifier, 'ln),
                    tag('opt_external_data_objects, 'a),
                    'equal,
                    tag('non_validated_specification_expression,
                        'b))))
  then unit_name(subtree(u, 'identifier))
  elseif identifierp(u) then gname(u)
  else nil endif

DEFINITION:
local_name(u)
= if rule(u,
        prodn(tag('unit_declaration, 'd),
              tag('type_declaration, 'd2)))
  then unit_name(subtree(u, 'type_declaration))
```

```
elseif rule(u,
        prodn(tag('unit_declaration, 'd),
            tag('procedure_declaration, 'd2)))
then unit_name(subtree(u, 'procedure_declaration))
elseif rule(u,
        prodn(tag('unit_declaration, 'd),
            tag('function_declaration, 'd2)))
then unit_name(subtree(u, 'function_declaration))
elseif rule(u,
        prodn(tag('unit_declaration, 'd),
            tag('constant_declaration, 'd2)))
then unit_name(subtree(u, 'constant_declaration))
elseif rule(u,
        prodn(tag('unit_declaration, 'd),
            tag('lemma_declaration, 'd2)))
then unit_name(subtree(u, 'lemma_declaration))
elseif rule(u,
        prodn(tag('name_declaration, 'd),
            list('name,
                tag('local_aliases, 'a),
                'from,
                tag('identifier, 'fs))))
then local_name(subtree(u, 'local_aliases))
elseif rule(u,
        prodn(tag('local_aliases, 'a),
            tag('local_renaming, 'r)))
then local_name(subtree(u, 'local_renaming))
elseif rule(u,
        prodn(tag('local_aliases, 'a),
            list(tag('local_aliases, 'a2),
                'comma,
                tag('local_renaming, 'r))))
then local_name(subtree(u, 'local_renaming))
elseif rule(u,
        prodn(tag('local_renaming, 'r),
            tag('identifier, 'fn)))
then local_name(subtree(u, 'identifier))
elseif rule(u,
        prodn(tag('local_renaming, 'r),
            list(tag('identifier, 'ln),
                'equal,
                tag('identifier, 'fn))))
then local_name(subtree_i(u, 'identifier, 1))
elseif identifierp(u) then gname(u)
```

```
    else unit_name (u) endif

DEFINITION:  named_unit (u, s) = mk_named_unit (u, s)

DEFINITION:
named_unit_list (u, fs)
=   if rule (u,
            prodn (tag ('local_aliases, 'a),
                   tag ('local_renaming, 'r)))
    then rcons (nil, named_unit (subtree (u, 'local_renaming), fs))
    elseif rule (u,
                 prodn (tag ('local_aliases, 'a),
                        list (tag ('local_aliases, 'a2),
                              'comma,
                              tag ('local_renaming, 'r))))
    then rcons (named_unit_list (subtree (u, 'local_aliases), fs),
                named_unit (subtree (u, 'local_renaming), fs))
    else nil endif

DEFINITION:
object_name (e)
=   if rule (e,
            prodn (tag ('expression, 'e),
                   tag ('modified_primary_value, 'm)))
    then object_name (subtree (e, 'modified_primary_value))
    elseif rule (e,
                 prodn (tag ('modified_primary_value, 'm),
                        tag ('primary_value, 'p)))
    then object_name (subtree (e, 'primary_value))
    elseif rule (e,
                 prodn (tag ('modified_primary_value, 'm),
                        list (tag ('modified_primary_value, 'm2),
                              tag ('value_modifiers, 'vm))))
    then object_name (subtree (e, 'modified_primary_value))
    elseif rule (e,
                 prodn (tag ('modified_primary_value, 'm),
                        list (tag ('modified_primary_value, 'm2),
                              tag ('actual_condition_parameters, 'cp))))
    then object_name (subtree (e, 'modified_primary_value))
    elseif rule (e,
                 prodn (tag ('primary_value, 'p), tag ('identifier, 'i)))
    then object_name (subtree (e, 'identifier))
    elseif identifierp (e)  then gname (e)
    else nil endif
```

DEFINITION:
pending_type_defnp (*d*)
= **if** rule (*d*,
prodn (tag (`'type_declaration`, `'d`),
list (`'type`,
tag (`'identifier`, `'tn`),
`'equal`,
tag (`'type_definition`, `'d2`))))
then pending_type_defnp (subtree (*d*, `'type_definition`))
elseif rule (*d*, prodn (tag (`'type_definition`, `'d`), `'pending`))
then t
else f endif

DEFINITION:
postc (*u*, *c*)
= **let** *e* **be** case_exit_list (*u*, *c*)
in
if length (*e*) = `0` **then** MK_TRUE_EXPRESSION
elseif length (*e*) = `1` **then** expression_from_spec (car (*e*))
else many_post_conditions_error (*u*, *c*) **endif endlet**

DEFINITION:
prec (*u*)
= **if** rule (*u*,
prodn (tag (`'procedure_declaration`, `'d`),
list (`'procedure`,
tag (`'identifier`, `'pn`),
tag (`'external_data_objects`, `'a`),
tag (`'opt_external_conditions`, `'c`),
`'equal`,
tag (`'procedure_body`, `'b`))))
then prec (subtree (*u*, `'procedure_body`))
elseif rule (*u*,
prodn (tag (`'function_declaration`, `'d`),
list (`'function`,
tag (`'identifier`, `'fn`),
tag (`'opt_external_data_objects`, `'a`),
`'colon`,
tag (`'type_specification`, `'rt`),
tag (`'opt_external_conditions`, `'c`),
`'equal`,
tag (`'procedure_body`, `'b`))))
then prec (subtree (*u*, `'procedure_body`))
elseif rule (*u*,

```
        prodn (tag ('constant_declaration, 'd),
                list ('const,
                    tag ('identifier, 'cn),
                    'colon,
                    tag ('type_specification, 'rt),
                    'colon_equal,
                    tag ('constant_body, 'b))))
then MK_TRUE_EXPRESSION
elseif rule (u, prodn (tag ('procedure_body, 'b), 'pending))
then MK_TRUE_EXPRESSION
elseif rule (u,
        prodn (tag ('procedure_body, 'b),
                list ('begin,
                    tag ('external_operational_specification,
                        'es),
                    tag ('opt_internal_environment, 'iv),
                    tag ('opt_keep_specification, 'k),
                    tag ('opt_internal_statements, 'st),
                    'end)))
then prec (subtree (u, 'external_operational_specification))
elseif rule (u,
        prodn (tag ('external_operational_specification,
                    's),
                list (tag ('opt_entry_specification, 'e),
                    tag ('opt_exit_specification, 'x))))
then prec (subtree (u, 'opt_entry_specification))
elseif rule (u,
        prodn (tag ('opt_entry_specification, 'e), 'empty))
then MK_TRUE_EXPRESSION
elseif rule (u,
        prodn (tag ('opt_entry_specification, 'e),
                list ('entry,
                    tag ('non_validated_specification_expression,
                        'se),
                    'semi_colon)))
then prec (subtree (u,
                'non_validated_specification_expression))
elseif rule (u,
        prodn (tag ('non_validated_specification_expression,
                    'se),
                list ('open_paren,
                    tag ('proof_directive, 'd),
                    tag ('expression, 'e),
                    'close_paren)))
```

```
        ∨   rule (u,
                prodn (tag ('non_validated_specification_expression,
                            'se),
                       list (tag ('proof_directive, 'd),
                             tag ('expression, 'e))))
        ∨   rule (u,
                prodn (tag ('non_validated_specification_expression,
                            'se),
                       tag ('expression, 'e)))
    then subtree (u, 'expression)
    else nil endif

DEFINITION:
record_field_names (d)
=   if rule (d,
            prodn (tag ('record_type, 'r),
                   list ('record,
                         'open_paren,
                         tag ('fields, 'f),
                         'close_paren)))
    then record_field_names (subtree (d, 'fields))
    elseif rule (d,
                 prodn (tag ('fields, 'f),
                        list (tag ('fields, 'f2),
                              'semi_colon,
                              tag ('similar_fields, 's))))
    then append (record_field_names (subtree (d, 'fields)),
                 record_field_names (subtree (d, 'similar_fields)))
    elseif rule (d, prodn (tag ('fields, 'f), tag ('similar_fields, 's)))
    then record_field_names (subtree (d, 'similar_fields))
    elseif rule (d,
                 prodn (tag ('similar_fields, 's),
                        list (tag ('identifier_list, 'is),
                              'colon,
                              tag ('type_specification, 'ft))))
    then record_field_names (subtree (d, 'identifier_list))
    elseif rule (d,
                 prodn (tag ('identifier_list, 'is),
                        list (tag ('identifier_list, 'is2),
                              'comma,
                              tag ('identifier, 'i))))
    then rcons (record_field_names (subtree (d, 'identifier_list)),
                record_field_names (subtree (d, 'identifier)))
    elseif rule (d,
```

```
            prodn (tag ('identifier_list, 'is),
                tag ('identifier, 'i)))
    then rcons (nil, record_field_names (subtree (d, 'identifier)))
    elseif identifierp (d) then gname (d)
    else nil endif
```

DEFINITION:

```
result_type (d)
=   if rule (d,
        prodn (tag ('function_declaration, 'd),
            list ('function,
                tag ('identifier, 'fn),
                tag ('opt_external_data_objects, 'a),
                'colon,
                tag ('type_specification, 'rt),
                tag ('opt_external_conditions, 'c),
                'equal,
                tag ('procedure_body, 'b))))
    then subtree (d, 'type_specification)
    elseif rule (d,
            prodn (tag ('constant_declaration, 'd),
                list ('const,
                    tag ('identifier, 'cn),
                    'colon,
                    tag ('type_specification, 'rt),
                    'colon_equal,
                    tag ('constant_body, 'b))))
    then subtree (d, 'type_specification)
    else nil endif
```

DEFINITION:

```
scalar_type_defnp (d)
=   if rule (d,
        prodn (tag ('type_declaration, 'd),
            list ('type,
                tag ('identifier, 'tn),
                'equal,
                tag ('type_definition, 'd2))))
    then scalar_type_defnp (subtree (d, 'type_definition))
    elseif rule (d,
            prodn (tag ('type_definition, 'd),
                tag ('scalar_type, 's))) then t
    else f endif
```

DEFINITION:

```
scope_list (u)
=   if rule (u,
            prodn (tag ('program_description, 'pd),
                   list (tag ('scope_declaration_list, 'ss),
                         'opt_semi_colon)))
    then scope_list (subtree (u, 'scope_declaration_list))
    elseif rule (u,
                 prodn (tag ('scope_declaration_list, 'ss),
                        tag ('scope_declaration, 'sd)))
    then rcons (nil, subtree (u, 'scope_declaration))
    elseif rule (u,
                 prodn (tag ('scope_declaration_list, 'ss),
                        list (tag ('scope_declaration_list, 'ss2),
                              'semi_colon,
                              tag ('scope_declaration, 'sd))))
    then rcons (scope_list (subtree (u, 'scope_declaration_list)),
                subtree (u, 'scope_declaration))
    else nil endif
```

DEFINITION:

```
scope_name (u)
=   if rule (u,
            prodn (tag ('scope_declaration, 'sd),
                   list ('scope,
                         tag ('identifier, 'sn),
                         'equal,
                         'begin,
                         tag ('unit_or_name_declaration_list, 'ul),
                         'opt_semi_colon,
                         'end)))
    then scope_name (subtree (u, 'identifier))
    elseif identifierp (u)  then gname (u)
    else nil endif
```

DEFINITION:

```
unit_list (u)
=   if rule (u,
            prodn (tag ('scope_declaration, 'sd),
                   list ('scope,
                         tag ('identifier, 'sn),
                         'equal,
                         'begin,
                         tag ('unit_or_name_declaration_list, 'ul),
                         'opt_semi_colon,
```

```
                'end)))
then unit_list (subtree (u, 'unit_or_name_declaration_list))
elseif rule (u,
        prodn (tag ('unit_or_name_declaration_list, 'us),
            tag ('unit_or_name_declaration, 'd)))
then unit_list (subtree (u, 'unit_or_name_declaration))
elseif rule (u,
        prodn (tag ('unit_or_name_declaration_list, 'us),
            list (tag ('unit_or_name_declaration_list,
                    'us2),
                'semi_colon,
                tag ('unit_or_name_declaration, 'd))))
then append (unit_list (subtree (u,
                        'unit_or_name_declaration_list)),
        unit_list (subtree (u, 'unit_or_name_declaration)))
elseif rule (u,
        prodn (tag ('unit_or_name_declaration, 'd),
            tag ('unit_declaration, 'd2)))
then unit_list (subtree (u, 'unit_declaration))
elseif rule (u,
        prodn (tag ('unit_declaration, 'd),
            tag ('type_declaration, 'd2)))
then unit_list (subtree (u, 'type_declaration))
elseif rule (u,
        prodn (tag ('type_declaration, 'd),
            list ('type,
                tag ('identifier, 'tn),
                'equal,
                tag ('type_definition, 'd2))))
then cons (u,
        derived_units (subtree (u, 'identifier),
                subtree (u, 'type_definition)))
elseif rule (u,
        prodn (tag ('unit_declaration, 'd),
            tag ('procedure_declaration, 'd2)))
then unit_list (subtree (u, 'procedure_declaration))
elseif rule (u,
        prodn (tag ('procedure_declaration, 'd),
            list ('procedure,
                tag ('identifier, 'pn),
                tag ('external_data_objects, 'a),
                tag ('opt_external_conditions, 'c),
                'equal,
                tag ('procedure_body, 'b))))
```

```
then cons (u, nil)
elseif rule (u,
            prodn (tag ('unit_declaration, 'd),
                   tag ('function_declaration, 'd2)))
then unit_list (subtree (u, 'function_declaration))
elseif rule (u,
            prodn (tag ('function_declaration, 'd),
                   list ('function,
                         tag ('identifier, 'fn),
                         tag ('opt_external_data_objects, 'a),
                         'colon,
                         tag ('type_specification, 'rt),
                         tag ('opt_external_conditions, 'c),
                         'equal,
                         tag ('procedure_body, 'b))))
then cons (u, nil)
elseif rule (u,
            prodn (tag ('unit_declaration, 'd),
                   tag ('constant_declaration, 'd2)))
then unit_list (subtree (u, 'constant_declaration))
elseif rule (u,
            prodn (tag ('constant_declaration, 'd),
                   list ('const,
                         tag ('identifier, 'cn),
                         'colon,
                         tag ('type_specification, 'rt),
                         'colon_equal,
                         tag ('constant_body, 'b))))
then cons (u, nil)
elseif rule (u,
            prodn (tag ('unit_declaration, 'd),
                   tag ('lemma_declaration, 'd2)))
then unit_list (subtree (u, 'lemma_declaration))
elseif rule (u,
            prodn (tag ('lemma_declaration, 'd),
                   list ('lemma,
                         tag ('identifier, 'ln),
                         tag ('opt_external_data_objects, 'a),
                         'equal,
                         tag ('non_validated_specification_expression,
                              'b')))) then cons (u, nil)
elseif rule (u,
            prodn (tag ('unit_or_name_declaration, 'd),
                   tag ('name_declaration, 'd2)))
```

then unit_list (subtree (*u*, 'name_declaration))
elseif rule (*u*,
prodn (tag ('name_declaration, 'd),
list ('name,
tag ('local_aliases, 'a),
'from,
tag ('identifier, 'fs))))
then named_unit_list (subtree (*u*, 'local_aliases),
subtree (*u*, 'identifier))
else nil endif

```
; **********************
;  Reserved Identifiers
; **********************

; (defn reserved_words () ...)
;   defined above under "Parse Tree Leaves"
```

DEFINITION:
STANDARD_IDS
= '(activationid boolean character integer rational true
false allfrom allto content domain empty first full
infrom infrommerge initial last lower max messages
min nonfirst nonlast null ord outto outtomerge pred
range scale size succ timedallfrom timedallto
timedinfrom timedinfrommerge timedmerge timedorder
timedoutto timedouttomerge upper routineerror
spaceerror)

DEFINITION:
reserved_idp (*n*)
= **let** *i* **be if** litatom (*n*) **then** pack (uc_list (unpack (*n*)))
else pack (uc_list (*n*)) **endif**
in
(*i* ∈ RESERVED_WORDS) ∨ (*i* ∈ STANDARD_IDS) **endlet**

DEFINITION:
some_reserved_idp (*s*)
= **if** *s* ≃ **nil then f**
else reserved_idp (car (*s*)) ∨ some_reserved_idp (cdr (*s*)) **endif**

```
;; *****************
;;  Unit references
;; *****************
```

DEFINITION:
local_unit_names (ul)
= **if** $ul \simeq$ **nil then nil**
else cons (local_name (car (ul)), local_unit_names (cdr (ul))) **endif**

DEFINITION:
local_names (sd) = cons (scope_name (sd), local_unit_names (unit_list (sd)))

DEFINITION:
all_scopes (n, s)
= **if** $s \simeq$ **nil then nil**
elseif scope_name (car (s)) = n
then cons (car (s), all_scopes (n, cdr (s)))
else all_scopes (n, cdr (s)) **endif**

DEFINITION:
all_units (n, u)
= **if** $u \simeq$ **nil then nil**
elseif local_name (car (u)) = n **then** cons (car (u), all_units (n, cdr (u)))
else all_units (n, cdr (u)) **endif**

EVENT: Disable reserved_idp.

EVENT: Disable unit_reserved_error.

EVENT: Disable scope_reserved_error.

EVENT: Disable scope_id_error.

EVENT: Disable alias_id_error.

EVENT: Disable no_scope_error.

EVENT: Disable no_unit_error.

EVENT: Disable many_unit_error.

EVENT: Disable many_scope_error.

DEFINITION:

```
mref (i, sn, x, m)
=   if reserved_idp (i) then cons (sn, unit_reserved_error (i))
    elseif reserved_idp (sn) then cons (sn, scope_reserved_error (sn))
    elseif i = sn then cons (sn, scope_id_error (sn))
    elseif fix (m) = 0 then cons (sn, alias_id_error (i, sn))
    else let slist be all_scopes (sn, scope_list (x))
        in
        if length (slist) = 0
        then cons (sn, no_scope_error (sn))
        elseif length (slist) = 1
        then let ulist be all_units (i,
                                     unit_list (car (slist)))
            in
            if length (ulist) = 0
            then cons (sn, no_unit_error (i, sn))
            elseif length (ulist) = 1
            then let u be car (ulist)
                in
                if kind (u) = 'name
                then mref (foreign_name (u),
                           foreign_scope_name (u),
                           x,
                           m − 1)
                else cons (sn, u) endif endlet
            else cons (sn,
                       many_unit_error (i, sn)) endif endlet
        else cons (sn, many_scope_error (sn)) endif endlet endif
```

EVENT: Enable reserved_idp.

EVENT: Enable unit_reserved_error.

EVENT: Enable scope_reserved_error.

EVENT: Enable scope_id_error.

EVENT: Enable alias_id_error.

EVENT: Enable no_scope_error.

EVENT: Enable no_unit_error.

EVENT: Enable many_unit_error.

EVENT: Enable many_scope_error.

DEFINITION: ref (i, sn, x) = mref $(i, sn, x, 2)$

DEFINITION: ref_unit (x) = cdr (x)

DEFINITION: ref_scope (x) = car (x)

```
; ****************
;  Marked Objects
; ****************
```

EVENT: Add the shell *marked*, with recognizer function symbol *markedp* and 2 accessors: *mark*, with type restriction (none-of) and default value false; *object*, with type restriction (none-of) and default value zero.

DEFINITION:
unmark (o)
= **if** markedp (o) **then** object (o)
else o **endif**

```
; ****************************
;  Pre-Computable Expressions
; ****************************
```

EVENT: Introduce the function symbol *precomputable_f* of 3 arguments.

```
; constraint on precomputable_F:
;   (or (indeterminate (precomputable_F e c x))
;       (equal (precomputable_F e c x)
;              (gF e c (empty_map) n x)))
; for some n.
```

```
; ****************************
;  Type Descriptor Primitives
; ****************************
```

```
; A type descriptor is a tree with different forms for integers/rationals,
; scalar types, arrays, records, mappings, sequences, and sets.  There are
; also special forms for pending types and errors.

; ========================
;  Primitive Constructors
; ========================

; --------------
;  Simple Types
; --------------
```

DEFINITION:
mk_integer_desc (*tmin*, *tmax*, *udv*)
= mk_tree ('`integer`,
list (mk_tree ('`tmin`, *tmin*),
mk_tree ('`tmax`, *tmax*),
mk_tree ('`udv`, *udv*)))

DEFINITION:
mk_rational_desc (*tmin*, *tmax*, *udv*)
= mk_tree ('`rational`,
list (mk_tree ('`tmin`, *tmin*),
mk_tree ('`tmax`, *tmax*),
mk_tree ('`udv`, *udv*)))

DEFINITION:
mk_scalar_desc (*tid*, *sid*, *crd*, *tmin*, *tmax*, *udv*)
= mk_tree ('`scalar`,
list (mk_tree ('`tid`, *tid*),
mk_tree ('`sid`, *sid*),
mk_tree ('`crd`, *crd*),
mk_tree ('`tmin`, *tmin*),
mk_tree ('`tmax`, *tmax*),
mk_tree ('`udv`, *udv*)))

```
; -------------
;  Array Types
; -------------
```

DEFINITION:
mk_array_desc (*id*, *cd*, *udv*)
= mk_tree ('`array`,

```
            list (mk_tree ('selector_td, id),
                  mk_tree ('component_td, cd),
                  mk_tree ('udv, udv)))

; --------------
;  Record Types
; --------------
```

DEFINITION:
mk_record_desc (*fds*, *udv*)
= mk_tree ('record,
            list (mk_tree ('field_tds, *fds*), mk_tree ('udv, *udv*)))

```
; ---------------
;  Mapping Types
; ---------------
```

DEFINITION:
mk_mapping_desc (*sl*, *sd*, *cd*, *udv*)
= mk_tree ('mapping,
            list (mk_tree ('max_size, *sl*),
                  mk_tree ('selector_td, *sd*),
                  mk_tree ('component_td, *cd*),
                  mk_tree ('udv, *udv*)))

```
; ----------------
;  Sequence Types
; ----------------
```

DEFINITION:
mk_sequence_desc (*sl*, *cd*, *udv*)
= mk_tree ('sequence,
            list (mk_tree ('max_size, *sl*),
                  mk_tree ('component_td, *cd*),
                  mk_tree ('udv, *udv*)))

```
; -----------
;  Set Types
; -----------
```

DEFINITION:

mk_set_desc (*sl*, *cd*, *udv*)
= mk_tree ('set,
list (mk_tree ('max_size, *sl*),
mk_tree ('component_td, *cd*),
mk_tree ('udv, *udv*)))

```
; ---------------
;  Pending Types
; ---------------
```

DEFINITION:
mk_pending_desc (*tid*, *sid*, *udv*)
= mk_tree ('pending,
list (mk_tree ('tid, *tid*),
mk_tree ('sid, *sid*),
mk_tree ('udv, *udv*)))

```
; -------------
;  Type Errors
; -------------

; The type descriptor constructor for type errors is
;
;     mk_error (<error message>)
;
; It and functions for specific type errors are defined in the section on
; errors, above.



; ======================
;  Primitive Extractors
; ======================
```

DEFINITION:
component_td (*td*) = subtree_body (*td*, 'component_td)

DEFINITION: crd (*td*) = subtree_body (*td*, 'crd)

DEFINITION: field_tds (*td*) = subtree_body (*td*, 'field_tds)

DEFINITION: field_td (*fn*, *td*) = mapped_value (field_tds (*td*), *fn*)

DEFINITION: max_size (*td*) = subtree_body (*td*, 'max_size)

DEFINITION: selector_td (*td*) = subtree_body (*td*, `'selector_td`)

DEFINITION: sid (*td*) = subtree_body (*td*, `'sid`)

DEFINITION:
tid (*td*)
= **if** root (*td*) ∈ `'(integer rational)` **then** root (*td*)
else subtree_body (*td*, `'tid`) **endif**

DEFINITION: tmax (*td*) = subtree_body (*td*, `'tmax`)

DEFINITION: tmin (*td*) = subtree_body (*td*, `'tmin`)

DEFINITION:
type_error_msg (*td*)
= **if** errorp (*td*) **then** error_msg (*td*)
else nil endif

DEFINITION:
udv (*td*)
= **if** errorp (*td*) **then** *td*
else subtree_body (*td*, `'udv`) **endif**

```
; ======================
;  Primitive Recognizers
; ======================
```

DEFINITION:
integer_descp (*td*) = (*td* = mk_integer_desc (tmin (*td*), tmax (*td*), udv (*td*)))

DEFINITION:
rational_descp (*td*) = (*td* = mk_rational_desc (tmin (*td*), tmax (*td*), udv (*td*)))

DEFINITION:
scalar_descp (*td*)
= (*td* = mk_scalar_desc (tid (*td*),
sid (*td*),
crd (*td*),
tmin (*td*),
tmax (*td*),
udv (*td*)))

DEFINITION:
simple_descp (*td*)
= (integer_descp (*td*) ∨ rational_descp (*td*) ∨ scalar_descp (*td*))

DEFINITION:
array_descp (td)
= (td = mk_array_desc (selector_td (td), component_td (td), udv (td)))

DEFINITION:
record_descp (td) = (td = mk_record_desc (field_tds (td), udv (td)))

DEFINITION:
mapping_descp (td)
= (td = mk_mapping_desc (max_size (td),
selector_td (td),
component_td (td),
udv (td)))

DEFINITION:
sequence_descp (td)
= (td = mk_sequence_desc (max_size (td), component_td (td), udv (td)))

DEFINITION:
set_descp (td)
= (td = mk_set_desc (max_size (td), component_td (td), udv (td)))

DEFINITION:
pending_descp (td) = (td = mk_pending_desc (tid (td), sid (td), udv (td)))

DEFINITION: error_descp (td) = errorp (td)

DEFINITION:
type_descp (td)
= (simple_descp (td)
∨ array_descp (td)
∨ record_descp (td)
∨ mapping_descp (td)
∨ sequence_descp (td)
∨ set_descp (td)
∨ pending_descp (td)
∨ error_descp (td))

```
; ==========================
;  Setting Descriptor Parts
; ==========================
```

DEFINITION:
set_tmax (td, v) = subst_tree (mk_tree (`'tmax`, v), subtree (td, `'tmax`), td)

DEFINITION:
set_tmin (td, v) = subst_tree (mk_tree ('tmin, v), subtree (td, 'tmin), td)

DEFINITION:
set_udv (td, v) = subst_tree (mk_tree ('udv, v), subtree (td, 'udv), td)

```
; ************************************
;  Descriptors for Standard Simple Types
; ************************************
```

DEFINITION:
BOOLEAN_DESC = mk_scalar_desc ('boolean, **nil**, 2, 0, 1, 0)

DEFINITION:
CHARACTER_DESC = mk_scalar_desc ('character, **nil**, 128, 0, 127, 0)

DEFINITION: INTEGER_DESC = mk_integer_desc (**nil**, **nil**, 0)

DEFINITION:
RATIONAL_DESC = mk_rational_desc (**nil**, **nil**, rational (0, 1))

```
; ******************
;  Type Recognizers
; ******************
```

DEFINITION: integer_typep (td) = integer_descp (td)

DEFINITION: rational_typep (td) = rational_descp (td)

DEFINITION: scalar_typep (td) = scalar_descp (td)

DEFINITION:
boolean_typep (td) = (scalar_typep (td) $\wedge$ (tid (td) = 'boolean))

DEFINITION:
character_typep (td) = (scalar_typep (td) $\wedge$ (tid (td) = 'character))

DEFINITION:
simple_typep (td)
= (integer_typep (td) $\vee$ rational_typep (td) $\vee$ scalar_typep (td))

DEFINITION:
bounded_typep (td)
= **if** simple_typep (td) **then** (tmin (td) $\neq$ **nil**) $\wedge$ (tmax (td) $\neq$ **nil**)
else f endif

DEFINITION: equality_typep (td) = type_descp (td)

DEFINITION:
non_rational_simple_typep (td)
= (simple_typep (td) $\wedge$ ($\neg$ rational_typep (td)))

DEFINITION: index_typep (td) = non_rational_simple_typep (td)

DEFINITION:
bounded_index_typep (td) = (index_typep (td) $\wedge$ bounded_typep (td))

```
; ***************************
;  Functions on Gypsy Values
; ***************************

; Gypsy values are represented as integers, rationals, selector-component
; maps, and lists.   A selector-component map is a list of pairs.  The car of
; each pair is a selector (index, field name).  The cdr is the Gypsy value of
; that component.
;
; Values of boolean, character, integer, and user-defined scalar types are all
; represented as integers.
;
; Values of rational types are represented as rationals.
;
; Values of arrays, records, and mappings are represented as
; selector-component maps from indexes, field names, and selectors,
; respectively, to components.
;
; Values of sequences and sets are represented as lists of components.
```

DEFINITION: NULL_MAP = EMPTY_MAP

DEFINITION: NULL_SEQ = **nil**

DEFINITION: NULL_SET = **nil**

DEFINITION: field_names (v) = keys (v)

DEFINITION: vselectors (v) = keys (v)

DEFINITION: vcomponents (v) = key_values (v)

DEFINITION: vdomain (v) = vselectors (v)

DEFINITION: vindexes (v) = vselectors (v)

DEFINITION: vrange (v) = vcomponents (v)

DEFINITION:
fselect $(m,\ k)$
= **if** in_map $(m,\ k)$ **then** mapped_value $(m,\ k)$
else no_such_component_error $(m,\ k)$ **endif**

```
; ================================
;  Lemmas for the Vequal Do-Mutual
; ================================
```

THEOREM: list_tree_size_not_zero
listp $(x) \rightarrow$ (tree_size $(x) \neq$ 0)

THEOREM: lessp_list_subtree_than_subtrees
listp $(x) \rightarrow$ (tree_size (list_subtree $(x,\ n,\ i)$) $<$ tree_size (x))

THEOREM: lessp_subtree_than_subtrees
listp (subtrees (x)) $\rightarrow$ (tree_size (subtree $(x,\ n)$) $<$ tree_size (subtrees (x)))

THEOREM: lessp_subtree_body_than_subtrees
listp (subtrees (x))
$\rightarrow$ (tree_size (subtree_body $(x,\ n)$) $<$ tree_size (subtrees (x)))

THEOREM: lessp_mapping_domain_tree_size_0
tree_size (mk_set_desc (**nil**, sd, **nil**))
$<$ tree_size (subtrees (mk_mapping_desc $(s,\ sd,\ cd,\ dv)$))

THEOREM: lessp_mapping_domain_tree_size
mapping_descp (x)
$\rightarrow$ (tree_size (mk_set_desc (**nil**, subtree_body $(x,$ `'selector_td`), **nil**))
$<$ tree_size (subtrees (x)))

THEOREM: listp_array_desc_subtrees
array_descp $(x) \rightarrow$ listp (subtrees (x))

THEOREM: listp_record_desc_subtrees
record_descp $(x) \rightarrow$ listp (subtrees (x))

THEOREM: listp_mapping_desc_subtrees
mapping_descp $(x) \rightarrow$ listp (subtrees (x))

THEOREM: listp_sequence_desc_subtrees
sequence_descp $(x) \rightarrow$ listp (subtrees (x))

THEOREM: listp_set_desc_subtrees
 set_descp $(x) \rightarrow$ listp (subtrees (x))

THEOREM: treep_type_desc
 type_descp $(x) \rightarrow$ treep (x)

THEOREM: lessp_cdr_tree_size
 listp $(x) \rightarrow$ (tree_size (cdr (x)) $<$ tree_size (x))

THEOREM: lessp_cdar_tree_size
 listp $(x) \rightarrow$ (tree_size (cdar (x)) $<$ tree_size (x))

```
; ======================
;  The Vequal Do-Mutual
; ======================
```

EVENT: Disable mk_set_desc.

EVENT: Disable type_descp.

EVENT: Disable array_descp.

EVENT: Disable record_descp.

EVENT: Disable mapping_descp.

EVENT: Disable sequence_descp.

EVENT: Disable set_descp.

```
#|
(do-mutual '(

(defn vmapped_value (m k td)
  ; m is a selector-component map
  ; k is a Gypsy value, the selector
  ; td is the type descriptor for k and keys of m
  ; result is m[k]
  (if (nlistp m)
      (no_such_component_error m k)
```

```
  (if (vequal (caar m) k td)
      (cdar m)
    (vmapped_value (cdr m) k td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
   (add1 (count m)))
     (count k))) ))

(defn vmapped_value_list (m ks td)
  ; m is a selector-component-map
  ; ks is a list of Gypsy values, the selectors
  ; td is the type descriptor for elements of ks and keys of m
  ; result is a list of components
  (if (nlistp ks)
      nil
    (cons (vmapped_value m (car ks) td)
  (vmapped_value_list m (cdr ks) td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
   (add1 (count m)))
     (count ks))) ))

(defn vequal_list (v1 v2 td)
  (if (nlistp v1)
      (nlistp v2)
  (if (nlistp v2)
      F
    (and (vequal (car v1) (car v2) td)
 (vequal_list (cdr v1) (cdr v2) td))))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
   (add1 (count v1)))
     (count v2))) ))

(defn varray_equal (v1 v2 td)
  ; entry (and (array_descp td)
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  ; Note: Because v1 and v2 are both in type td
  ;       (and (equal (vindexes v1) (value_set (selector_td td)))
  ;            (equal (vindexes v2) (value_set (selector_td td))))
  (if (array_descp td)
      (vequal_list (vcomponents v1)
   (vmapped_value_list v2 (vindexes v1) (selector_td td))
   (component_td td))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
```

```
   (add1 (count v1)))
     (count v2))) ))

(defn vfields_equal (v1 v2 tds)
  ; entry (and (equal (field_names v1) (field_names v2))
  ;            (subset (field_names tds) (field_names v1)))
  (if (nlistp tds)
      T
    (and (vequal (fselect v1 (caar tds))
 (fselect v2 (caar tds))
 (cdar tds))
 (vfields_equal v1 v2 (cdr tds))))
  ( (ord-lessp (cons (cons (add1 (tree_size tds))
   (add1 (count v1)))
     (count v2))) ))

(defn vrecord_equal (v1 v2 td)
  ; entry (and (record_descp td)
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  ; Note: Because v1 and v2 are both in type td
  ;       (and (equal (field_names v1) (field_names (field_tds td)))
  ;            (equal (field_names v2) (field_names (field_tds td))))
  (if (record_descp td)
      (vfields_equal v1 v2 (field_tds td))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
   (add1 (count v1)))
     (count v2))) ))

(defn vmapping_equal (v1 v2 td)
  ; entry (and (mapping_descp td)
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  (if (mapping_descp td)
      (and (vequal (vdomain v1) (vdomain v2)
   (mk_set_desc nil (selector_td td) (null_set)))
   (vequal_list (vcomponents v1)
(vmapped_value_list v2 (vdomain v1) (selector_td td))
(component_td td)))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
   (add1 (count v1)))
     (count v2))) ))
```

```
(defn vsequence_equal (v1 v2 td)
  ; entry (and (sequence_descp td)
  ;             (truep (dtype td v1))
  ;             (truep (dtype td v2)))
  (if (sequence_descp td)
      (vequal_list v1 v2 (component_td td))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
   (add1 (count v1)))
     (count v2))) ))

(defn vset_equal (v1 v2 td)
  ; entry (and (set_descp td)
  ;             (truep (dtype td v1))
  ;             (truep (dtype td v2)))
  (if (set_descp td)
      (and (vsubsetp v1 v2 (component_td td))
   (vsubsetp v2 v1 (component_td td)))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
   (add1 (count v1)))
     (count v2))) ))

(defn vequal (v1 v2 td)
  ; entry (and (type_descp td)
  ;             (truep (dtype td v1))
  ;             (truep (dtype td v2)))
  (if (type_descp td)
      (case (root td)
(array (varray_equal v1 v2 td))
(record (vrecord_equal v1 v2 td))
(mapping (vmapping_equal v1 v2 td))
(sequence (vsequence_equal v1 v2 td))
(set (vset_equal v1 v2 td))
(rational (requal v1 v2))
(otherwise ; td is a non-rational simple type
  (equal v1 v2)))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size td))
   (add1 (count v1)))
     (count v2))) ))

(defn vsubsetp (v1 v2 td)
```

```
  ; td is the type descriptor for v1's and v2's components
  (if (nlistp v1)
      T
    (and (vmember (car v1) v2 td)
 (vsubsetp (cdr v1) v2 td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
   (add1 (count v1)))
     (count v2))) ))

(defn vmember (e s td)
  ; td is the type descriptor for e and s's components
  (if (nlistp s)
      F
    (or (vequal e (car s) td)
(vmember e (cdr s) td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
   (add1 (count e)))
     (count s))) ))

))
|#
```

DEFINITION:
mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (*mutual-flg*,
e,
k,
ks,
m,
s,
td,
tds,
v1,
v2)

= **case on** *mutual-flg*:
case = *vmember*
then if *s* ≃ **nil then f**
else mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vequal**,
t,
t,
t,
t,
t,
td,

t,
e,
car (*s*))
∨ mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('vmembe
e,
t,
t,
t,
cdr (*s*),
td,
t,
t,
t) **endif**

case = *vsubsetp*
then if *v1* ≃ **nil then t**
else mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('vmember,
car (*v1*),
t,
t,
t,
v2,
td,
t,
t,
t)
∧ mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('vsubs
t,
t,
t,
t,
t,
td,
t,
cdr (*v1*
v2) **en**

case = *vequal*
then if type_descp (*td*)
then case on root (*td*):
case = *array*
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('va
t,
t,
t,
t,

t,
td,
t,
v1,
v2)
case = *record*
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vr**
t,
t,
t,
t,
t,
td,
t,
v1,
v2)
case = *mapping*
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vr**
t,
t,
t,
t,
t,
td,
t,
v1,
v2)
case = *sequence*
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vs**
t,
t,
t,
t,
t,
td,
t,
v1,
v2)
case = *set*
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vs**
t,
t,
t,
t,

t,
td,
t,
v1,
v2)

case = *rational*
then requal (*v1*, *v2*)
otherwise *v1* = *v2* **endcase**
else f endif
case = *vset_equal*
then if set_descp (*td*)
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('vsubsetp
t,
t,
t,
t,
t,
componen
t,
v1,
v2)
∧ mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('vsub
t,
t,
t,
t,
t,
compo
t,
v2,
v1)
else f endif
case = *vsequence_equal*
then if sequence_descp (*td*)
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('vequal_l
t,
t,
t,
t,
t,
componen
t,
v1,
v2)

else f endif
case = *vmapping_equal*
then if mapping_descp (*td*)
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vequal**,
t,
t,
t,
t,
t,
mk_set_des
t,
vdomain (
vdomain (
∧ mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vequ**
t,
t,
t,
t,
t,
compo
t,
vcomp
mutua
else f endif
case = *vrecord_equal*
then if record_descp (*td*)
then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vfields_**
t,
t,
t,
t,
t,
t,

field_tds (*t*
v1,
v2)

else f endif

case = *vfields_equal*

then if *tds* ≃ **nil then t**

else mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vequal**,
t,
t,
t,
t,
t,
cdar (*tds*),
t,
fselect (*v1*,
caa
fselect (*v2*,
caa

∧ mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vfiel**
t,
t,
t,
t,
t,
t,
cdr (*tds*
v1,
v2) **en**

case = *varray_equal*

then if array_descp (*td*)

then mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vequal_l**
t,
t,
t,
t,
t,
componen
t,
vcompone
mutual-vm

else f endif
case = *vequal_list*
then if *v1* ≃ **nil then** *v2* ≃ **nil**
elseif *v2* ≃ **nil then f**
else mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('**vequal**,
t,
t,
t,
t,
t,
td,
t,
car (*v1*),
car (*v2*))
∧ mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('**vequa**
t,
t,
t,
t,
t,
td,
t,
cdr (*v1*
cdr (*v2*
case = *vmapped_value_list*
then if *ks* ≃ **nil then nil**
else cons (mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('**vmap**
t,
car (*k*
t,
m,
t,
td,
t,
t,
t),
mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('**vmap**
t,
t,

cdr (k
m,
t,
td,
t,
t,
t)) **en**

otherwise if $m \simeq$ **nil then** no_such_component_error (m, k)
elseif mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'veq**
t,
t,
t,
t,
t,
td,
t,
caar (
k)
then cdar (m)
else mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vmapp**
t,
k,
t,
cdr (m
t,
td,
t,
t,
t) **end**

DEFINITION:
vmember (e, s, td)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value (**'vmember**,
e,
t,
t,
t,
s,
td,
t,
t,
t)

DEFINITION:

vsubsetp (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vsubsetp`,
t,
t,
t,
t,
t,
td,
t,
v1,
v2)

Definition:
vequal (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vequal`,
t,
t,
t,
t,
t,
td,
t,
v1,
v2)

Definition:
vset_equal (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vset_equal`,
t,
t,
t,
t,
t,
td,
t,
v1,
v2)

Definition:
vsequence_equal (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vsequence_equal`,
t,
t,
t,
t,

t,
td,
t,
v1,
v2)

DEFINITION:
vmapping_equal (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vmapping_equal`,
t,
t,
t,
t,
t,
td,
t,
v1,
v2)

DEFINITION:
vrecord_equal (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vrecord_equal`,
t,
t,
t,
t,
t,
td,
t,
v1,
v2)

DEFINITION:
vfields_equal (*v1*, *v2*, *tds*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vfields_equal`,
t,
t,
t,
t,
t,
t,
tds,
v1,
v2)

DEFINITION:
varray_equal (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`varray_equal`,
t,
t,
t,
t,
t,
td,
t,
v1,
v2)

DEFINITION:
vequal_list (*v1*, *v2*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vequal_list`,
t,
t,
t,
t,
t,
td,
t,
v1,
v2)

DEFINITION:
vmapped_value_list (*m*, *ks*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vmapped_value_list`,
t,
t,
ks,
m,
t,
td,
t,
t,
t)

DEFINITION:
vmapped_value (*m*, *k*, *td*)
= mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_equal-varray_equal-vequal_list-vmapped_value_list-vmapped_value ('`vmapped_value`,
t,
k,
t,

m,
t,
td,
t,
t,
t)

DEFINITION: vsize (v) = length (v)

DEFINITION:
vseq_select $(v,\ k)$
= **if** $(k \in \mathbf{N}) \wedge (1 \leq k) \wedge (k \leq \text{vsize}\,(v))$ **then** nth $(k,\ v)$
else no_such_component_error $(v,\ k)$ **endif**

DEFINITION:
vsubseq_select $(v,\ lo,\ hi)$
= **if** $lo >$ vsize (v) **then nil**
elseif $(1 \leq lo) \wedge (lo \leq hi)$
then cons (vseq_select $(v,\ lo)$, vsubseq_select $(v,\ 1 + lo,\ hi)$)
else nil endif

DEFINITION:
vselect $(v,\ k,\ td)$
= **if** type_descp (td)
then case on root (td):
case = *array*
then vmapped_value $(v,\ k,$ selector_td $(td))$
case = *record*
then fselect $(v,\ k)$
case = *mapping*
then vmapped_value $(v,\ k,$ selector_td $(td))$
case = *sequence*
then vseq_select $(v,\ k)$
otherwise not_selectable_error (td) **endcase**
else not_type_descriptor_error (td) **endif**

DEFINITION:
vmap_put $(m,\ k,\ v,\ td)$
= **if** $m \simeq$ **nil then** cons (map_entry $(k,\ v)$, $m)$
elseif vequal (caar (m), k, $td)$ **then** cons (map_entry $(k,\ v)$, cdr $(m))$
else cons (car (m), vmap_put (cdr (m), k, v, $td))$ **endif**

DEFINITION:
varray_put $(a,\ i,\ v,\ td)$ = vmap_put $(a,\ i,\ v,$ selector_td $(td))$

DEFINITION: vrecord_put $(r,\ fn,\ v)$ = add_to_map $(r,\ fn,\ v)$

DEFINITION:
vmapping_put (m, i, v, td) = vmap_put (m, i, v, selector_td (td))

DEFINITION:
vsequence_put (s, i, v)
= **if** $s \simeq$ **nil** **then** s
elseif $i = 1$ **then** cons (v, cdr (s))
else cons (car (s), vsequence_put (cdr (s), $i - 1$, v)) **endif**

DEFINITION:
vsetp (s, td)
= **if** $s \simeq$ **nil** **then t**
else ($\neg$ vmember (car (s), cdr (s), td)) $\wedge$ vsetp (cdr (s), td) **endif**

DEFINITION:
vset (vs, td)
= **if** $vs \simeq$ **nil** **then nil**
elseif vmember (car (vs), cdr (vs), td) **then** vset (cdr (vs), td)
else cons (car (vs), vset (cdr (vs), td)) **endif**

DEFINITION:
vsubmapp ($v1$, $v2$, td)
= **if** $v1 \simeq$ **nil** **then t**
else let c **be** vselect ($v2$, caar ($v1$), td)
in
if errorp (c) **then f**
else vequal (cdar ($v1$), c, component_td (td))
$\wedge$ vsubmapp (cdr ($v1$), $v2$, td) **endif endlet endif**

DEFINITION:
vsubseqp ($v1$, $v2$, td)
= **if** $v1 \simeq$ **nil** **then t**
elseif $v2 \simeq$ **nil** **then f**
elseif vequal (car ($v1$), car ($v2$), td) **then** vsubseqp (cdr ($v1$), cdr ($v2$), td)
else vsubseqp ($v1$, cdr ($v2$), td) **endif**

DEFINITION:
vsubp ($v1$, $v2$, td)
= **case on** root (td):
case = *mapping*
then vsubmapp ($v1$, $v2$, td)
case = *sequence*
then vsubseqp ($v1$, $v2$, component_td (td))
case = *set*
then vsubsetp ($v1$, $v2$, component_td (td))
otherwise f endcase

DEFINITION:
vmap_remove (m, k, td)
= **if** $m \simeq$ **nil** **then** m
elseif vequal (caar (m), k, selector_td (td)) **then** cdr (m)
else cons (car (m), vmap_remove (cdr (m), k, td)) **endif**

DEFINITION:
vremove ($v1$, $v2$, td)
= **if** listp ($v2$)
then if vequal ($v1$, car ($v2$), td) **then** vremove ($v1$, cdr ($v2$), td)
else cons (car ($v2$), vremove ($v1$, cdr ($v2$), td)) **endif**
else $v2$ **endif**

DEFINITION:
vdifference ($v1$, $v2$, td)
= **if** $v1 \simeq$ **nil** **then** $v1$
elseif vmember (car ($v1$), $v2$, component_td (td))
then vdifference (cdr ($v1$), $v2$, td)
else cons (car ($v1$), vdifference (cdr ($v1$), $v2$, td)) **endif**

DEFINITION:
vdifference_maps ($v1$, $v2$, td)
= **if** $v1 \simeq$ **nil** **then** $v1$
else let c **be** vselect ($v2$, caar ($v1$), td)
in
if errorp (c)
then cons (car ($v1$), vdifference_maps (cdr ($v1$), $v2$, td))
elseif vequal (cdar ($v1$), c, component_td (td))
then vdifference_maps (cdr ($v1$), $v2$, td)
else mapping_merge_error ($v1$, $v2$) **endif endlet endif**

DEFINITION:
vintersect ($v1$, $v2$, td)
= **if** $v1 \simeq$ **nil** **then** $v1$
elseif vmember (car ($v1$), $v2$, component_td (td))
then cons (car ($v1$), vintersect (cdr ($v1$), $v2$, td))
else vintersect (cdr ($v1$), $v2$, td) **endif**

DEFINITION:
vintersect_maps ($v1$, $v2$, td)
= **if** $v1 \simeq$ **nil** **then** $v1$
else let c **be** vselect ($v2$, caar ($v1$), td)
in
if errorp (c) **then** vintersect_maps (cdr ($v1$), $v2$, td)
elseif vequal (cdar ($v1$), c, component_td (td))

then cons (car (*v1*), vintersect_maps (cdr (*v1*), *v2*, *td*))
else mapping_merge_error (*v1*, *v2*) **endif endlet endif**

DEFINITION:
vunion (*v1*, *v2*, *td*)
= **if** *v1* $\simeq$ **nil then** *v2*
elseif vmember (car (*v1*), *v2*, component_td (*td*))
then vunion (cdr (*v1*), *v2*, *td*)
else cons (car (*v1*), vunion (cdr (*v1*), *v2*, *td*)) **endif**

DEFINITION:
vunion_maps (*v1*, *v2*, *td*)
= **if** *v1* $\simeq$ **nil then** *v2*
else let *c* **be** vselect (*v2*, caar (*v1*), *td*)
in
if errorp (*c*)
then cons (car (*v1*), vunion_maps (cdr (*v1*), *v2*, *td*))
elseif vequal (cdar (*v1*), *c*, component_td (*td*))
then vunion_maps (cdr (*v1*), *v2*, *td*)
else mapping_merge_error (*v1*, *v2*) **endif endlet endif**

```
; ********************
;  Value Sets of Types
; ********************

; =====================
;  Value Set Utilities
; =====================
```

DEFINITION:
remove_larger (*s*, *n*)
= **if** *s* $\simeq$ **nil then** *s*
elseif length (car (*s*)) $>$ *n* **then** remove_larger (cdr (*s*), *n*)
else cons (car (*s*), remove_larger (cdr (*s*), *n*)) **endif**

DEFINITION:
list_cons (*x*, *y*)
= **if** *y* $\simeq$ **nil then nil**
else cons (cons (*x*, car (*y*)), list_cons (*x*, cdr (*y*))) **endif**

DEFINITION:
all_subsets (*vs*)
= **if** *vs* $\simeq$ **nil then** list (**nil**)
else append (list_cons (car (*vs*), all_subsets (cdr (*vs*))),
all_subsets (cdr (*vs*))) **endif**

DEFINITION:
all_seqs_n (*rvs*, *avs*, *n*)
= **if** *n* ≃ 0 **then** list (**nil**)
elseif *rvs* ≃ **nil then nil**
else append (list_cons (car (*rvs*), all_seqs_n (*avs*, *avs*, *n* − 1)),
all_seqs_n (cdr (*rvs*), *avs*, *n*)) **endif**

DEFINITION:
all_seqs_le_n (*vs*, *n*)
= **if** *n* ≃ 0 **then** list (**nil**)
else append (all_seqs_n (*vs*, *vs*, *n*), all_seqs_le_n (*vs*, *n* − 1)) **endif**

DEFINITION:
indexed_value_set (*is*, *rcs*, *acs*)
= **if** *is* ≃ **nil then** list (**nil**)
elseif *rcs* ≃ **nil then nil**
else append (list_cons (cons (car (*is*), car (*rcs*)),
indexed_value_set (cdr (*is*), *acs*, *acs*)),
indexed_value_set (*is*, cdr (*rcs*), *acs*)) **endif**

DEFINITION:
index_set_value_set (*iss*, *cs*)
= **if** *iss* ≃ **nil then nil**
else append (indexed_value_set (car (*iss*), *cs*, *cs*),
index_set_value_set (cdr (*iss*), *cs*)) **endif**

```
; ======================
;  Value Set Computation
; ======================
```

DEFINITION:
simple_value_set (*td*)
= **if** bounded_typep (*td*)
then if non_rational_simple_typep (*td*)
then if errorp (tmin (*td*)) **then** tmin (*td*)
elseif errorp (tmax (*td*)) **then** tmax (*td*)
else number_list (tmin (*td*), tmax (*td*)) **endif**
else rational_value_set_error (*td*) **endif**
else unbounded_value_set_error (*td*) **endif**

DEFINITION:
array_value_set (*is*, *cs*)
= **if** errorp (*is*) **then** *is*
elseif errorp (*cs*) **then** *cs*
else indexed_value_set (*is*, *cs*, *cs*) **endif**

THEOREM: count_cons
listp $(x) \rightarrow (\text{count}\,(x) = (1 + (\text{count}\,(\text{car}\,(x)) + \text{count}\,(\text{cdr}\,(x)))))$

THEOREM: zero_count_imp_not_list
$(\text{count}\,(x) = 0) \rightarrow (\neg\ \text{listp}\,(x))$

THEOREM: listp_imp_count_not_zero
listp $(x) \rightarrow (\text{count}\,(x) \neq 0)$

EVENT: Disable count_cons.

EVENT: Disable zero_count_imp_not_list.

THEOREM: listp_cdar_imp_sub1_count_not_zero
listp $(\text{cdar}\,(x)) \rightarrow ((\text{count}\,(x) - 1) \neq 0)$

THEOREM: lessp_plus_cdr_caar_cddar
$(\text{listp}\,(fs) \wedge \text{listp}\,(\text{cdar}\,(fs)))$
$\rightarrow \quad ((\text{count}\,(\text{cdr}\,(fs)) + \text{count}\,(\text{caar}\,(fs)) + \text{count}\,(\text{cddar}\,(fs)))$
$\quad < \quad ((\text{count}\,(fs) - 1) - 1))$

DEFINITION:
field_list_sets (fs)
= **if** $fs \simeq$ **nil then** list (**nil**)
else let fn **be** caar (fs),
fvs **be** cdar (fs)
in
if $fvs \simeq$ **nil then nil**
else append (list_cons (cons (fn, car (fvs)),
field_list_sets (cdr (fs)))),
field_list_sets (cons (cons (fn,
cdr (fvs))),
cdr (fs))))) **endif endlet endif**

DEFINITION:
record_value_set (fs)
= **if** errorp (fs) **then** fs
else field_list_sets (fs) **endif**

DEFINITION:
mapping_value_set (sl, ss, cs)
= **if** errorp (sl) **then** sl
elseif errorp (ss) **then** ss
elseif errorp (cs) **then** cs

else let r **be** index_set_value_set (all_subsets (ss), cs)
in
if sl = **nil then** r
else remove_larger (r, sl) **endif endlet endif**

DEFINITION:
sequence_value_set (sl, cs)
= **if** errorp (sl) **then** sl
elseif errorp (cs) **then** cs
elseif $sl \in$ **N then** all_seqs_le_n (cs, sl)
else UNBOUNDED_SEQUENCE_VALUE_SET_ERROR **endif**

DEFINITION:
set_value_set (sl, cs)
= **if** errorp (sl) **then** sl
elseif errorp (cs) **then** cs
elseif $sl \in$ **N then** remove_larger (all_subsets (cs), sl)
else all_subsets (cs) **endif**

```
#|
(do-mutual '( ; the value_set do-mutual

(defn field_value_sets (fds)
  (if (nlistp fds)
      fds
    (let ((fv1 (value_set (cdar fds)))
  (fvs (field_value_sets (cdr fds))))
      (if (errorp fv1)
  fv1
      (if (errorp fvs)
  fvs
(cons (map_entry (caar fds) fv1) fvs)))))
  ( (lessp (tree_size fds)) ))

(defn value_set (td)
  ; entry td is a correct type descriptor
  ; returns an errorp or a list of values
  (if (type_descp td)
      (case (root td)
(array (array_value_set (value_set (selector_td td))
(value_set (component_td td))))
(record (record_value_set (field_value_sets (field_tds td))))
(mapping (mapping_value_set (max_size td)
    (value_set (selector_td td))
    (value_set (component_td td))))
```

```
(sequence (sequence_value_set (max_size td)
      (value_set (component_td td))))
(set (set_value_set (max_size td)
    (value_set (component_td td))))
(pending (pending_type_value_set_error td))
(error* td)
(otherwise ; (simple_descp td)
  (simple_value_set td)))
    (not_type_descriptor_error td))
  ( (lessp (tree_size td)) ))

))
|#
```

DEFINITION:
mutual-value_set-field_value_sets (*mutual-flg*, *fds*, *td*)
= **if** *mutual-flg* = `'value_set`
then if type_descp (*td*)
then case on root (*td*):
case = *array*
then array_value_set (mutual-value_set-field_value_sets (`'value_set`,
t,
selector_td (*td*)),
mutual-value_set-field_value_sets (`'value_set`,
t,
component_td (*td*)))
case = *record*
then record_value_set (mutual-value_set-field_value_sets (`'field_value_sets`,
field_tds (*td*),
t))
case = *mapping*
then mapping_value_set (max_size (*td*),
mutual-value_set-field_value_sets (`'value_set`,
t,
selector_td (*td*)),
mutual-value_set-field_value_sets (`'value_set`,
t,
component_td (*td*)))
case = *sequence*
then sequence_value_set (max_size (*td*),
mutual-value_set-field_value_sets (`'value_set`,
t,
component_td (*td*)))

```
            case = set
             then set_value_set (max_size (td),
                                 mutual-value-set-field_value_sets ('value_set,
                                                                    t,
                                                                    component_td (td)))
            case = pending
             then pending_type_value_set_error (td)
            case = error*
             then td
            otherwise simple_value_set (td) endcase
       else not_type_descriptor_error (td) endif
  elseif fds ≃ nil then fds
  elseif errorp (mutual-value_set-field_value_sets ('value_set,
                                                    t,
                                                    cdar (fds)))
  then mutual-value_set-field_value_sets ('value_set, t, cdar (fds))
  elseif errorp (mutual-value_set-field_value_sets ('field_value_sets,
                                                    cdr (fds),
                                                    t))
  then mutual-value_set-field_value_sets ('field_value_sets,
                                          cdr (fds),
                                          t)
  else cons (map_entry (caar (fds),
                        mutual-value_set-field_value_sets ('value_set,
                                                           t,
                                                           cdar (fds))),
             mutual-value_set-field_value_sets ('field_value_sets,
                                                cdr (fds),
                                                t)) endif
```

DEFINITION:
value_set (*td*) = mutual-value_set-field_value_sets (`'value_set`, **t**, *td*)

DEFINITION:
field_value_sets (*fds*)
= mutual-value_set-field_value_sets (`'field_value_sets`, *fds*, **t**)

```
; See below for marked and typed value sets.



; **************
;  Typed Values
; **************
```

```
; ------------------------------------------------------------------------
; A typed value is constructed by calling function
;
;     typed (td,u)
;
; where td is a type descriptor and u is a member of td's value set.
; The value of dtype(td,u) = T iff u is a member of td's value set.
; ------------------------------------------------------------------------
```

DEFINITION:
dtype_size (n, v)
= **if** errorp (n) **then** n
elseif $n \in \mathbf{N}$ **then** length $(v) \leq n$
else t endif

THEOREM: count_keys
(key_value_mapp $(v) \wedge$ (count (keys (v)) $\not<$ count (v)))
$\rightarrow$ (count (keys (v)) = count (v))

THEOREM: count_key_values
(key_value_mapp $(v) \wedge$ (count (key_values (v)) $\not<$ count (v)))
$\rightarrow$ (count (key_values (v)) = count (v))

```
#|
(do-mutual '(

(defn dtype_list (td vs)
  (if (nlistp vs)
      (equal vs nil)
    (let ((r (dtype td (car vs))))
      (if (truep r)
  (dtype_list td (cdr vs))
r)))
  ( (ord-lessp (cons (add1 (count vs))
     (tree_size td))) ))

(defn dtype_fields (fds fvs)
  (if (nlistp fvs)
      (equal fvs nil)
    (let ((r (dtype (mapped_value fds (caar fvs)) (cdar fvs))))
      (if (truep r)
  (dtype_fields fds (cdr fvs))
r)))
  ( (ord-lessp (cons (add1 (count fvs))
```

```
     (tree_size fds))) ))

(defn dtype (td v)
  ; entry td is a correct type descriptor
  ; v is a Gypsy value
  ; returns T, F, or an errorp
  (if (type_descp td)
      (case (root td)
(array (if (key_value_mapp v)
   (let ((ivs (value_set (selector_td td)))
 (r (dtype_list (selector_td td) (vindexes v))))
     (if (truep r)
 (if (errorp ivs)
     ivs
   (if (and (vsetp (vindexes v) (selector_td td))
    (vset_equal ivs (vindexes v)
(mk_set_desc nil
     (selector_td td)
     (null_set))))
       (dtype_list (component_td td) (vcomponents v))
     F))
       r))
 F))
(record (if (and (key_value_mapp v)
 (setp (field_names v))
 (set_equal (field_names (field_tds td))
    (field_names v)))
    (dtype_fields (field_tds td) v)
  F))
(mapping (if (key_value_mapp v)
     (let ((r (dtype_list (selector_td td) (vdomain v))))
       (if (truep r)
   (if (vsetp (vdomain v) (selector_td td))
       (let ((r (dtype_list (component_td td)
    (vrange v))))
 (if (truep r)
     (dtype_size (max_size td) v)
   r))
     F)
 r))
   F))
(sequence (let ((r (dtype_list (component_td td) v)))
    (if (truep r)
(dtype_size (max_size td) v)
```

```
      r)))
(set (if (vsetp v (component_td td))
 (let ((r (dtype_list (component_td td) v)))
   (if (truep r)
       (dtype_size (max_size td) v)
     r))
       F))
(pending (pending_in_type_error td))
(error* td)
(otherwise ; (simple_descp td)
  (if (errorp (tmin td))
      (tmin td)
  (if (errorp (tmax td))
      (tmax td)
  (if (equal (tid td) 'rational)
      (and (rationalp v)
   (if (bounded_typep td)
       (and (rleq (tmin td) v)
    (rleq v (tmax td)))
     T))
    (and (integerp v)
 (if (bounded_typep td)
     (and (ileq (tmin td) v)
  (ileq v (tmax td)))
   T)))))))
    (not_type_descriptor_error td))
  ( (ord-lessp (cons (add1 (count v))
     (tree_size td))) ))

))
|#
```

DEFINITION:
mutual-dtype-dtype_fields-dtype_list (*mutual-flg*, *fds*, *fvs*, *td*, *v*, *vs*)
= **case on** *mutual-flg*:
case = *dtype*
then if type_descp (*td*)
then case on root (*td*):
case = *array*
then if key_value_mapp (*v*)
then if truep (mutual-dtype-dtype_fields-dtype_list (`'dtype_list`,
t,
t,

```
                                                    selector_td (td),
                                                    t,
                                                    vindexes (v)))
            then if errorp (value_set (selector_td (td)))
                 then value_set (selector_td (td))
                 elseif vsetp (vindexes (v), selector_td (td))
                        ∧   vset_equal (value_set (selector_td (td)),
                                        vindexes (v),
                                        mk_set_desc (nil,
                                                     selector_td (td),
                                                     NULL_SET))
                 then mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                            t,
                                                            t,
                                                            component_td (td),
                                                            t,
                                                            vcomponents (v))
                 else f endif
            else mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                       t,
                                                       t,
                                                       selector_td (td),
                                                       t,
                                                       vindexes (v)) endif
       else f endif
case = record
 then if key_value_mapp (v)
         ∧   setp (field_names (v))
         ∧   set_equal (field_names (field_tds (td)),
                        field_names (v))
       then mutual-dtype-dtype_fields-dtype_list ('dtype_fields,
                                                  field_tds (td),
                                                  v,
                                                  t,
                                                  t,
                                                  t)
       else f endif
case = mapping
 then if key_value_mapp (v)
       then if truep (mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                            t,
                                                            t,
                                                            selector_td (td),
                                                            t,
```

```
                                                    vdomain (v)))
            then if vsetp (vdomain (v), selector_td (td))
                 then if truep (mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                                      t,
                                                                      t,
                                                                      component_td (td),
                                                                      t,
                                                                      vrange (v)))
                      then dtype_size (max_size (td), v)
                      else mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                                 t,
                                                                 t,
                                                                 component_td (td),
                                                                 t,
                                                                 vrange (v)) endif
                 else f endif
            else mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                       t,
                                                       t,
                                                       selector_td (td),
                                                       t,
                                                       vdomain (v)) endif
       else f endif
case = sequence
 then if truep (mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                      t,
                                                      t,
                                                      component_td (td),
                                                      t,
                                                      v))
       then dtype_size (max_size (td), v)
       else mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                  t,
                                                  t,
                                                  component_td (td),
                                                  t,
                                                  v) endif
case = set
 then if vsetp (v, component_td (td))
       then if truep (mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                            t,
                                                            t,
                                                            component_td (td),
                                                            t,
```

```
                                                                    v))
                    then dtype_size (max_size (td), v)
                    else mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                               t,
                                                               t,
                                                               component_td (td),
                                                               t,
                                                               v) endif
                else f endif
            case = pending
             then pending_in_type_error (td)
            case = error*
             then td
            otherwise if errorp (tmin (td))  then tmin (td)
                      elseif errorp (tmax (td))  then tmax (td)
                      elseif tid (td) = 'rational
                      then rationalp (v)
                           ∧   if bounded_typep (td)
                               then rleq (tmin (td), v)
                                    ∧   rleq (v, tmax (td))
                               else t endif
                      else integerp (v)
                           ∧   if bounded_typep (td)
                               then ileq (tmin (td), v)
                                    ∧   ileq (v, tmax (td))
                               else t endif endif endcase
      else not_type_descriptor_error (td) endif
case = dtype_fields
 then if fvs ≃ nil  then fvs = nil
      elseif truep (mutual-dtype-dtype_fields-dtype_list ('dtype,
                                                          t,
                                                          t,
                                                          mapped_value (fds,
                                                                        caar (fvs)),
                                                          cdar (fvs),
                                                          t))
      then mutual-dtype-dtype_fields-dtype_list ('dtype_fields,
                                                 fds,
                                                 cdr (fvs),
                                                 t,
                                                 t,
                                                 t)
      else mutual-dtype-dtype_fields-dtype_list ('dtype,
                                                 t,
```

```
                                                t,
                                                mapped_value (fds,
                                                              caar (fvs)),
                                                cdar (fvs),
                                                t) endif
    otherwise if vs ≃ nil then vs = nil
              elseif truep (mutual-dtype-dtype_fields-dtype_list ('dtype,
                                                                  t,
                                                                  t,
                                                                  td,
                                                                  car (vs),
                                                                  t))
              then mutual-dtype-dtype_fields-dtype_list ('dtype_list,
                                                         t,
                                                         t,
                                                         td,
                                                         t,
                                                         cdr (vs))
              else mutual-dtype-dtype_fields-dtype_list ('dtype,
                                                         t,
                                                         t,
                                                         td,
                                                         car (vs),
                                                         t) endif endcase
```

DEFINITION:
dtype (*td*, *v*) = mutual-dtype-dtype_fields-dtype_list (`'dtype`, **t**, **t**, *td*, *v*, **t**)

DEFINITION:
dtype_fields (*fds*, *fvs*)
= mutual-dtype-dtype_fields-dtype_list (`'dtype_fields`, *fds*, *fvs*, **t**, **t**, **t**)

DEFINITION:
dtype_list (*td*, *vs*)
= mutual-dtype-dtype_fields-dtype_list (`'dtype_list`, **t**, **t**, *td*, **t**, *vs*)

EVENT: Add the shell *mk_typed*, with recognizer function symbol *typedp* and 2 accessors: *type_part*, with type restriction (none-of) and default value false; *value_part*, with type restriction (none-of) and default value zero.

```
; member of value set of type_part
```

DEFINITION:

typed (td, u)
= **if** truep (dtype (td, u)) **then** mk_typed (td, u)
else mk_typed (INTEGER_DESC, 0) **endif**

```
; *********************
;  Marked Typed Values
; *********************

; ------------------------------------------------------------------------
; Whether a typed value is determinate or indeterminate is indicated by
; marking it with nil for determinate typed values and non-nil for
; indeterminate typed values.
; ------------------------------------------------------------------------
```

DEFINITION:
determinate (x)
= **if** markedp (x) **then** mark (x) = **nil**
else f endif

DEFINITION: indeterminate (x) = ($\neg$ determinate (x))

DEFINITION:
marked_typed (td, u)
= **if** truep (dtype (td, u)) **then** marked (**nil**, typed (td, u))
else marked (not_in_type_error (u, td), typed (td, u)) **endif**

DEFINITION:
marked_typed_list (td, us)
= **if** $us \simeq$ **nil then** us
else cons (marked_typed (td, car (us)),
marked_typed_list (td, cdr (us))) **endif**

```
; ***********************************
;  Extraction of Type and Value Parts
; ***********************************
```

DEFINITION:
type (x)
= **if** markedp (x) **then** type (unmark (x))
elseif typedp (x) **then** type_part (x)
else f endif

DEFINITION:

value (x)
= **if** markedp (x) **then** value (unmark (x))
elseif typedp (x) **then** value_part (x)
else x **endif**

DEFINITION:
values (s)
= **if** $s \simeq$ **nil then nil**
else cons (value (car (s)), values (cdr (s))) **endif**

```
; *************************
;  Marked Typed Value Sets
; *************************
```

DEFINITION:
marked_typed_value_set (td)
= **let** vs **be** value_set (td)
in
if errorp (vs) **then** vs
else marked_typed_list (td, vs) **endif endlet**

```
; *************************
;  Default Values of Types
; *************************
```

DEFINITION:
mk_array_default (id, cd)
= **let** is **be** value_set (id),
cv **be** udv (cd)
in
if errorp (is) **then** is
elseif errorp (cv) **then** cv
else pair_list_map (is, ncopies (length (is), cv)) **endif endlet**

DEFINITION:
mk_record_default (fds)
= **if** $fds \simeq$ **nil then nil**
else let $fv1$ **be** udv (cdar (fds))
in
if errorp ($fv1$) **then** $fv1$
else let fvs **be** mk_record_default (cdr (fds))
in
if errorp (fvs) **then** fvs

else cons (map_entry (caar (*fds*),
fv1),
fvs) **endif endlet endif endlet endif**

DEFINITION:
default_value (*td*)
= **if** type_descp (*td*) ∧ (¬ errorp (udv (*td*))) **then** typed (*td*, udv (*td*))
else typed (INTEGER_DESC, 0) **endif**

```
; ***********
;  Base Type
; ***********

#|
(do-mutual '(

(defn field_base_types (fds)
  (if (nlistp fds)
      fds
    (cons (map_entry (caar fds) (base_type (cdar fds)))
  (field_base_types (cdr fds))))
  ( (lessp (tree_size fds)) ))

(defn base_type (td)
  ; entry td is a correct type descriptor
  (if (type_descp td)
      (case (root td)
(integer (integer_desc))
(rational (rational_desc))
(scalar (mk_scalar_desc (tid td) (sid td) (crd td)
0 (sub1 (crd td)) 0))
        (array (let ((id (selector_td td))
     (cd (base_type (component_td td))))
 (mk_array_desc id cd (mk_array_default id cd))))
(record (let ((fds (field_base_types (field_tds td))))
  (mk_record_desc fds (mk_record_default fds))))
(mapping (mk_mapping_desc nil (base_type (selector_td td))
  (base_type (component_td td))
  (null_map)))
(sequence (mk_sequence_desc nil (base_type (component_td td))
    (null_seq)))
(set (mk_set_desc nil (base_type (component_td td)) (null_set)))
(pending td)
(error* td)
```

```
(otherwise (not_type_descriptor_error td)))
    (not_type_descriptor_error td))
  ( (lessp (tree_size td)) ))

))
|#
```

DEFINITION:
mutual-base_type-field_base_types (*mutual-flg*, *fds*, *td*)
= **if** *mutual-flg* = 'base_type
then if type_descp (*td*)
then case on root (*td*):
case = *integer*
then INTEGER_DESC
case = *rational*
then RATIONAL_DESC
case = *scalar*
then mk_scalar_desc (tid (*td*),
sid (*td*),
crd (*td*),
0,
crd (*td*) − 1,
0)
case = *array*
then mk_array_desc (selector_td (*td*),
mutual-base_type-field_base_types ('base_type,
t,
component_td (*td*)),
mk_array_default (selector_td (*td*),
mutual-base_type-field_base_types ('base_type,
t,
component_td (*td*))))
case = *record*
then mk_record_desc (mutual-base_type-field_base_types ('field_base_types,
field_tds (*td*),
t),
mk_record_default (mutual-base_type-field_base_types ('field_base_types,
field_tds (*td*),
t)))
case = *mapping*
then mk_mapping_desc (**nil**,
mutual-base_type-field_base_types ('base_type,
t,

```
                                                                    selector_td (td)),
                                        mutual-base_type-field_base_types ('base_type,
                                                                    t,
                                                                    component_td (td)),
                                        NULL_MAP)
            case = sequence
             then mk_sequence_desc (nil,
                                        mutual-base_type-field_base_types ('base_type,
                                                                    t,
                                                                    component_td (td)),
                                        NULL_SEQ)
            case = set
             then mk_set_desc (nil,
                                mutual-base_type-field_base_types ('base_type,
                                                                t,
                                                                component_td (td)),
                                NULL_SET)
            case = pending
             then td
            case = error*
             then td
            otherwise not_type_descriptor_error (td) endcase
        else not_type_descriptor_error (td) endif
    elseif fds ≃ nil then fds
    else cons (map_entry (caar (fds),
                        mutual-base_type-field_base_types ('base_type,
                                                        t,
                                                        cdar (fds))),
            mutual-base_type-field_base_types ('field_base_types,
                                            cdr (fds),
                                            t)) endif
```

DEFINITION:
base_type (*td*) = mutual-base_type-field_base_types ('base_type, t, *td*)

DEFINITION:
field_base_types (*fds*)
= mutual-base_type-field_base_types ('field_base_types, *fds*, t)

```
; ***************
;  Type Equality
; ***************
```

DEFINITION:

type_vequal (*v1*, *v2*, *td*)
= **if** errorp (*v1*) ∨ errorp (*v2*) **then f**
elseif *v1* = **nil then** *v2* = **nil**
else vequal (*v1*, *v2*, *td*) **endif**

```
#|
(do-mutual '(

(defn field_tds_equal (t1 t2)
  (if (nlistp t1)
      T
    (and (type_equal (cdar t1) (mapped_value t2 (caar t1)))
 (field_tds_equal (cdr t1) t2)))
  ( (lessp (tree_size t1)) ))

(defn type_equal (t1 t2)
  (if (and (type_descp t1) (type_descp t2)
   (equal (root t1) (root t2)))
      (case (root t1)
(integer (and (type_vequal (tmin t1) (tmin t2) t1)
      (type_vequal (tmax t1) (tmax t2) t1)))
(rational (and (type_vequal (tmin t1) (tmin t2) t1)
       (type_vequal (tmax t1) (tmax t2) t1)))
(scalar (and (equal (tid t1) (tid t2))
     (equal (sid t1) (sid t2))
     (type_vequal (crd t1) (crd t2) (integer_desc))
     (type_vequal (tmin t1) (tmin t2) t1)
     (type_vequal (tmax t1) (tmax t2) t1)))
(array (and (type_equal (selector_td t1) (selector_td t2))
    (type_equal (component_td t1) (component_td t2))))
(record (and (set_equal (field_names t1) (field_names t2))
     (field_tds_equal (field_tds t1) (field_tds t2))))
(mapping (and (type_vequal (max_size t1) (max_size t2) (integer_desc))
      (type_equal (selector_td t1) (selector_td t2))
      (type_equal (component_td t1) (component_td t2))))
(sequence (and (type_vequal (max_size t1) (max_size t2) (integer_desc))
       (type_equal (component_td t1) (component_td t2))))
(set (and (type_vequal (max_size t1) (max_size t2) (integer_desc))
  (type_equal (component_td t1) (component_td t2))))
(pending (and (equal (tid t1) (tid t2))
      (equal (sid t1) (sid t2))))
(otherwise F))
    F)
  ( (lessp (tree_size t1)) ))
```

```
))
|#
```

DEFINITION:

```
mutual-type_equal-field_tds_equal (mutual-flg, t1, t2)
=   if mutual-flg = 'type_equal
    then if type_descp (t1)
            ∧   type_descp (t2)
            ∧   (root (t1) = root (t2))
         then case on root (t1):
              case = integer
              then type_vequal (tmin (t1), tmin (t2), t1)
                   ∧   type_vequal (tmax (t1), tmax (t2), t1)
              case = rational
               then type_vequal (tmin (t1), tmin (t2), t1)
                    ∧   type_vequal (tmax (t1), tmax (t2), t1)
              case = scalar
               then (tid (t1) = tid (t2))
                    ∧   (sid (t1) = sid (t2))
                    ∧   type_vequal (crd (t1), crd (t2), INTEGER_DESC)
                    ∧   type_vequal (tmin (t1), tmin (t2), t1)
                    ∧   type_vequal (tmax (t1), tmax (t2), t1)
              case = array
               then mutual-type_equal-field_tds_equal ('type_equal,
                                                       selector_td (t1),
                                                       selector_td (t2))
                    ∧   mutual-type_equal-field_tds_equal ('type_equal,
                                                           component_td (t1),
                                                           component_td (t2))
              case = record
               then set_equal (field_names (t1), field_names (t2))
                    ∧   mutual-type_equal-field_tds_equal ('field_tds_equal,
                                                           field_tds (t1),
                                                           field_tds (t2))
              case = mapping
               then type_vequal (max_size (t1), max_size (t2), INTEGER_DESC)
                    ∧   mutual-type_equal-field_tds_equal ('type_equal,
                                                           selector_td (t1),
                                                           selector_td (t2))
                    ∧   mutual-type_equal-field_tds_equal ('type_equal,
                                                           component_td (t1),
                                                           component_td (t2))
```

```
                case = sequence
                 then type_vequal(max_size(t1), max_size(t2), INTEGER_DESC)
                      ∧  mutual-type_equal-field_tds_equal('type_equal,
                                                           component_td(t1),
                                                           component_td(t2))
                case = set
                 then type_vequal(max_size(t1), max_size(t2), INTEGER_DESC)
                      ∧  mutual-type_equal-field_tds_equal('type_equal,
                                                           component_td(t1),
                                                           component_td(t2))
                case = pending
                 then (tid(t1) = tid(t2)) ∧ (sid(t1) = sid(t2))
                otherwise f endcase
          else f endif
     elseif t1 ≃ nil then t
     else mutual-type_equal-field_tds_equal('type_equal,
                                            cdar(t1),
                                            mapped_value(t2, caar(t1)))
          ∧  mutual-type_equal-field_tds_equal('field_tds_equal,
                                               cdr(t1),
                                               t2) endif
```

DEFINITION:

```
type_equal(t1, t2)
=  mutual-type_equal-field_tds_equal('type_equal, t1, t2)
```

DEFINITION:

```
field_tds_equal(t1, t2)
=  mutual-type_equal-field_tds_equal('field_tds_equal, t1, t2)

; *********
;  In Type
; *********


DEFINITION:
in_type(td, u)
=  if error_descp(td) then td
   elseif error_descp(type(u)) then type(u)
   elseif type_equal(td, type(u)) then t
   elseif base_type(td) = base_type(type(u)) then dtype(td, value(u))
   else f endif

; ***************************************
;  Lemmas for Proving Type_Desc Terminates
```

; **

DEFINITION:
all_type_units (ul, sn)
= **if** $ul \simeq$ **nil then nil**
elseif kind (car (ul)) = **'type**
then cons (cons (sn, car (ul)), all_type_units (cdr (ul), sn))
else all_type_units (cdr (ul), sn) **endif**

DEFINITION:
all_scope_types (sl)
= **if** $sl \simeq$ **nil then nil**
else append (all_type_units (unit_list (car (sl)),
scope_name (car (sl))),
all_scope_types (cdr (sl))) **endif**

DEFINITION: all_gypsy_types (x) = all_scope_types (scope_list (x))

THEOREM: all_type_units_members
(($u \in ul$) $\wedge$ (kind (u) = **'type**))
$\rightarrow$ (cons (sn, u) $\in$ all_type_units (ul, sn))

THEOREM: member_append
(($e \in x$) $\vee$ ($e \in y$)) $\rightarrow$ ($e \in$ append (x, y))

THEOREM: all_scope_types_members
(($s \in sl$) $\wedge$ ($u \in$ unit_list (s)) $\wedge$ (kind (u) = **'type**))
$\rightarrow$ (cons (scope_name (s), u) $\in$ all_scope_types (sl))

THEOREM: all_gypsy_type_members
(($s \in$ scope_list (x)) $\wedge$ ($u \in$ unit_list (s)) $\wedge$ (kind (u) = **'type**))
$\rightarrow$ (cons (scope_name (s), u) $\in$ all_gypsy_types (x))

THEOREM: rule_imp_root_equal_lhs
rule (tr, prodn (lhs, rhs)) $\rightarrow$ (root (tr) = untag (lhs))

THEOREM: not_root_equal_lhs_imp_not_rule
(($r \neq nt$) $\wedge$ litatom (nt))
$\rightarrow$ (rule (mk_tree (r, s), prodn (tag (nt, l), rhs)) = **f**)

THEOREM: mk_error_kind
kind (mk_error (x)) = **'error**

THEOREM: all_scopes_car
(length (all_scopes (sn, sl)) = **1**)
$\rightarrow$ ((scope_name (car (all_scopes (sn, sl)))) = sn)
$\wedge$ (car (all_scopes (sn, sl)) $\in$ sl))

THEOREM: all_units_car
(length (all_units (un, ul)) = 1)
→ ((local_name (car (all_units (un, ul))) = un)
∧ (car (all_units (un, ul)) ∈ ul))

THEOREM: mref_result
(kind (ref_unit (mref (un, sn, x, n))) ≠ 'error)
→ ((car (all_scopes (ref_scope (mref (un, sn, x, n)), scope_list (x)))
∈ scope_list (x))
∧ (scope_name (car (all_scopes (ref_scope (mref (un, sn, x, n)),
scope_list (x))))
= ref_scope (mref (un, sn, x, n)))
∧ (ref_unit (mref (un, sn, x, n))
∈ unit_list (car (all_scopes (ref_scope (mref (un, sn, x, n)),
scope_list (x))))))

THEOREM: refed_type
(kind (ref_unit (ref (un, sn, x))) = 'type)
→ ((car (all_scopes (ref_scope (ref (un, sn, x)), scope_list (x)))
∈ scope_list (x))
∧ (ref_unit (ref (un, sn, x))
∈ unit_list (car (all_scopes (ref_scope (ref (un, sn, x)),
scope_list (x)))))
∧ (scope_name (car (all_scopes (ref_scope (ref (un, sn, x)),
scope_list (x))))
= ref_scope (ref (un, sn, x))))

THEOREM: type_in_all_types
(kind (ref_unit (ref (un, sn, x))) = 'type)
→ (ref (un, sn, x) ∈ all_gypsy_types (x))

THEOREM: set_difference_remove
listp (y)
→ (set_difference (x, y) = remove (car (y), set_difference (x, cdr (y))))

THEOREM: set_difference_member
((e ∈ x) ∧ (e ∉ y)) → (e ∈ set_difference (x, y))

DEFINITION:
available_types (ut, x) = set_difference (all_gypsy_types (x), ut)

THEOREM: lessp_available_types
((kind (ref_unit (ref (un, sn, x))) = 'type) ∧ (ref (un, sn, x) ∉ ut))
→ (length (available_types (cons (ref (un, sn, x), ut), x))
< length (available_types (ut, x)))

```
; ****************************
;  Creation of Type Descriptors
; ****************************

; ==========================
;  Descriptors for the Types
; ==========================

; --------------
;  Simple Types
; --------------

; Descriptors for standard types are defined above:
;
;     type name       function
;     ---------     --------------
;     boolean       boolean_desc
;     character     character_desc
;     integer       integer_desc
;     rational      rational_desc
```

DEFINITION:
subrange_desc (td, lo, hi)
= **let** rd **be** set_tmin (set_tmax (td, hi), lo)
in
if truep (dtype (rd, udv (td))) **then** rd
else set_udv (rd, lo) **endif endlet**

DEFINITION:
scalar_desc (tn, sn, sc) = mk_scalar_desc (tn, sn, sc, 0, $sc - 1$, 0)

```
; ------------------
;  Structured Types
; ------------------
```

DEFINITION:
array_desc (id, cd)
= **if** error_descp (id) **then** id
elseif error_descp (cd) **then** cd
elseif index_typep (id)
then mk_array_desc (id, cd, mk_array_default (id, cd))
else array_index_error (id) **endif**

DEFINITION:
record_desc (*fds*)
= **if** error_descp (*fds*) **then** *fds*
elseif setp (field_names (*fds*))
then if some_reserved_idp (field_names (*fds*))
then field_name_reserved_error (*fds*)
else mk_record_desc (*fds*, mk_record_default (*fds*)) **endif**
else duplicate_field_names_error (*fds*) **endif**

DEFINITION:
mapping_desc (*sl*, *sd*, *cd*)
= **if** error_descp (*sd*) **then** *sd*
elseif error_descp (*cd*) **then** *cd*
elseif equality_typep (*sd*) **then** mk_mapping_desc (*sl*, *sd*, *cd*, NULL_MAP)
else mapping_selector_type_error (*sd*) **endif**

DEFINITION:
sequence_desc (*sl*, *cd*)
= **if** error_descp (*cd*) **then** *cd*
else mk_sequence_desc (*sl*, *cd*, NULL_SEQ) **endif**

DEFINITION:
set_desc (*sl*, *cd*)
= **if** error_descp (*cd*) **then** *cd*
else mk_set_desc (*sl*, *cd*, NULL_SET) **endif**

DEFINITION:
pending_desc (*tid*, *sid*)
= mk_pending_desc (*tid*, *sid*, pending_default_value_error (*tid*, *sid*))

```
; ================================================
;  Evaluation of Expressions from Type Declarations
; ================================================
```

DEFINITION:
default_initial_value (*d*, *sn*, *x*)
= **if** rule (*d*,
prodn (tag (`'opt_default_initial_value_expression`,
`'v`),
list (`'colon_equal`, tag (`'expression`, `'e`))))
then precomputable_f (subtree (*d*, `'expression`), *sn*, *x*)
elseif rule (*d*,
prodn (tag (`'opt_default_initial_value_expression`,
`'v`),
`'empty`)) **then nil**
else opt_default_value_error (*d*, *sn*) **endif**

```
DEFINITION:
size_limit (r, sn, x)
=  if rule (r,
          prodn (tag ('opt_size_limit_restriction, 'r),
                 list ('open_paren,
                       tag ('expression, 'e),
                       'close_paren)))
   then let sl be precomputable_f (subtree (r, 'expression), sn, x)
        in
        if determinate (sl)
        then let ok be in_type (INTEGER_DESC, sl)
             in
             if errorp (ok) then ok
             elseif truep (ok) ∧ ileq (0, value (sl))
             then value (sl)
             else size_limit_error (r, sn) endif endlet
        elseif errorp (mark (sl)) then mark (sl)
        else mk_error (mark (sl)) endif endlet
   elseif rule (r,
              prodn (tag ('opt_size_limit_restriction, 'r),
                     'empty)) then nil
   else opt_size_limit_error (r, sn) endif

DEFINITION:
range_min (r, sn, x)
=  if rule (r,
          prodn (tag ('range, 'r),
                 list ('open_paren,
                       tag ('range_limits, 'r2),
                       'close_paren)))
   then range_min (subtree (r, 'range_limits), sn, x)
   elseif rule (r,
              prodn (tag ('range_limits, 'r),
                     list (tag ('expression, 'lo),
                           'dot_dot,
                           tag ('expression, 'hi))))
   then precomputable_f (subtree_i (r, 'expression, 1), sn, x)
   else not_range_error (r, sn) endif

DEFINITION:
range_max (r, sn, x)
=  if rule (r,
          prodn (tag ('range, 'r),
                 list ('open_paren,
```

```
                    tag('range_limits, 'r2),
                    'close_paren)))
    then range_max(subtree(r, 'range_limits), sn, x)
    elseif rule(r,
                prodn(tag('range_limits, 'r),
                    list(tag('expression, 'lo),
                        'dot_dot,
                        tag('expression, 'hi))))
    then precomputable_f(subtree_i(r, 'expression, 2), sn, x)
    else not_range_error(r, sn) endif

; ============================
;  Range and Default Setting
; ============================
```

DEFINITION:

```
value_setting(td, v, vkind)
=   if errorp(v) then v
    elseif indeterminate(v) then mk_error(v)
    else let r be in_type(td, unmark(v))
        in
        if truep(r) then value(v)
        elseif errorp(r) then r
        else mk_error(list(vkind,
                           unmark(v),
                           'not,
                           'in,
                           'type,
                           td)) endif endlet endif
```

DEFINITION:

```
range_max_setting(td, min, max)
=   let lo be value_setting(td, min, 'range_minimum),
        hi be value_setting(td, max, 'range_maximum)
    in
    if errorp(hi) then hi
    elseif errorp(lo) then mk_error(marked(lo, unmark(max)))
    else hi endif endlet
```

DEFINITION:

```
range_min_setting(td, min, max)
=   let lo be value_setting(td, min, 'range_minimum),
        hi be value_setting(td, max, 'range_maximum)
    in
```

if errorp (*lo*) **then** *lo*
elseif errorp (*hi*) **then** mk_error (marked (*hi*, unmark (*min*)))
else *lo* **endif endlet**

DEFINITION:
set_range (*td*, *min*, *max*)
= **if** simple_typep (*td*)
then let *lo* **be** range_min_setting (*td*, *min*, *max*),
hi **be** range_max_setting (*td*, *min*, *max*)
in
if (integerp (*lo*) ∧ (¬ ileq (*lo*, *hi*)))
∨ (rationalp (*lo*) ∧ (¬ rleq (*lo*, *hi*)))
then empty_type_error (set_tmin (set_tmax (*td*, *hi*), *lo*))
else subrange_desc (*td*, *lo*, *hi*) **endif endlet**
elseif error_descp (*td*) **then** *td*
else non_simple_subrange_type_error (*td*) **endif**

DEFINITION:
set_default_value (*td*, *dv*)
= **if** (*dv* = **nil**) ∨ error_descp (*td*) **then** *td*
else let *v* **be** value_setting (*td*,
dv,
'default_initial_value)
in
set_udv (*td*, *v*) **endlet endif**

```
; ======================
;  Checking Scalar Types
; ======================
```

EVENT: Disable ref.

EVENT: Disable ref_unit.

EVENT: Disable errorp.

DEFINITION:
scalar_check (*svs*, *sn*, *x*)
= **if** *svs* ≃ **nil then t**
else let *r* **be** ref (car (*svs*), *sn*, *x*)
in
if errorp (ref_unit (*r*)) **then** ref_unit (*r*)
else scalar_check (cdr (*svs*), *sn*, *x*) **endif endlet endif**

DEFINITION:
construct_scalar_desc (*tn*, *sn*, *svs*, *x*)
= **let** *err* **be** scalar_check (*svs*, *sn*, *x*)
in
if errorp (*err*) **then** *err*
else scalar_desc (*tn*, *sn*, length (*svs*)) **endif endlet**

```
; ====================
;  Function Type_Desc
; ====================
```

EVENT: Disable *1*boolean_desc.

EVENT: Disable *1*character_desc.

EVENT: Disable *1*integer_desc.

EVENT: Disable *1*rational_desc.

EVENT: Disable array_desc.

EVENT: Disable available_types.

EVENT: Disable boolean_desc.

EVENT: Disable character_desc.

EVENT: Disable construct_scalar_desc.

EVENT: Disable default_initial_value.

EVENT: Disable error_descp.

EVENT: Disable errorp.

EVENT: Disable gname.

EVENT: Disable identifierp.

EVENT: Disable integer_desc.

EVENT: Disable kind.

EVENT: Disable length.

EVENT: Disable mapping_desc.

EVENT: Disable ncopies.

EVENT: Disable not_record_fields_error.

EVENT: Disable not_type_error.

EVENT: Disable pair_list_map.

EVENT: Disable pending_desc.

EVENT: Disable pending_type_defnp.

EVENT: Disable range_max.

EVENT: Disable range_min.

EVENT: Disable rational_desc.

EVENT: Disable record_desc.

EVENT: Disable record_field_names.

EVENT: Disable ref.

EVENT: Disable ref_scope.

EVENT: Disable ref_unit.

EVENT: Disable scalar_type_defnp.

EVENT: Disable scalar_value_list.

EVENT: Disable sequence_desc.

EVENT: Disable set_default_value.

EVENT: Disable set_desc.

EVENT: Disable set_range.

EVENT: Disable size_limit.

EVENT: Disable type_defn_cycle_error.

EVENT: Disable unit_name.

```
#|
(do-mutual '(


(defn field_descs (s sn ut x)

  (if (rule s (prodn (tag 'fields 'f)
     (list (tag 'fields 'f2) 'SEMI_COLON
   (tag 'similar_fields 's))))
      (let ((f1 (field_descs (subtree s 'fields) sn ut x))
    (f2 (field_descs (subtree s 'similar_fields) sn ut x)))
(if (error_descp f1)
    f1
(if (error_descp f2)
    f2
  (append f1 f2))))

  (if (rule s (prodn (tag 'fields 'f)
     (tag 'similar_fields 's)))
```

```
      (field_descs (subtree s 'similar_fields) sn ut x)

  (if (rule s (prodn (tag 'similar_fields 's)
     (list (tag 'identifier_list 'is) 'COLON
   (tag 'type_specification 'ft))))
      (let ((ftd (type_desc (subtree s 'type_specification) sn ut x))
    (fns (record_field_names (subtree s 'identifier_list))))
(if (error_descp ftd)
    ftd
  (pair_list_map fns (ncopies (length fns) ftd))))

    (not_record_fields_error s sn))))

  ( (ord-lessp (cons (add1 (length (available_types ut x)))
     (tree_size s))) ))


(defn type_desc (s sn ut x)
  ; sn is the name of the scope where s is to be interpreted
  ; s is a type specification parse tree or a parse tree that arises in its
  ;   interpretation
  ; ut (used types) is a list of types that have already been looked up in x
  ; x is the Gypsy parse tree

  (if (rule s (prodn (tag 'array_type 'a)
     (list 'ARRAY 'OPEN_PAREN (tag 'type_specification 'it)
   'CLOSE_PAREN 'OF (tag 'type_specification 'ct))))
      (array_desc (type_desc (subtree_i s 'type_specification 1) sn ut x)
  (type_desc (subtree_i s 'type_specification 2) sn ut x))

  (if (rule s (prodn (tag 'mapping_type 'm)
     (list 'MAPPING (tag 'opt_size_limit_restriction 'r)
   'FROM (tag 'type_specification 'st)
   'TO (tag 'type_specification 'ct))))
      (mapping_desc (size_limit (subtree s 'opt_size_limit_restriction) sn x)
    (type_desc (subtree_i s 'type_specification 1) sn ut x)
    (type_desc (subtree_i s 'type_specification 2) sn ut x))

  (if (rule s (prodn (tag 'record_type 'r)
     (list 'RECORD 'OPEN_PAREN (tag 'fields 'f)
   'CLOSE_PAREN)))
      (record_desc (field_descs (subtree s 'fields) sn ut x))

  (if (rule s (prodn (tag 'sequence_type 's)
```

```
     (list 'SEQUENCE (tag 'opt_size_limit_restriction 'r)
   'OF (tag 'type_specification 'ct))))
      (sequence_desc (size_limit (subtree s 'opt_size_limit_restriction)
 sn x)
     (type_desc (subtree s 'type_specification) sn ut x))

  (if (rule s (prodn (tag 'set_type 's)
     (list 'SET (tag 'opt_size_limit_restriction 'r)
   'OF (tag 'type_specification 'ct))))
      (set_desc (size_limit (subtree s 'opt_size_limit_restriction) sn x)
(type_desc (subtree s 'type_specification) sn ut x))

  (if (rule s (prodn (tag 'type_declaration 'd)
     (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
   (tag 'type_definition 'd2))))
      (if (pending_type_defnp s)
  (pending_desc (unit_name s) sn)
      (if (scalar_type_defnp s)
  (construct_scalar_desc (unit_name s) sn (scalar_value_list s) x)
(type_desc (subtree s 'type_definition) sn ut x)))

  (if (rule s (prodn (tag 'type_definition 'd)
     (tag 'array_type 'a)))
      (type_desc (subtree s 'array_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 'd)
     (tag 'record_type 'r)))
      (type_desc (subtree s 'record_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 'd)
     (tag 'mapping_type 'm)))
      (type_desc (subtree s 'mapping_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 'd)
     (tag 'sequence_type 's)))
      (type_desc (subtree s 'sequence_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 'd)
     (tag 'set_type 's)))
      (type_desc (subtree s 'set_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 's)
     (list (tag 'type_specification 's)
   (tag 'opt_default_initial_value_expression 'v))))
```

```
      (set_default_value (type_desc (subtree s 'type_specification) sn ut x)
 ; set_default_value does not change the default
 ; value if (default_initial_value ...) is nil
 (default_initial_value
   (subtree s 'opt_default_initial_value_expression)
   sn x))

  (if (rule s (prodn (tag 'type_specification 's)
     (tag 'IDENTIFIER 'tn)))
      (type_desc (subtree s 'IDENTIFIER) sn ut x)

  (if (rule s (prodn (tag 'type_specification 's)
     (list (tag 'IDENTIFIER 'tn) (tag 'range 'r))))
      (set_range (type_desc (subtree s 'IDENTIFIER) sn ut x)
 (range_min (subtree s 'range) sn x)
 (range_max (subtree s 'range) sn x))

  (if (identifierp s)
      (let ((tn (gname s)))
(if (equal tn 'boolean)
    (boolean_desc)
(if (equal tn 'character)
    (character_desc)
(if (equal tn 'integer)
    (integer_desc)
(if (equal tn 'rational)
    (rational_desc)
  (let ((r (ref tn sn x)))
  (let ((h (ref_scope r))
(u (ref_unit r)))
    (if (member r ut)
(type_defn_cycle_error tn sn)
    (if (equal (kind u) 'type)
(type_desc u h (cons r ut) x)
    (if (errorp u)
u
      (not_type_error tn sn)))))))))))

    (not_type_error s sn))))))))))))))))

  ( (ord-lessp (cons (add1 (length (available_types ut x)))
     (tree_size s))) ))

))
```

|#

DEFINITION:
mutual-type_desc-field_descs (*mutual-flg*, *s*, *sn*, *ut*, *x*)
= **if** *mutual-flg* = 'type_desc
then if rule (*s*,
prodn (tag ('array_type, 'a),
list ('array,
'open_paren,
tag ('type_specification, 'it),
'close_paren,
'of,
tag ('type_specification, 'ct))))
then array_desc (mutual-type_desc-field_descs ('type_desc,
subtree_i (*s*,
'type_specification,
1),
sn,
ut,
x),
mutual-type_desc-field_descs ('type_desc,
subtree_i (*s*,
'type_specification,
2),
sn,
ut,
x))
elseif rule (*s*,
prodn (tag ('mapping_type, 'm),
list ('mapping,
tag ('opt_size_limit_restriction,
'r),
'from,
tag ('type_specification, 'st),
'to,
tag ('type_specification, 'ct))))
then mapping_desc (size_limit (subtree (*s*,
'opt_size_limit_restriction),
sn,
x),
mutual-type_desc-field_descs ('type_desc,
subtree_i (*s*,
'type_specification,

```
                                                    1),
                                            sn,
                                            ut,
                                            x),
                    mutual-type_desc-field_descs ('type_desc,
                                            subtree_i (s,
                                                    'type_specification,
                                                    2),
                                            sn,
                                            ut,
                                            x))
elseif rule (s,
        prodn (tag ('record_type, 'r),
                list ('record,
                    'open_paren,
                    tag ('fields, 'f),
                    'close_paren)))
then record_desc (mutual-type_desc-field_descs ('field_descs,
                                            subtree (s,
                                                    'fields),
                                            sn,
                                            ut,
                                            x))
elseif rule (s,
        prodn (tag ('sequence_type, 's),
                list ('sequence,
                    tag ('opt_size_limit_restriction,
                        'r),
                    'of,
                    tag ('type_specification, 'ct))))
then sequence_desc (size_limit (subtree (s,
                                    'opt_size_limit_restriction),
                            sn,
                            x),
                    mutual-type_desc-field_descs ('type_desc,
                                            subtree (s,
                                                    'type_specification),
                                            sn,
                                            ut,
                                            x))
elseif rule (s,
        prodn (tag ('set_type, 's),
                list ('set,
                    tag ('opt_size_limit_restriction,
```

```
                    'r),
                'of,
                tag('type_specification, 'ct))))
then set_desc(size_limit(subtree(s,
                        'opt_size_limit_restriction),
                    sn,
                    x),
            mutual-type_desc-field_descs('type_desc,
                                         subtree(s,
                                                 'type_specification),
                                         sn,
                                         ut,
                                         x))
elseif rule(s,
        prodn(tag('type_declaration, 'd),
            list('type,
                tag('identifier, 'tn),
                'equal,
                tag('type_definition, 'd2))))
then if pending_type_defnp(s) then pending_desc(unit_name(s), sn)
     elseif scalar_type_defnp(s)
     then construct_scalar_desc(unit_name(s),
                                sn,
                                scalar_value_list(s),
                                x)
     else mutual-type_desc-field_descs('type_desc,
                                       subtree(s,
                                               'type_definition),
                                       sn,
                                       ut,
                                       x) endif
elseif rule(s,
        prodn(tag('type_definition, 'd),
            tag('array_type, 'a)))
then mutual-type_desc-field_descs('type_desc,
                                  subtree(s, 'array_type),
                                  sn,
                                  ut,
                                  x)
elseif rule(s,
        prodn(tag('type_definition, 'd),
            tag('record_type, 'r)))
then mutual-type_desc-field_descs('type_desc,
                                  subtree(s, 'record_type),
```

```
                                    sn,
                                    ut,
                                    x)
elseif rule (s,
            prodn (tag ('type_definition, 'd),
                   tag ('mapping_type, 'm)))
then mutual-type_desc-field_descs ('type_desc,
                                    subtree (s, 'mapping_type),
                                    sn,
                                    ut,
                                    x)
elseif rule (s,
            prodn (tag ('type_definition, 'd),
                   tag ('sequence_type, 's)))
then mutual-type_desc-field_descs ('type_desc,
                                    subtree (s, 'sequence_type),
                                    sn,
                                    ut,
                                    x)
elseif rule (s,
            prodn (tag ('type_definition, 'd),
                   tag ('set_type, 's)))
then mutual-type_desc-field_descs ('type_desc,
                                    subtree (s, 'set_type),
                                    sn,
                                    ut,
                                    x)
elseif rule (s,
            prodn (tag ('type_definition, 's),
                   list (tag ('type_specification, 's),
                         tag ('opt_default_initial_value_expression,
                              'v))))
then set_default_value (mutual-type_desc-field_descs ('type_desc,
                                                      subtree (s,
                                                               'type_specification),
                                                      sn,
                                                      ut,
                                                      x),
                        default_initial_value (subtree (s,
                                                        'opt_default_initial_value_expression),
                                               sn,
                                               x))
elseif rule (s,
            prodn (tag ('type_specification, 's),
```

```
                    tag ('identifier, 'tn)))
then mutual-type_desc-field_descs ('type_desc,
                                   subtree (s, 'identifier),
                                   sn,
                                   ut,
                                   x)
elseif rule (s,
             prodn (tag ('type_specification, 's),
                    list (tag ('identifier, 'tn),
                          tag ('range, 'r))))
then set_range (mutual-type_desc-field_descs ('type_desc,
                                              subtree (s,
                                                       'identifier),
                                              sn,
                                              ut,
                                              x),
                range_min (subtree (s, 'range), sn, x),
                range_max (subtree (s, 'range), sn, x))
elseif identifierp (s)
then case on gname (s):
     case = boolean
     then BOOLEAN_DESC
     case = character
      then CHARACTER_DESC
     case = integer
      then INTEGER_DESC
     case = rational
      then RATIONAL_DESC
     otherwise if ref (gname (s), sn, x) ∈ ut
               then type_defn_cycle_error (gname (s), sn)
               elseif kind (ref_unit (ref (gname (s), sn, x)))
                      =   'type
               then mutual-type_desc-field_descs ('type_desc,
                                                  ref_unit (ref (gname (s),
                                                                 sn,
                                                                 x)),
                                                  ref_scope (ref (gname (s),
                                                                  sn,
                                                                  x)),
                                                  cons (ref (gname (s),
                                                             sn,
                                                             x),
                                                        ut),
                                                  x)
```

```
                    elseif errorp (ref_unit (ref (gname (s), sn, x)))
                    then ref_unit (ref (gname (s), sn, x))
                    else not_type_error (gname (s), sn) endif endcase
    else not_type_error (s, sn) endif
elseif rule (s,
          prodn (tag ('fields, 'f),
                 list (tag ('fields, 'f2),
                       'semi_colon,
                       tag ('similar_fields, 's))))
then if error_descp (mutual-type_desc-field_descs ('field_descs,
                                                   subtree (s, 'fields),
                                                   sn,
                                                   ut,
                                                   x))
    then mutual-type_desc-field_descs ('field_descs,
                                       subtree (s, 'fields),
                                       sn,
                                       ut,
                                       x)
    elseif error_descp (mutual-type_desc-field_descs ('field_descs,
                                                      subtree (s,
                                                               'similar_fields),
                                                      sn,
                                                      ut,
                                                      x))
    then mutual-type_desc-field_descs ('field_descs,
                                       subtree (s, 'similar_fields),
                                       sn,
                                       ut,
                                       x)
    else append (mutual-type_desc-field_descs ('field_descs,
                                               subtree (s,
                                                        'fields),
                                               sn,
                                               ut,
                                               x),
                 mutual-type_desc-field_descs ('field_descs,
                                               subtree (s,
                                                        'similar_fields),
                                               sn,
                                               ut,
                                               x)) endif
elseif rule (s, prodn (tag ('fields, 'f), tag ('similar_fields, 's)))
then mutual-type_desc-field_descs ('field_descs,
```

```
                                    subtree (s, 'similar_fields),
                                    sn,
                                    ut,
                                    x)
    elseif rule (s,
                prodn (tag ('similar_fields, 's),
                       list (tag ('identifier_list, 'is),
                             'colon,
                             tag ('type_specification, 'ft))))
    then if error_descp (mutual-type_desc-field_descs ('type_desc,
                                                       subtree (s,
                                                                'type_specification),
                                                       sn,
                                                       ut,
                                                       x))
         then mutual-type_desc-field_descs ('type_desc,
                                            subtree (s,
                                                     'type_specification),
                                            sn,
                                            ut,
                                            x)
         else pair_list_map (record_field_names (subtree (s,
                                                          'identifier_list)),
                             ncopies (length (record_field_names (subtree (s,
                                                                           'identifier_list))),
                                      mutual-type_desc-field_descs ('type_desc,
                                                                    subtree (s,
                                                                             'type_specification),
                                                                    sn,
                                                                    ut,
                                                                    x))) endif
    else not_record_fields_error (s, sn) endif
```

DEFINITION:
type_desc (*s*, *sn*, *ut*, *x*)
= mutual-type_desc-field_descs ('type_desc, *s*, *sn*, *ut*, *x*)

DEFINITION:
field_descs (*s*, *sn*, *ut*, *x*)
= mutual-type_desc-field_descs ('field_descs, *s*, *sn*, *ut*, *x*)

```
; ****************
;  Literal Values
; ****************
```

```
; ----------------
;  Boolean Values
; ----------------
```

DEFINITION: GFALSE = marked (**nil**, typed (BOOLEAN_DESC, 0))

DEFINITION: GTRUE = marked (**nil**, typed (BOOLEAN_DESC, 1))

DEFINITION:
gtruep (v)
= (determinate (v)
$\wedge$ truep (in_type (BOOLEAN_DESC, v))
$\wedge$ (value (v) = value (GTRUE)))

```
; ------------------
;  Character Values
; ------------------
```

DEFINITION:
gchar (c)
= **if** character_valuep (c)
then marked (**nil**, typed (CHARACTER_DESC, cadr (lexeme (c))))
else marked (character_error (c), default_value (CHARACTER_DESC)) **endif**

```
; -------------------------------
;  Numbers - Integer and Rational
; -------------------------------
```

DEFINITION: GIZERO = marked (**nil**, typed (INTEGER_DESC, 0))

DEFINITION:
GRZERO = marked (**nil**, typed (RATIONAL_DESC, rational (0, 1)))

DEFINITION: GIONE = marked (**nil**, typed (INTEGER_DESC, 1))

DEFINITION: GITWO = marked (**nil**, typed (INTEGER_DESC, 2))

DEFINITION:
char_digit (c)
= **if** (ASCII_0 $\leq c$) $\wedge$ ($c \leq$ ASCII_9) **then** $c -$ ASCII_0
elseif (ASCII_A $\leq c$) $\wedge$ ($c \leq$ ASCII_F)
then ($c -$ ASCII_A) + 10
elseif (ASCII_LC_A $\leq c$) $\wedge$ ($c \leq$ ASCII_LC_F)
then ($c -$ ASCII_LC_A) + 10
else nil endif

DEFINITION:
digit_valid (d, b)
= **let** n **be** char_digit (d)
in
$(n \in \mathbf{N}) \wedge (0 \leq n) \wedge (n < b)$ **endlet**

EVENT: Disable digit_valid.

EVENT: Disable char_digit.

DEFINITION:
digit_value (s, b)
= **if** $s \simeq$ **nil** **then** 0
elseif digit_value (rcdr (s), b) = 'ind **then** 'ind
elseif digit_valid (rcar (s), b)
then (digit_value (rcdr (s), b) $*$ b) + char_digit (rcar (s))
else 'ind **endif**

DEFINITION:
tdigit_value (s, b)
= **if** digit_listp (s) **then** digit_value (lexeme (s), b)
else 'ind **endif**

DEFINITION:
mdigit_value (s, b)
= **if** tdigit_value (s, b) = 'ind
then marked (number_error (s, b), default_value (INTEGER_DESC))
else marked (**nil**, typed (INTEGER_DESC, tdigit_value (s, b))) **endif**

DEFINITION:
minteger (i)
= **if** rule (i, prodn (tag ('number, 'n), tag ('digit_list, 's)))
then mdigit_value (subtree (i, 'digit_list), 10)
elseif digit_listp (i) **then** mdigit_value (i, 10)
elseif rule (i,
prodn (tag ('number, 'n),
list (tag ('base, 'b), tag ('digit_list, 's))))
then mdigit_value (subtree (i, 'digit_list), ibase (subtree (i, 'base)))
else marked (number_error (i, 10), default_value (INTEGER_DESC)) **endif**

```
; ---------------
;  String Values
; ---------------
```

DEFINITION:
string_char_seq (*s*)
= **if** *s* ≃ **nil** **then** **nil**
elseif car (*s*) = ASCII_DOUBLE_QUOTE
then cons (car (*s*), string_char_seq (cddr (*s*)))
else cons (car (*s*), string_char_seq (cdr (*s*))) **endif**

DEFINITION:
gstring_seq (*s*)
= **if** string_valuep (*s*)
then marked (**nil**,
typed (sequence_desc (**nil**, CHARACTER_DESC),
string_char_seq (rcdr (cdr (lexeme (*s*))))))
else marked (bad_string_error (*s*),
default_value (sequence_desc (**nil**, CHARACTER_DESC))) **endif**

```
; ************************************
;  Arguments to Functions and Operations
; ************************************
```

DEFINITION:
in_arg_type (*td*, *a*)
= **if** *td* = '`field_name`
then (mark (*a*) = *td*) ∧ litatom (unmark (*a*))
elseif *td* = '`type_descriptor`
then (mark (*a*) = *td*) ∧ type_descp (unmark (*a*))
else in_type (*td*, *a*) **endif**

DEFINITION:
arg_check (*as*, *ts*)
= **if** *as* ≃ **nil**
then if *ts* ≃ **nil** **then** **nil**
else mk_error ('`(number of formal and actual parameters not equal)`) **endif**
elseif *ts* ≃ **nil**
then mk_error ('`(number of formal and actual parameters not equal)`)
elseif indeterminate (car (*as*)) **then** mark (car (*as*))
else **let** *r* **be** in_arg_type (car (*ts*), car (*as*))
in
if truep (*r*) **then** arg_check (cdr (*as*), cdr (*ts*))
elseif errorp (*r*) **then** *r*
else actual_formal_type_error (car (*as*), car (*ts*)) **endif** **endlet** **endif**

```
; ******************
;  Gypsy Operations
; ******************

; ----------------------
;  Component Selection
; ----------------------
```

DEFINITION:
array_get (a, i, td)
= **let** *arg_err* **be** arg_check (list (a, i), list (td, selector_td (td)))
in
if *arg_err* = **nil**
then if array_descp (td)
then marked (**nil**,
typed (component_td (td),
vselect (value (a), value (i), td)))
else marked (not_array_error (a),
default_value (INTEGER_DESC)) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
mapping_get (m, d, td)
= **let** *arg_err* **be** arg_check (list (m, d), list (td, selector_td (td)))
in
if *arg_err* = **nil**
then if mapping_descp (td)
then let v **be** vselect (value (m), value (d), td)
in
if errorp (v)
then marked (no_such_component_error (m, d),
default_value (component_td (td)))
else marked (**nil**,
typed (component_td (td), v)) **endif endlet**
else marked (not_mapping_error (m),
default_value (INTEGER_DESC)) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
record_get (r, *fn*, td)
= **let** *arg_err* **be** arg_check (list (r, *fn*), list (td, `'field_name`))
in
if *arg_err* = **nil**
then if record_descp (td)

```
        then let fd be field_td (unmark (fn), td)
             in
             if fd = nil
             then marked (no_such_field_error (r, unmark (fn)),
                          default_value (INTEGER_DESC))
             else marked (nil,
                          typed (fd,
                                 fselect (value (r),
                                          unmark (fn)))) endif endlet
        else marked (not_record_error (r),
                     default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
sequence_get (s, i, td)
=   let arg_err be arg_check (list (s, i), list (td, INTEGER_DESC))
    in
    if arg_err = nil
    then if sequence_descp (td)
         then let v be vselect (value (s), value (i), td)
              in
              if errorp (v)
              then marked (no_such_component_error (s, i),
                           default_value (component_td (td)))
              else marked (nil,
                           typed (component_td (td), v)) endif endlet
         else marked (not_sequence_error (s),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet
```

EVENT: Disable type.

EVENT: Disable array_get.

EVENT: Disable record_get.

EVENT: Disable mapping_get.

EVENT: Disable sequence_get.

EVENT: Disable not_selectable_error.

EVENT: Disable default_value.

DEFINITION:
select_op (v, s)
= **if** $s \simeq$ **nil** **then** v
else case on root (type (v)):
case = *array*
then select_op (array_get (v, car (s), type (v)), cdr (s))
case = *record*
then select_op (record_get (v, car (s), type (v)), cdr (s))
case = *mapping*
then select_op (mapping_get (v, car (s), type (v)), cdr (s))
case = *sequence*
then select_op (sequence_get (v, car (s), type (v)), cdr (s))
otherwise marked (not_selectable_error (v),
default_value (INTEGER_DESC)) **endcase endif**

```
; ---------------------------
;  Sequence/Set Constructors
; ---------------------------
```

DEFINITION:
gseq (es, td)
= **let** *arg_err* **be** arg_check (es, ncopies (length (es), td))
in
if *arg_err* = **nil**
then marked (**nil**, typed (sequence_desc (**nil**, td), values (es)))
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
gset (es, td)
= **let** *arg_err* **be** arg_check (es, ncopies (length (es), td))
in
if *arg_err* = **nil**
then marked (**nil**, typed (set_desc (**nil**, td), vset (values (es), td)))
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
grange_elements (lo, hi)
= **let** *arg_err* **be** arg_check (list (lo, hi),
list (base_type (type (lo)),
base_type (type (lo))))
in

if *arg_err* = **nil**
then if non_rational_simple_typep (type (*lo*))
then marked_typed_list (base_type (type (*lo*)),
number_list (value (*lo*), value (*hi*)))
else list (marked (range_limits_error (*lo*, *hi*),
default_value (base_type (type (*lo*))))) **endif**
else list (marked (*arg_err*,
default_value (base_type (type (*lo*))))) **endif endlet**

DEFINITION:
gset_or_seq (*m*, *a*, *td*)
= **if** rule (*m*, prodn (tag (`'set_or_seq_mark`, `'m`), list (`'set`, `'colon`)))
then gset (*a*, *td*)
elseif rule (*m*,
prodn (tag (`'set_or_seq_mark`, `'m`), list (`'seq`, `'colon`)))
then gseq (*a*, *td*)
else gseq (*a*, *td*) **endif**

```
; ------------------------------------
;  Interface to Boyer-Moore Functions
; ------------------------------------
```

DEFINITION:
t_or_f (*x*)
= **if** *x* ≃ `0` **then f**
else t endif

DEFINITION: band (*x*, *y*) = (t_or_f (*x*) ∧ t_or_f (*y*))

DEFINITION: bimp (*x*, *y*) = (t_or_f (*x*) → t_or_f (*y*))

DEFINITION: bnot (*x*) = (¬ t_or_f (*x*))

DEFINITION: bor (*x*, *y*) = (t_or_f (*x*) ∨ t_or_f (*y*))

```
; -----------------
;  Unary Operators
; -----------------
```

DEFINITION:
gminus (*u*)
= **let** *iarg_err* **be** arg_check (list (*u*), list (INTEGER_DESC)),
rarg_err **be** arg_check (list (*u*), list (RATIONAL_DESC))
in

```
    if iarg_err = nil
    then marked (nil, typed (INTEGER_DESC, ineg (value (u))))
    elseif rarg_err = nil
    then marked (nil, typed (RATIONAL_DESC, rneg (value (u))))
    else marked (mk_error (list (iarg_err, rarg_err)),
                 default_value (INTEGER_DESC)) endif endlet

DEFINITION:
gnot (u)
=   let arg_err be arg_check (list (u), list (BOOLEAN_DESC))
    in
    if arg_err = nil
    then if bnot (value (u)) then GTRUE
         else GFALSE endif
    else marked (arg_err, default_value (BOOLEAN_DESC)) endif endlet

DEFINITION:
apply_unary_op (op, v)
=   if rule (op, prodn (tag ('unary_operator, 'op), 'minus))
    then gminus (v)
    elseif rule (op, prodn (tag ('unary_operator, 'op), 'not))
    then gnot (v)
    else marked (not_unary_op_error (op), default_value (INTEGER_DESC)) endif

; ------------------
;  Binary Operators
; ------------------


DEFINITION:
gequal (v1, v2)
=   if equality_typep (type (v1))
    then let td be base_type (type (v1))
         in
         let arg_err be arg_check (list (v1, v2), list (td, td))
         in
         if arg_err = nil
         then if vequal (value (v1), value (v2), td) then GTRUE
              else GFALSE endif
         else marked (arg_err,
                      default_value (BOOLEAN_DESC)) endif endlet endlet
    else marked (not_equality_type_error (type (v1)),
                 default_value (BOOLEAN_DESC)) endif

DEFINITION:  gne (v1, v2) = gnot (gequal (v1, v2))
```

```
DEFINITION:
glt (v1, v2)
=  let td be base_type (type (v1))
   in
   let arg_err be arg_check (list (v1, v2), list (td, td))
   in
   if arg_err = nil
   then if simple_typep (td)
        then let r be if root (td) = 'rational
                      then rlessp (value (v1), value (v2))
                      else ilessp (value (v1),
                                   value (v2)) endif
             in
             if truep (r) then GTRUE
             else GFALSE endif endlet
        else marked (not_defined_on_type_error ('lt,
                                                type (v1)),
                     default_value (BOOLEAN_DESC)) endif
   else marked (arg_err, default_value (BOOLEAN_DESC)) endif endlet endlet

DEFINITION:
gor (v1, v2)
=  let arg_err be arg_check (list (v1, v2),
                             list (BOOLEAN_DESC, BOOLEAN_DESC))
   in
   if arg_err = nil
   then if bor (value (v1), value (v2)) then GTRUE
        else GFALSE endif
   else marked (arg_err, default_value (BOOLEAN_DESC)) endif endlet

DEFINITION:  gle (v1, v2) = gor (glt (v1, v2), gequal (v1, v2))

DEFINITION:  ggt (v1, v2) = glt (v2, v1)

DEFINITION:  gge (v1, v2) = gor (ggt (v1, v2), gequal (v1, v2))

DEFINITION:
gand (v1, v2)
=  let arg_err be arg_check (list (v1, v2),
                             list (BOOLEAN_DESC, BOOLEAN_DESC))
   in
   if arg_err = nil
   then if band (value (v1), value (v2)) then GTRUE
        else GFALSE endif
   else marked (arg_err, default_value (BOOLEAN_DESC)) endif endlet
```

```
DEFINITION:
gimp (v1, v2)
=  let arg_err be arg_check (list (v1, v2),
                             list (BOOLEAN_DESC, BOOLEAN_DESC))
   in
   if arg_err = nil
   then if bimp (value (v1), value (v2)) then GTRUE
        else GFALSE endif
   else marked (arg_err, default_value (BOOLEAN_DESC)) endif endlet

DEFINITION:
giff (v1, v2)
=  let arg_err be arg_check (list (v1, v2),
                             list (BOOLEAN_DESC, BOOLEAN_DESC))
   in
   if arg_err = nil then gequal (v1, v2)
   else marked (arg_err, default_value (BOOLEAN_DESC)) endif endlet

DEFINITION:
gpower (v1, v2)
=  let iarg_err be arg_check (list (v1, v2),
                              list (INTEGER_DESC, INTEGER_DESC)),
       rarg_err be arg_check (list (v1, v2),
                              list (RATIONAL_DESC, INTEGER_DESC))
   in
   if iarg_err = nil
   then if gtruep (glt (v2, GIZERO))
        then marked (NEGATIVE_EXPONENT_ERROR,
                     default_value (INTEGER_DESC))
        elseif gtruep (gand (gequal (v1, GIZERO), gequal (v2, GIZERO)))
        then marked (ZERO_TO_THE_ZERO_POWER_ERROR,
                     default_value (INTEGER_DESC))
        else marked (nil,
                     typed (INTEGER_DESC,
                            ipower (value (v1), value (v2)))) endif
   elseif rarg_err = nil
   then if gtruep (glt (v2, GIZERO))
        then marked (NEGATIVE_EXPONENT_ERROR,
                     default_value (RATIONAL_DESC))
        elseif gtruep (gand (gequal (v1, GRZERO), gequal (v2, GIZERO)))
        then marked (ZERO_TO_THE_ZERO_POWER_ERROR,
                     default_value (RATIONAL_DESC))
        else marked (nil,
                     typed (RATIONAL_DESC,
```

```
                                rpower (value (v1), value (v2)))) endif
    else marked (mk_error (list (iarg_err, rarg_err)),
                 default_value (INTEGER_DESC)) endif endlet

DEFINITION:
gtimes (v1, v2)
=   let iarg_err be arg_check (list (v1, v2),
                               list (INTEGER_DESC, INTEGER_DESC)),
        rarg_err be arg_check (list (v1, v2),
                               list (RATIONAL_DESC, RATIONAL_DESC))
    in
    if iarg_err = nil
    then marked (nil,
                 typed (INTEGER_DESC, itimes (value (v1), value (v2))))
    elseif rarg_err = nil
    then marked (nil,
                 typed (RATIONAL_DESC, rtimes (value (v1), value (v2))))
    else marked (mk_error (list (iarg_err, rarg_err)),
                 default_value (INTEGER_DESC)) endif endlet

DEFINITION:
gquotient (v1, v2)
=   let iarg_err be arg_check (list (v1, v2),
                               list (INTEGER_DESC, INTEGER_DESC)),
        rarg_err be arg_check (list (v1, v2),
                               list (RATIONAL_DESC, RATIONAL_DESC))
    in
    if iarg_err = nil
    then if izerop (value (v2))
         then marked (ZERO_DIVIDE_ERROR,
                      default_value (RATIONAL_DESC))
         else marked (nil,
                      typed (RATIONAL_DESC,
                             reduce (rational (value (v1),
                                               value (v2))))) endif
    elseif rarg_err = nil
    then if rzerop (value (v2))
         then marked (ZERO_DIVIDE_ERROR,
                      default_value (RATIONAL_DESC))
         else marked (nil,
                      typed (RATIONAL_DESC,
                             rquotient (value (v1), value (v2)))) endif
    else marked (mk_error (list (iarg_err, rarg_err)),
                 default_value (RATIONAL_DESC)) endif endlet
```

DEFINITION:
gdiv (*v1*, *v2*)
= **let** *arg_err* **be** arg_check (list (*v1*, *v2*),
list (INTEGER_DESC, INTEGER_DESC))
in
if *arg_err* = **nil**
then if izerop (value (*v2*))
then marked (ZERO_DIVIDE_ERROR, default_value (INTEGER_DESC))
else marked (**nil**,
typed (INTEGER_DESC,
iquotient (value (*v1*), value (*v2*)))) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
gmod (*v1*, *v2*)
= **let** *arg_err* **be** arg_check (list (*v1*, *v2*),
list (INTEGER_DESC, INTEGER_DESC))
in
if *arg_err* = **nil**
then if izerop (value (*v2*))
then marked (ZERO_DIVIDE_ERROR, default_value (INTEGER_DESC))
else marked (**nil**,
typed (INTEGER_DESC,
iremainder (value (*v1*), value (*v2*)))) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
gplus (*v1*, *v2*)
= **let** *iarg_err* **be** arg_check (list (*v1*, *v2*),
list (INTEGER_DESC, INTEGER_DESC)),
rarg_err **be** arg_check (list (*v1*, *v2*),
list (RATIONAL_DESC, RATIONAL_DESC))
in
if *iarg_err* = **nil**
then marked (**nil**, typed (INTEGER_DESC, iplus (value (*v1*), value (*v2*))))
elseif *rarg_err* = **nil**
then marked (**nil**,
typed (RATIONAL_DESC, rplus (value (*v1*), value (*v2*))))
else marked (mk_error (list (*iarg_err*, *rarg_err*)),
default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
gsubtract (*v1*, *v2*)
= **let** *iarg_err* **be** arg_check (list (*v1*, *v2*),
list (INTEGER_DESC, INTEGER_DESC)),

```
        rarg_err be arg_check (list (v1, v2),
                               list (RATIONAL_DESC, RATIONAL_DESC))
    in
    if iarg_err = nil
    then marked (nil,
                 typed (INTEGER_DESC,
                        idifference (value (v1), value (v2))))
    elseif rarg_err = nil
    then marked (nil,
                 typed (RATIONAL_DESC,
                        rdifference (value (v1), value (v2))))
    else marked (mk_error (list (iarg_err, rarg_err)),
                 default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
gin (v1, v2)
=   let td be base_type (type (v2))
    in
    if sequence_descp (td) ∨ set_descp (td)
    then let arg_err be arg_check (list (v1, v2),
                                   list (component_td (td), td))
         in
         if arg_err = nil
         then if vmember (value (v1),
                          value (v2),
                          component_td (td)) then GTRUE
              else GFALSE endif
         else marked (arg_err,
                      default_value (BOOLEAN_DESC)) endif endlet
    else marked (in_arg_error (v2), default_value (BOOLEAN_DESC)) endif endlet
```

DEFINITION:

```
mapping_merge_arg_check2 (ks, v1, v2)
=   if ks ≃ nil then t
    else let c1 be mapping_get (v1, car (ks), base_type (type (v1))),
             c2 be mapping_get (v2, car (ks), base_type (type (v2)))
         in
         if determinate (c1) ∧ determinate (c2)
         then gtruep (gequal (c1, c2))
              ∧   mapping_merge_arg_check2 (cdr (ks), v1, v2)
         else mapping_merge_arg_check2 (cdr (ks), v1, v2) endif endlet endif
```

DEFINITION:

```
mapping_merge_arg_check (v1, v2)
=   if mapping_descp (type (v1)) ∧ mapping_descp (type (v2))
```

```
    then let ks be marked_typed_list (selector_td (base_type (type (v1))),
                                                  vdomain (value (v1)))
         in
         mapping_merge_arg_check2 (ks, v1, v2) endlet
    else f endif
```

DEFINITION:

```
gunion (v1, v2)
=   let td be base_type (type (v1))
    in
    let arg_err be arg_check (list (v1, v2), list (td, td))
    in
    if arg_err = nil
    then if set_descp (td)
         then marked (nil,
                      typed (td,
                             vunion (value (v1), value (v2), td)))
         elseif mapping_descp (td)
         then if mapping_merge_arg_check (v1, v2)
              then marked (nil,
                           typed (td,
                                  vunion_maps (value (v1),
                                               value (v2),
                                               td)))
              else marked (mapping_merge_error (v1, v2),
                           default_value (td)) endif
         else marked (union_arg_error (v1, v2),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
gadjoin (v1, v2)
=   if set_descp (type (v1))
    then gunion (v1, gset (list (v2), base_type (type (v2))))
    else marked (adjoin_args_error (v1), default_value (INTEGER_DESC)) endif
```

DEFINITION:

```
gomit (v1, v2)
=   let td be base_type (type (v1))
    in
    if set_descp (td)
    then let arg_err be arg_check (list (v1, v2),
                                   list (td, component_td (td)))
         in
         if arg_err = nil
```

```
        then if gtruep (gin (v2, v1))
                then marked (nil,
                             typed (td,
                                    vremove (value (v2),
                                             value (v1),
                                             component_td (td))))
                else marked (not_in_set_error (v2, v1),
                             default_value (td)) endif
        else marked (arg_err,
                     default_value (INTEGER_DESC)) endif endlet
    else marked (omit_args_error (v1),
                 default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
gsub (v1, v2)
=   let td be base_type (type (v1))
    in
    let arg_err be arg_check (list (v1, v2), list (td, td))
    in
    if arg_err = nil
    then if mapping_descp (td)
            ∨   sequence_descp (td)
            ∨   set_descp (td)
         then if vsubp (value (v1), value (v2), td) then GTRUE
              else GFALSE endif
         else marked (sub_args_error (v1, v2),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
gintersect (v1, v2)
=   let td be base_type (type (v1))
    in
    let arg_err be arg_check (list (v1, v2), list (td, td))
    in
    if arg_err = nil
    then if set_descp (td)
         then marked (nil,
                      typed (td,
                             vintersect (value (v1),
                                         value (v2),
                                         td)))
         elseif mapping_descp (td)
         then if mapping_merge_arg_check (v1, v2)
```

```
        then marked (nil,
                     typed (td,
                            vintersect_maps (value (v1),
                                             value (v2),
                                             td)))
        else marked (mapping_merge_error (v1, v2),
                     default_value (td)) endif
    else marked (intersect_args_error (v1, v2),
                 default_value (INTEGER_DESC)) endif
else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
gdifference (v1, v2)
=   let td be base_type (type (v1))
    in
    let arg_err be arg_check (list (v1, v2), list (td, td))
    in
    if arg_err = nil
    then if set_descp (td)
         then marked (nil,
                      typed (td,
                             vdifference (value (v1),
                                          value (v2),
                                          td)))
         elseif mapping_descp (td)
         then if mapping_merge_arg_check (v1, v2)
              then marked (nil,
                           typed (td,
                                  vdifference_maps (value (v1),
                                                    value (v2),
                                                    td)))
              else marked (mapping_merge_error (v1, v2),
                           default_value (td)) endif
         else marked (difference_args_error (v1, v2),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
gappend (v1, v2)
=   let td be base_type (type (v1))
    in
    let arg_err be arg_check (list (v1, v2), list (td, td))
    in
    if arg_err = nil
```

then if sequence_descp (*td*)
then marked (**nil**,
typed (*td*, append (value (*v1*), value (*v2*))))
else marked (append_args_error (*v1*, *v2*),
default_value (INTEGER_DESC)) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet endlet**

DEFINITION:
gcons (*v1*, *v2*) = gappend (gseq (list (*v1*), base_type (type (*v1*))), *v2*)

DEFINITION:
grcons (*v1*, *v2*) = gappend (*v1*, gseq (list (*v2*), base_type (type (*v2*))))

DEFINITION:
apply_binary_op (*op*, *v1*, *v2*)
= **if** rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'eq`))
∨ rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'equal`))
then gequal (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'ne`))
then gne (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'lt`))
then glt (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'le`))
then gle (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'gt`))
then ggt (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'ge`))
then gge (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'and`))
then gand (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'or`))
then gor (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'imp`))
then gimp (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'iff`))
then giff (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'star_star`))
then gpower (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'star`))
then gtimes (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'slash`))
then gquotient (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'div`))
then gdiv (*v1*, *v2*)
elseif rule (*op*, prodn (tag (`'binary_operator`, `'op`), `'mod`))

```
    then gmod (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'plus))
    then gplus (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'minus))
    then gsubtract (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'in))
    then gin (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'adjoin))
    then gadjoin (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'omit))
    then gomit (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'sub))
    then gsub (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'union))
    then gunion (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'intersect))
    then gintersect (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'difference))
    then gdifference (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'colon_gt))
    then gcons (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'lt_colon))
    then grcons (v1, v2)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'append))
    then gappend (v1, v2)
    else marked (not_binary_op_error (op), default_value (INTEGER_DESC)) endif

; ------------------------
;  Subsequence Selection
; ------------------------


DEFINITION:
subsequence_get (s, lo, hi)
=   let arg_err  be  arg_check (list (s, lo, hi),
                                list (type (s), INTEGER_DESC, INTEGER_DESC))
    in
    if arg_err = nil
    then if sequence_descp (type (s))
         then if gtruep (gle (lo, hi))
              then if indeterminate (sequence_get (s, lo, type (s)))
                   then marked (no_such_component_error (s, lo),
                                default_value (type (s)))
                   elseif indeterminate (sequence_get (s, hi, type (s)))
```

```
                then marked (no_such_component_error (s, hi),
                             default_value (type (s)))
                else marked (nil,
                             typed (type (s),
                                    vsubseq_select (value (s),
                                                    value (lo),
                                                    value (hi)))) endif
            else marked (nil, typed (type (s), NULL_SEQ)) endif
        else marked (not_sequence_error (s),
                     default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet

; ------------------
;  Value Alteration
; ------------------
```

DEFINITION:

```
array_put (a, i, v)
=   let td  be  base_type (type (a))
    in
    let arg_err  be  arg_check (list (a, i, v),
                                list (td,
                                      selector_td (td),
                                      component_td (td)))
    in
    if arg_err = nil
    then if array_descp (td)
         then marked (nil,
                      typed (td,
                             varray_put (value (a),
                                         value (i),
                                         value (v),
                                         td)))
         else marked (not_array_error (a),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
record_put (r, fn, v)
=   let td  be  base_type (type (r))
    in
    let arg_err  be  arg_check (list (r, fn, v),
                                list (td,
                                      'field_name,
```

```
                                    field_td (unmark (fn), td)))
    in
    if arg_err = nil
    then if record_descp (td)
         then marked (nil,
                      typed (td,
                             vrecord_put (value (r),
                                          unmark (fn),
                                          value (v))))
         else marked (not_record_error (r),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
mapping_put (m, d, v)
=   let td be base_type (type (m))
    in
    let arg_err be arg_check (list (m, d, v),
                              list (td,
                                    selector_td (td),
                                    component_td (td)))
    in
    if arg_err = nil
    then if mapping_descp (td)
         then if determinate (mapping_get (m, d, type (m)))
              then marked (nil,
                           typed (td,
                                  vmapping_put (value (m),
                                                value (d),
                                                value (v),
                                                td)))
              else marked (no_such_component_error (m, d),
                           default_value (td)) endif
         else marked (not_mapping_error (m),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
sequence_put (s, i, v)
=   let td be base_type (type (s))
    in
    let arg_err be arg_check (list (s, i, v),
                              list (td,
                                    INTEGER_DESC,
```

```
                                    component_td (td)))
    in
    if arg_err = nil
    then if sequence_descp (td)
         then if determinate (sequence_get (s, i, type (s)))
              then marked (nil,
                           typed (td,
                                  vsequence_put (value (s),
                                                 value (i),
                                                 value (v))))
              else marked (no_such_component_error (s, i),
                           default_value (td)) endif
         else marked (not_sequence_error (s),
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
put_op (bv, s, v)
=   if s ≃ nil then v
    else case on root (type (bv)):
         case = array
         then array_put (bv,
                         car (s),
                         put_op (array_get (bv, car (s), type (bv)),
                                 cdr (s),
                                 v))
         case = record
          then record_put (bv,
                           car (s),
                           put_op (record_get (bv, car (s), type (bv)),
                                   cdr (s),
                                   v))
         case = mapping
          then mapping_put (bv,
                            car (s),
                            put_op (mapping_get (bv, car (s), type (bv)),
                                    cdr (s),
                                    v))
         case = sequence
          then sequence_put (bv,
                             car (s),
                             put_op (sequence_get (bv, car (s), type (bv)),
                                     cdr (s),
                                     v))
```

```
        otherwise marked (component_assign_error (bv),
                          default_value (INTEGER_DESC)) endcase endif
```

DEFINITION:

```
gmapomit (m, i)
=   if determinate (mapping_get (m, i, type (m)))
    then marked (nil, typed (type (m), vmap_remove (value (m), value (i), type (m))))
    else marked (mark (mapping_get (m, i, type (m))),
                 default_value (type (m))) endif
```

DEFINITION:

```
gseqomit (s, i)
=   if determinate (sequence_get (s, i, type (s)))
    then let s2 be gappend (subsequence_get (s, GIONE, gsubtract (i, GIONE)),
                            subsequence_get (s,
                                             gplus (i, GIONE),
                                             marked (nil,
                                                     typed (INTEGER_DESC,
                                                            vsize (value (s))))))
         in
         marked (mark (s2), typed (type (s), value (s2))) endlet
    else marked (mark (sequence_get (s, i, type (s))),
                 default_value (type (s))) endif
```

DEFINITION:

```
gmap_insert (m, d, v)
=   let td be base_type (type (m))
    in
    let arg_err be arg_check (list (m, d, v),
                              list (td,
                                    selector_td (td),
                                    component_td (td)))
    in
    if arg_err = nil
    then if mapping_descp (td)
         then if determinate (mapping_get (m, d, type (m)))
              then mapping_put (m, d, v)
              else marked (nil,
                           typed (td,
                                  vmapping_put (value (m),
                                                value (d),
                                                value (v),
                                                td))) endif
         else marked (not_mapping_error (m),
                      default_value (INTEGER_DESC)) endif
```

else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet endlet**

DEFINITION:
gseq_insert_before (*s*, *i*, *v*)
= **if** determinate (sequence_get (*s*, *i*, type (*s*)))
then gappend (subsequence_get (*s*, GIONE, gsubtract (*i*, GIONE)),
gappend (gseq (list (*v*), base_type (type (*v*))),
subsequence_get (*s*,
i,
marked (**nil**,
typed (INTEGER_DESC,
vsize (value (*s*)))))))
else marked (mark (sequence_get (*s*, *i*, type (*s*))),
default_value (base_type (type (*s*)))) **endif**

DEFINITION:
gseq_insert_behind (*s*, *i*, *v*)
= **if** determinate (sequence_get (*s*, *i*, type (*s*)))
then gappend (subsequence_get (*s*, GIONE, *i*),
gappend (gseq (list (*v*), base_type (type (*v*))),
subsequence_get (*s*,
gplus (*i*, GIONE),
marked (**nil**,
typed (INTEGER_DESC,
vsize (value (*s*)))))))
else marked (mark (sequence_get (*s*, *i*, type (*s*))),
default_value (base_type (type (*s*)))) **endif**

```
; ********************
;  Standard Functions
; ********************
```

DEFINITION:
std_domain (*d*)
= **let** *arg_err* **be** arg_check (*d*, list (type (car (*d*))))
in
if *arg_err* = **nil**
then if mapping_descp (type (car (*d*)))
then marked (**nil**,
typed (set_desc (**nil**, selector_td (type (car (*d*)))),
vdomain (value (car (*d*)))))
else marked (DOMAIN_ARG_ERROR,
default_value (INTEGER_DESC)) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
std_first (*d*)
= **let** *arg_err* **be** arg_check (*d*, list (type (car (*d*))))
in
if *arg_err* = **nil**
then if sequence_descp (type (car (*d*)))
then sequence_get (car (*d*), GIONE, type (car (*d*)))
else marked (FIRST_ARG_ERROR,
default_value (INTEGER_DESC)) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
std_initial (*d*)
= **let** *arg_err* **be** arg_check (*d*, list (`'type_descriptor`))
in
if *arg_err* = **nil**
then let *td* **be** unmark (car (*d*))
in
if errorp (udv (*td*))
then marked (udv (*td*), default_value (*td*))
else marked (**nil**, typed (*td*, udv (*td*))) **endif endlet**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
std_last (*d*)
= **let** *arg_err* **be** arg_check (*d*, list (type (car (*d*))))
in
if *arg_err* = **nil**
then if sequence_descp (type (car (*d*)))
then sequence_get (car (*d*),
marked (**nil**,
typed (INTEGER_DESC, vsize (car (*d*)))),
type (car (*d*)))
else marked (LAST_ARG_ERROR,
default_value (INTEGER_DESC)) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
std_lower (*d*)
= **let** *arg_err* **be** arg_check (*d*, list (`'type_descriptor`))
in
if *arg_err* = **nil**
then let *td* **be** unmark (car (*d*))
in
let *r* **be** tmin (*td*)

```
        in
        if r = nil
        then marked (unbounded_type_error (td),
                     default_value (td))
        elseif errorp (r)
        then marked (r, default_value (td))
        else marked (nil, typed (td, r)) endif endlet endlet
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
std_max (d)
=   let td be base_type (type (car (d)))
    in
    let arg_err be arg_check (d, list (td, td))
    in
    if arg_err = nil
    then if simple_typep (td)
         then if gtruep (ggt (car (d), cadr (d)))
              then marked (nil, typed (td, value (car (d))))
              else marked (nil,
                           typed (td, value (cadr (d)))) endif
         else marked (MAX_ARG_ERROR,
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
std_min (d)
=   let td be base_type (type (car (d)))
    in
    let arg_err be arg_check (d, list (td, td))
    in
    if arg_err = nil
    then if simple_typep (td)
         then if gtruep (glt (car (d), cadr (d)))
              then marked (nil, typed (td, value (car (d))))
              else marked (nil,
                           typed (td, value (cadr (d)))) endif
         else marked (MIN_ARG_ERROR,
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
std_nonfirst (d)
=   let arg_err be arg_check (d, list (type (car (d))))
    in
```

```
    if arg_err = nil
    then if sequence_descp (type (car (d)))
         then if 0 = vsize (value (car (d)))
              then marked (empty_seq_error ('nonfirst, car (d)),
                           default_value (type (car (d))))
              else subsequence_get (car (d),
                                    GITWO,
                                    marked (nil,
                                            typed (INTEGER_DESC,
                                                   vsize (value (car (d))))))) endif
         else marked (NONFIRST_ARG_ERROR,
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
std_nonlast (d)
=   let arg_err be arg_check (d, list (type (car (d))))
    in
    if arg_err = nil
    then if sequence_descp (type (car (d)))
         then if 0 = vsize (value (car (d)))
              then marked (empty_seq_error ('nonlast, car (d)),
                           default_value (type (car (d))))
              else subsequence_get (car (d),
                                    GIONE,
                                    marked (nil,
                                            typed (INTEGER_DESC,
                                                   vsize (value (car (d))) − 1))) endif
         else marked (NONLAST_ARG_ERROR,
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
gnull_map (td)
=   if mapping_descp (td)
    then let r be dtype (td, NULL_MAP)
         in
         if truep (r) then marked (nil, typed (td, NULL_MAP))
         elseif errorp (r) then marked (r, default_value (td))
         else marked (not_in_type_error (NULL_MAP, td),
                      default_value (td)) endif endlet
    else marked (not_mapping_type_error (td),
                 default_value (INTEGER_DESC)) endif
```

DEFINITION:

```
gnull_seq (td)
=   if sequence_descp (td)
    then let r be dtype (td, NULL_SEQ)
         in
         if truep (r) then marked (nil, typed (td, NULL_SEQ))
         elseif errorp (r) then marked (r, default_value (td))
         else marked (not_in_type_error (NULL_SEQ, td),
                      default_value (td)) endif endlet
    else marked (not_sequence_type_error (td),
                 default_value (INTEGER_DESC)) endif
```

DEFINITION:

```
gnull_set (td)
=   if set_descp (td)
    then let r be dtype (td, NULL_SET)
         in
         if truep (r) then marked (nil, typed (td, NULL_SET))
         elseif errorp (r) then marked (r, default_value (td))
         else marked (not_in_type_error (NULL_SET, td),
                      default_value (td)) endif endlet
    else marked (not_set_type_error (td), default_value (INTEGER_DESC)) endif
```

DEFINITION:

```
std_null (d)
=   let arg_err be arg_check (d, list ('type_descriptor))
    in
    if arg_err = nil
    then let td be unmark (car (d))
         in
         case on root (td):
         case = mapping
         then gnull_map (td)
         case = sequence
          then gnull_seq (td)
         case = set
          then gnull_set (td)
         otherwise marked (null_undefined_error (td),
                           default_value (INTEGER_DESC)) endcase endlet
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
std_ord (d)
=   let td be type (car (d))
    in
    let arg_err be arg_check (d, list (td))
```

```
    in
    if arg_err = nil
    then if scalar_typep (td)
         then marked (nil, typed (INTEGER_DESC, value (car (d))))
         else marked (ORD_ARG_ERROR,
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
std_pred (d)
=   let td  be  base_type (type (car (d)))
    in
    let arg_err  be  arg_check (d, list (td))
    in
    if arg_err = nil
    then if scalar_typep (td)
         then if gtruep (gequal (car (d),
                                 std_lower (list (marked ('type_descriptor,
                                                          td)))))
              then marked (lower_pred_error (td),
                           default_value (td))
              else marked (nil,
                           typed (td, value (car (d)) - 1)) endif
         else marked (PRED_ARG_ERROR,
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet endlet
```

DEFINITION:

```
std_range (d)
=   let arg_err  be  arg_check (d, list (type (car (d))))
    in
    if arg_err = nil
    then if mapping_descp (type (car (d)))
         then marked (nil,
                      typed (set_desc (nil, component_td (type (car (d)))),
                             vrange (value (car (d)))))
         else marked (RANGE_ARG_ERROR,
                      default_value (INTEGER_DESC)) endif
    else marked (arg_err, default_value (INTEGER_DESC)) endif endlet
```

DEFINITION:

```
std_scale (d)
=   let arg_err  be  arg_check (d,
                                list (INTEGER_DESC, 'type_descriptor))
    in
```

```
    if arg_err = nil
    then let td be base_type(unmark(cadr(d)))
         in
         if scalar_typep(td)
         then let ok be dtype(td, value(car(d)))
              in
              if truep(ok)
              then marked(nil,
                          typed(td, value(car(d))))
              elseif errorp(ok)
              then marked(ok, default_value(td))
              else marked(scale_int_arg_error(value(car(d)),
                                              td),
                          default_value(td)) endif endlet
         else marked(scale_type_arg_error(td),
                     default_value(INTEGER_DESC)) endif endlet
    else marked(arg_err, default_value(INTEGER_DESC)) endif endlet
```

DEFINITION:

```
std_size(d)
=   let arg_err be arg_check(d, list(type(car(d))))
    in
    if arg_err = nil
    then if mapping_descp(type(car(d)))
            ∨  sequence_descp(type(car(d)))
            ∨  set_descp(type(car(d)))
         then marked(nil, typed(INTEGER_DESC, vsize(value(car(d)))))
         else marked(SIZE_ARG_ERROR,
                     default_value(INTEGER_DESC)) endif
    else marked(arg_err, default_value(INTEGER_DESC)) endif endlet
```

DEFINITION:

```
std_upper(d)
=   let arg_err be arg_check(d, list('type_descriptor))
    in
    if arg_err = nil
    then let td be unmark(car(d))
         in
         let r be tmax(td)
         in
         if r = nil
         then marked(unbounded_type_error(td),
                     default_value(td))
         elseif errorp(r)
```

then marked (*r*, default_value (*td*))
else marked (**nil**, typed (*td*, *r*)) **endif endlet endlet**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet**

DEFINITION:
std_succ (*d*)
= **let** *td* **be** base_type (type (car (*d*)))
in
let *arg_err* **be** arg_check (*d*, list (*td*))
in
if *arg_err* = **nil**
then if scalar_typep (*td*)
then if gtruep (gequal (car (*d*),
std_upper (list (marked (`'type_descriptor`,
td)))))
then marked (upper_succ_error (*td*),
default_value (*td*))
else marked (**nil**,
typed (*td*, 1 + value (car (*d*)))) **endif**
else marked (SUCC_ARG_ERROR,
default_value (INTEGER_DESC)) **endif**
else marked (*arg_err*, default_value (INTEGER_DESC)) **endif endlet endlet**

```
; ***********
;  Variables
; ***********
```

DEFINITION:
new_namep (*n*, *v*) = ((¬ in_map (*v*, *n*)) ∧ (¬ reserved_idp (*n*)))

DEFINITION: free_variablep (*n*, *v*) = in_map (*v*, *n*)

DEFINITION: free_value (*n*, *v*) = mapped_value (*v*, *n*)

DEFINITION:
apply_var (*fn*, *v*, *d*)
= **if** free_variablep (*fn*, *v*) **then** select_op (free_value (*fn*, *v*), *d*)
else marked (unknown_name_error (*fn*), default_value (INTEGER_DESC)) **endif**

```
; ***************
;  Function Calls
; ***************
```

```
DEFINITION:
gf_prec (u)
=   subst_tree (mk_entry_value ('result),
                mk_identifier ('result),
                prec (u))

DEFINITION:
eq_opp (op, etype)
=   if rule (op, prodn (tag ('binary_operator, 'op), 'eq))
      ∨   rule (op, prodn (tag ('binary_operator, 'op), 'equal))
    then equality_typep (etype)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'iff))
    then boolean_typep (etype)
    else f endif

DEFINITION:
f_of_formals (fn, fs)
=   if fs ≃ nil
    then mk_expression (mk_modified_primary_value (mk_identifier (fn)))
    else mk_expression (mk_tree ('modified_primary_value,
                                 list (mk_modified_primary_value (mk_identifier (fn)),
                                       mk_value_modifiers (namelist_to_actuals (dparam_name_list (fs),
                                                                                nil))))) endif

DEFINITION:
function_defn (u, ftype)
=   let fc be f_of_formals (unit_name (u), formal_dargs (u)),
        result be mk_expression ('result)
    in
    let e be postc (u, 'normal)
    in
    if rule (e,
             prodn (tag ('expression, 'e),
                    list (tag ('expression, 'e1),
                          tag ('binary_operator, 'op),
                          tag ('expression, 'e2))))
    then if eq_opp (subtree (e, 'binary_operator), ftype)
         then if (subtree_i (e, 'expression, 1) = result)
                 ∨   (subtree_i (e, 'expression, 1)
                      =   fc)
              then subst_tree (fc,
                               result,
                               subtree_i (e, 'expression, 2))
              elseif (subtree_i (e, 'expression, 2)
                      =   result)
```

```
                    ∨   (subtree_i (e, 'expression, 2)
                            =   fc)
                then subst_tree (fc,
                                 result,
                                 subtree_i (e, 'expression, 1))
                else nil endif
            else nil endif
        else nil endif endlet endlet

; Previous definition of function_defn
;(defn function_defn (u ftype)
;  (let ((fc (f_of_formals (unit_name u) (formal_dargs u)))
; (result (mk_expression 'result)))
;  (let ((e (subst_tree result fc (postc u 'NORMAL))))
;
;    (if (rule e (prodn (tag 'expression 'e)
;        (list (tag 'expression 'e1)
;      (tag 'binary_operator 'op)
;      (tag 'expression 'e2))))
; (if (eq_opp (subtree e 'binary_operator) ftype)
;     (if (and (equal (subtree_i e 'expression 1) result)
;      (not (subtreep result (subtree_i e 'expression 2))))
; (subtree_i e 'expression 2)
;     (if (and (equal (subtree_i e 'expression 2) result)
;      (not (subtreep result (subtree_i e 'expression 1))))
; (subtree_i e 'expression 1)
;       nil))
;   nil)
;      nil))))
```

DEFINITION:
has_defn (u, *ftype*)
= **if** kind (u) = `'function` **then** function_defn (u, *ftype*) $\neq$ **nil**
elseif kind (u) = `'constant` **then** constant_value_exp (u) $\neq$ **nil**
else f endif

DEFINITION:
gdefn (u, *ftype*)
= **if** kind (u) = `'function` **then** function_defn (u, *ftype*)
elseif kind (u) = `'constant` **then** constant_value_exp (u)
else nil endif

DEFINITION:

formal_type_list (*fs*, *sn*, *x*)
= **if** *fs* ≃ **nil then nil**
else cons (type_desc (formal_type (car (*fs*)), *sn*, **nil**, *x*),
formal_type_list (cdr (*fs*), *sn*, *x*)) **endif**

EVENT: Disable reserved_idp.

EVENT: Disable dparam_name.

EVENT: Disable param_reserved_error.

EVENT: Disable dparam_name_list.

EVENT: Disable duplicate_param_names_error.

EVENT: Disable access.

EVENT: Disable function_access_error.

DEFINITION:
fformals_check (*fs*)
= **if** *fs* ≃ **nil then nil**
elseif reserved_idp (dparam_name (car (*fs*)))
then param_reserved_error (car (*fs*))
elseif (dparam_name (car (*fs*)) ∈ dparam_name_list (cdr (*fs*)))
∨ (dparam_name (car (*fs*)) = **'result**)
then duplicate_param_names_error (car (*fs*))
elseif access (car (*fs*)) = **'var then** function_access_error (car (*fs*))
else fformals_check (cdr (*fs*)) **endif**

DEFINITION:
farg_check (*fs*, *as*, *fsn*, *x*)
= **let** *r* **be** fformals_check (*fs*)
in
if *r* = **nil then** arg_check (*as*, formal_type_list (*fs*, *fsn*, *x*))
else *r* **endif endlet**

DEFINITION:
bind_args (*fs*, *as*, *fsn*, *x*)
= **if** *fs* ≃ **nil then** EMPTY_MAP
else add_to_map (bind_args (rcdr (*fs*), rcdr (*as*), *fsn*, *x*),

dparam_name (rcar (*fs*)),
marked (**nil**,
typed (type_desc (formal_type (rcar (*fs*)),
fsn,
nil,
x),
value (rcar (*as*))))) **endif**

DEFINITION:
mk_entry_name (*n*) = pack (rcons (unpack (*n*), ASCII_SINGLE_QUOTE))

DEFINITION:
type_namep (*tn*, *sn*, *x*)
= ((*tn* ∈ '(boolean character integer rational))
∨ (kind (ref_unit (ref (*tn*, *sn*, *x*))) = 'type))

DEFINITION:
type_name_expp (*fn*, *ap*, *sn*, *x*) = (type_namep (*fn*, *sn*, *x*) ∧ (*ap* ≃ **nil**))

```
; ***********************
;  Expression Evaluation
; ***********************

; Note:  Semantic errors that don't come up during evaluation of expressions
;        are not detected.
;
;        1. Uniqueness of local names requirements are enforced only by the
;           reference resolution mechanism and by requiring unique names for
;           data objects in the environment.  For example, in
;
;               f(x) & all f : t, g(x,f)
;
;           the use of f both as a function name and as a bound identifier is
;           not detected, and in
;
;               function f (i : i) : boolean = pending
;
;           the use of i both as a parameter name and as a type name is not
;           detected.
;
;        2. Errors in conditional exit specifications of functions are
;           detected only if they make it impossible to locate the function
;           definition.  None of the errors in
;
;               function f (x : integer) : boolean =
```

```
;               begin
;                 exit case (is normal : (assume result = g(x));
;                   is     c1 : (assume c1(x));
;              is     c1 : (assume c2(x)));
;               end;
;
;          are detected.
;
;       3. Errors are detected in only the evaluated parts of if expressions.
;          For example, no errors are detected in
;
;               if false then
;                 'a'
;               elif true then
;                 3
;               elif 1/2 then
;                 [set: binary 28]
;               else 4 + "a string"
;               fi
;
;       4. Errors in scope, unit, and name declarations are detected only if
;          they come up in a unit reference.
```

DEFINITION:
bound_values (e, c, x)
= **if** rule (e,
prodn (tag ('expression, 'e),
list ('all, tag ('bound_expression, 'b))))
then bound_values (subtree (e, 'bound_expression), c, x)
elseif rule (e,
prodn (tag ('expression, 'e),
list ('some, tag ('bound_expression, 'b))))
then bound_values (subtree (e, 'bound_expression), c, x)
elseif rule (e,
prodn (tag ('bound_expression, 'b),
list (tag ('identifier_list, 'q),
'colon,
tag ('type_specification, 's),
'comma,
tag ('expression, 'e))))
then marked_typed_value_set (type_desc (subtree (e,

```
                                        'type_specification),
                                    c,
                                    nil,
                                    x))
    elseif rule(e,
                prodn(tag('opt_each_clause, 'e),
                      list('each,
                           tag('identifier, 'i),
                           'colon,
                           tag('type_specification, 'ts),
                           'comma)))
    then let td be type_desc(subtree(e, 'type_specification),
                             c,
                             nil,
                             x)
         in
         if bounded_index_typep(td)
         then marked_typed_value_set(td)
         else each_id_type_error(e, c) endif endlet
    else nil endif

; ================================
;  Lemmas for GF Group Termination
; ================================

; Make these lemmas more reasonable if there is time.


; ============================================
;  Case of Characters Does Not Affect Tree Size
; ============================================
```

THEOREM: upper_case_tree_size
ascii_characterp(x) → (tree_size(upper_case(x)) = tree_size(x))

THEOREM: uc_list_tree_size
ascii_character_listp(x) → (tree_size(uc_list(x)) = tree_size(x))

EVENT: Disable upper_case_tree_size.

EVENT: Disable uc_list_tree_size.

```
; ========================================================
```

```
;  Else Part of If Expression Is Smaller than If Expression
; ========================================================

; -------------------------
;  IF is smaller than ELIF
; -------------------------
```

THEOREM: lessp_subtrees_imp_lessp_tree_size
(treep (*x*)
 ∧ treep (*y*)
 ∧ (tree_size (subtrees (*x*)) < tree_size (subtrees (*y*))))
→ (tree_size (*x*) < tree_size (*y*))

THEOREM: same_leaf_imp_same_tree_size1
(leafp (*x*)
 ∧ leafp (*y*)
 ∧ (root (*x*) = root (*y*))
 ∧ (lexeme (*x*) = lexeme (*y*)))
→ (tree_size (*x*) = tree_size (*y*))

THEOREM: same_leaf_imp_same_tree_size2
(leafp (*x*)
 ∧ leafp (*y*)
 ∧ (root (*x*) = root (*y*))
 ∧ (uc_list (lexeme (*x*)) = uc_list (lexeme (*y*))))
→ (tree_size (*x*) = tree_size (*y*))

THEOREM: leaf_equal_imp_tree_size_equal
leaf_equal (*x*, *y*) → (tree_size (*x*) = tree_size (*y*))

THEOREM: tree_equal_imp_tree_size_equal
tree_equal (*x*, *y*) → (tree_size (*x*) = tree_size (*y*))

THEOREM: elif_leafp
(parse_tree_leafp (*e*) ∧ (root (*e*) = 'elif))
→ tree_equal (*e*, mk_reserved_word ('elif))

THEOREM: mk_rhs_imp_root
(parse_treep (*e*) ∧ (mk_rhs (*e*) = 'elif))
→ (parse_tree_leafp (*e*) ∧ (root (*e*) = 'elif))

THEOREM: if_exp_else_subtrees_car
rule (*e*,
      prodn (tag ('if_expression_else_part, 'e),

```
            list('elif,
                 tag('expression, 'b),
                 'then,
                 tag('expression, 'p),
                 tag('if_expression_else_part, 'e2))))
→   tree_equal(car(subtrees(e)), mk_reserved_word('elif))

THEOREM: lessp_if_than_elif
rule(e,
     prodn(tag('if_expression_else_part, 'e),
            list('elif,
                 tag('expression, 'b),
                 'then,
                 tag('expression, 'p),
                 tag('if_expression_else_part, 'e2))))
→   (tree_size(mk_reserved_word('if)) < tree_size(car(subtrees(e))))

; ---------------------
;  Subtrees Are a List
; ---------------------


THEOREM: listp_if_exp_else_subtrees
rule(e,
     prodn(tag('if_expression_else_part, 'e),
            list('elif,
                 tag('expression, 'b),
                 'then,
                 tag('expression, 'p),
                 tag('if_expression_else_part, 'e2))))
→   listp(subtrees(e))

; --------------------------
;  The Else Part Is Smaller
; --------------------------


THEOREM: lessp_car_imp_lessp_tree_size
(listp(x)
 ∧  listp(y)
 ∧  (tree_size(car(x)) < tree_size(car(y)))
 ∧  (tree_size(cdr(x)) = tree_size(cdr(y))))
→   (tree_size(x) < tree_size(y))

THEOREM: lessp_mk_tree_car_imp_lessp_tree_size
```

(treep (e)
 ∧ listp (subtrees (e))
 ∧ (tree_size (k) < tree_size (car (subtrees (e)))))
→ (tree_size (mk_tree (nt, cons (k, cdr (subtrees (e))))) < tree_size (e))

EVENT: Disable lessp_car_imp_lessp_tree_size.

THEOREM: leq_if_exp_else_part
(tree_size (if_else_exp (e)) ≮ tree_size (e))
→ (tree_size (if_else_exp (e)) = tree_size (e))

THEOREM: lessp_if_exp_else_part
rule (e, prodn (x, y))
→ (tree_size (if_else_exp (subtree (e, n))) < tree_size (e))

```
; ======================================
;  Gname Is Smaller Than Identifier Tree
; ======================================
```

THEOREM: identifierp_imp_treep
identifierp (e) → treep (e)

THEOREM: lessp_gname_tree_size_0
treep (e) → (tree_size (gname (e)) < tree_size (e))

THEOREM: lessp_gname_tree_size
identifierp (e) → (tree_size (gname (e)) < tree_size (e))

```
; ================================================
;  Object Name Is Smaller Than Containing Expression
; ================================================
```

THEOREM: leq_object_name_tree_size
(tree_size (object_name (e)) ≮ tree_size (e))
→ (tree_size (object_name (e)) = tree_size (e))

EVENT: Disable leq_object_name_tree_size.

THEOREM: lessp_object_name_tree_size
rule (e, prodn (x, y))
→ (tree_size (object_name (subtree (e, n))) < tree_size (e))

```
; ================================================
;  Argument List Smaller Than Containing Expression
; ================================================
```

THEOREM: leq_arg_list_tree_size
(tree_size (arg_list (*e*)) $\not<$ tree_size (*e*))
$\rightarrow$ (tree_size (arg_list (*e*)) = tree_size (*e*))

EVENT: Disable leq_arg_list_tree_size.

THEOREM: lessp_arg_list_tree_size
rule (*e*, prodn (*x*, *y*)) $\rightarrow$ (tree_size (arg_list (subtree (*e*, *n*))) < tree_size (*e*))

```
; ========================================================
;  Bound Identifier Is Smaller Than Containing Expression
; ========================================================
```

THEOREM: leq_bound_id_tree_size
(tree_size (bound_id (*e*)) $\not<$ tree_size (*e*))
$\rightarrow$ (tree_size (bound_id (*e*)) = tree_size (*e*))

EVENT: Disable leq_bound_id_tree_size.

THEOREM: lessp_bound_id_tree_size
rule (*e*, prodn (*x*, *y*)) $\rightarrow$ (tree_size (bound_id (subtree (*e*, *n*))) < tree_size (*e*))

EVENT: Disable *1*gfalse.

EVENT: Disable *1*gizero.

EVENT: Disable *1*gtrue.

EVENT: Disable *1*boolean_desc.

EVENT: Disable *1*integer_desc.

EVENT: Disable *1*mk_unary_operator.

EVENT: Disable gf_prec.

EVENT: Disable gand.

EVENT: Disable gchar.

EVENT: Disable gdefn.

EVENT: Disable gfalse.

EVENT: Disable gizero.

EVENT: Disable gle.

EVENT: Disable gmap_insert.

EVENT: Disable gmapomit.

EVENT: Disable gor.

EVENT: Disable grange_elements.

EVENT: Disable gseq_insert_before.

EVENT: Disable gseq_insert_behind.

EVENT: Disable gseqomit.

EVENT: Disable gset_or_seq.

EVENT: Disable gstring_seq.

EVENT: Disable gtrue.

EVENT: Disable gtruep.

EVENT: Disable arg_list.

EVENT: Disable add_to_map.

EVENT: Disable apply_binary_op.

EVENT: Disable apply_unary_op.

EVENT: Disable apply_var.

EVENT: Disable bad_value_modifiers_error.

EVENT: Disable base_type.

EVENT: Disable bind_args.

EVENT: Disable boolean_desc.

EVENT: Disable bound_id.

EVENT: Disable bound_values.

EVENT: Disable cdr_quantified_exp.

EVENT: Disable character_valuep.

EVENT: Disable condition_params_error.

EVENT: Disable default_value.

EVENT: Disable determinate.

EVENT: Disable digit_listp.

EVENT: Disable each_clausep.

EVENT: Disable entry_name.

EVENT: Disable entry_not_true_error.

EVENT: Disable entry_valuep.

EVENT: Disable errorp.

EVENT: Disable farg_check.

EVENT: Disable fn_call_formp.

EVENT: Disable formal_dargs.

EVENT: Disable free_variablep.

EVENT: Disable gname.

EVENT: Disable has_defn.

EVENT: Disable identifierp.

EVENT: Disable if_else_exp.

EVENT: Disable in_type.

EVENT: Disable indeterminate.

EVENT: Disable indeterminate_fn_result_error.

EVENT: Disable integer_desc.

EVENT: Disable kind.

EVENT: Disable length.

EVENT: Disable minteger.

EVENT: Disable mk_entry_name.

EVENT: Disable mk_identifier.

EVENT: Disable mk_unary_operator.

EVENT: Disable n_too_small.

EVENT: Disable name_already_in_use_error.

EVENT: Disable new_namep.

EVENT: Disable no_function_defn_error.

EVENT: Disable not_expression_error.

EVENT: Disable not_function_or_const_error.

EVENT: Disable object_name.

EVENT: Disable put_op.

EVENT: Disable rcar.

EVENT: Disable rcdr.

EVENT: Disable rcons.

EVENT: Disable record_get.

EVENT: Disable ref.

EVENT: Disable result_type.

EVENT: Disable select_op.

EVENT: Disable std_domain.

EVENT: Disable std_first.

EVENT: Disable std_initial.

EVENT: Disable std_last.

EVENT: Disable std_lower.

EVENT: Disable std_max.

EVENT: Disable std_min.

EVENT: Disable std_nonfirst.

EVENT: Disable std_nonlast.

EVENT: Disable std_null.

EVENT: Disable std_ord.

EVENT: Disable std_pred.

EVENT: Disable std_range.

EVENT: Disable std_scale.

EVENT: Disable std_size.

EVENT: Disable std_succ.

EVENT: Disable std_upper.

EVENT: Disable string_valuep.

EVENT: Disable subsequence_get.

EVENT: Disable tree_size.

EVENT: Disable type.

EVENT: Disable type_desc.

EVENT: Disable type_name_expp.

EVENT: Disable typed.

EVENT: Disable unmark.

EVENT: Disable value.

```
#|
(do-mutual '(


; ***************************
;  Set/Sequence Constructors
; ***************************

(defn GF_element_list (e c v n x)

  (if (rule e (prodn (tag 'range 'r)
     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GF_element_list (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'value_list 'v)))
      (GF_element_list (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'range_limits 'r)))
      (GF_element_list (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'range_limits 'r)
     (list (tag 'expression 'lo) 'DOT_DOT
```

```
   (tag 'expression 'hi))))
      (Grange_elements (GF (subtree_i e 'expression 1) c v n x)
       (GF (subtree_i e 'expression 2) c v n x))

  (if (rule e (prodn (tag 'value_list 'v)
     (tag 'expression 'e)))
      (rcons nil (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'value_list 'v)
     (list (tag 'value_list 'v2) 'COMMA
   (tag 'expression 'e))))
      (rcons (GF_element_list (subtree e 'value_list) c v n x)
     (GF (subtree e 'expression) c v n x))

    nil))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler

(defn GF_element_type (e c v n x)

  (if (rule e (prodn (tag 'range 'r)
     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GF_element_type (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'value_list 'v)))
      (GF_element_type (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'range_limits 'r)))
      (GF_element_type (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'range_limits 'r)
     (list (tag 'expression 'lo) 'DOT_DOT
   (tag 'expression 'hi))))
      (base_type (type (GF (subtree_i e 'expression 1) c v n x)))

  (if (rule e (prodn (tag 'value_list 'v)
     (tag 'expression 'e)))
      (base_type (type (GF (subtree e 'expression) c v n x)))

  (if (rule e (prodn (tag 'value_list 'v)
     (list (tag 'value_list 'v2) 'COMMA
```

```
   (tag 'expression 'e))))
      (GF_element_type (subtree e 'value_list) c v n x)

    nil))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler



; ************************
;  Quantified expressions
; ************************

(defn GF_all (id vs e c v n x)
  (if (nlistp vs)
      (Gtrue)
  (if (new_namep id v)
      (if (zerop n)
  (marked (n_too_small)
  (default_value (boolean_desc)))
        (Gand (GF_all id (rcdr vs) e c v n x)
      (GF e c (add_to_map v id (rcar vs)) (sub1 n) x)))
    (marked (name_already_in_use_error id)
    (default_value (boolean_desc)))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size id)))
     (count vs))) ))

(defn GF_some (id vs e c v n x)
  (if (nlistp vs)
      (Gfalse)
    (if (new_namep id v)
      (if (zerop n)
  (marked (n_too_small)
  (default_value (boolean_desc)))
(Gor (GF_some id (rcdr vs) e c v n x)
     (GF e c (add_to_map v id (rcar vs)) (sub1 n) x)))
      (marked (name_already_in_use_error id)
      (default_value (boolean_desc)))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size id)))
     (count vs))) ))
```

```
; *********************
;  Value Modifications
; *********************

(defn GF_each (id vs bv e c v n x)
  ; e is the <component modification>
  (if (nlistp vs)
      bv
  (if (new_namep id v)
      (if (zerop n)
  (marked (n_too_small)
  (default_value (base_type (type bv))))
(GF_each id (cdr vs)
 (GF_modifiers bv e c (add_to_map v id (car vs)) (sub1 n) x)
 e c v n x))
    (marked (name_already_in_use_error id)
    (default_value (base_type (type bv))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count vs))) ))

(defn GF_adp (e c v n x)

  (if (rule e (prodn (tag 'arg_list 'as)
     (list 'OPEN_PAREN (tag 'value_list 'vs)
   'CLOSE_PAREN)))
      (GF_adp (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'value_list 'vs)
     (tag 'expression 'e)))
      (rcons nil (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'value_list 'vs)
     (list (tag 'value_list 'vs2)
   'COMMA (tag 'expression 'e))))
      (rcons (GF_adp (subtree e 'value_list) c v n x)
     (GF (subtree e 'expression) c v n x))
    nil)))

  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler
```

```
(defn GF_selectors (e c v n x)

  (if (rule e (prodn (tag 'selector_list 's)
     (tag 'component_selectors 's2)))
      (GF_selectors (subtree e 'component_selectors) c v n x)

  (if (rule e (prodn (tag 'selector_list 's)
     (list (tag 'selector_list 's2)
   (tag 'component_selectors 's3))))
      (append (GF_selectors (subtree e 'selector_list) c v n x)
      (GF_selectors (subtree e 'component_selectors) c v n x))

  (if (rule e (prodn (tag 'component_selectors 's)
     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (list (marked 'field_name
    (gname (subtree e 'IDENTIFIER))))

  (if (rule e (prodn (tag 'component_selectors 's)
     (tag 'arg_list 'd)))
      (GF_adp (subtree e 'arg_list) c v n x)

  (if (rule e (prodn (tag 'arg_list 'as)
     (list 'OPEN_PAREN (tag 'value_list 'vs)
   'CLOSE_PAREN)))
      (GF_adp (subtree e 'value_list) c v n x)

    nil)))))

  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler

(defn GF_modifiers (bv e c v n x)
  ; e is the <value modifiers>

  (if (rule e (prodn (tag 'value_modifiers 'm)
     (tag 'component_selectors 's)))
      (GF_modifiers bv (subtree e 'component_selectors) c v n x)

  (if (rule e (prodn (tag 'component_selectors 's)
     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (record_get bv
  (marked 'field_name
  (gname (subtree e 'IDENTIFIER)))
```

```
(type bv))

(if (rule e (prodn (tag 'component_selectors 's)
   (tag 'arg_list 'd)))
    (select_op bv (GF_adp (subtree e 'arg_list) c v n x))

(if (rule e (prodn (tag 'value_modifiers 'm)
   (tag 'range 'r)))
    (GF_modifiers bv (subtree e 'range) c v n x)

(if (rule e (prodn (tag 'range 'r)
   (list 'OPEN_PAREN (tag 'range_limits 'r2)
 'CLOSE_PAREN)))
    (GF_modifiers bv (subtree e 'range_limits) c v n x)

(if (rule e (prodn (tag 'range_limits 'r)
   (list (tag 'expression 'lo) 'DOT_DOT
 (tag 'expression 'hi))))
    (subsequence_get bv
     (GF (subtree_i e 'expression 1) c v n x)
     (GF (subtree_i e 'expression 2) c v n x))

(if (rule e (prodn (tag 'value_modifiers 'm)
   (tag 'value_alterations 'a)))
    (GF_modifiers bv (subtree e 'value_alterations) c v n x)

(if (rule e (prodn (tag 'value_alterations 'a)
   (list 'WITH 'OPEN_PAREN
 (tag 'component_alterations_list 'al)
 'CLOSE_PAREN)))
    (GF_modifiers bv (subtree e 'component_alterations_list) c v n x)

(if (rule e (prodn (tag 'component_alterations_list 'al)
   (tag 'component_alterations 'a)))
    (GF_modifiers bv (subtree e 'component_alterations) c v n x)

(if (rule e (prodn (tag 'component_alterations_list 'al)
   (list (tag 'component_alterations_list 'al2)
 'SEMI_COLON (tag 'component_alterations 'a))))
    (GF_modifiers (GF_modifiers bv (subtree e 'component_alterations_list)
c v n x)
  (subtree e 'component_alterations) c v n x)

(if (rule e (prodn (tag 'component_alterations 'as)
```

```
     (list (tag 'opt_each_clause 'e)
   (tag 'component_assignment 'a))))
      (if (each_clausep (subtree e 'opt_each_clause))
  (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
    (if (errorp vs)
(marked vs (default_value (base_type (type bv))))
      (GF_each (bound_id (subtree e 'opt_each_clause))
       vs bv (subtree e 'component_assignment) c v n x)))
(GF_modifiers bv (subtree e 'component_assignment) c v n x))

  (if (rule e (prodn (tag 'component_alterations 'as)
     (list (tag 'opt_each_clause 'e)
   (tag 'component_creation 'c))))
      (if (each_clausep (subtree e 'opt_each_clause))
  (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
    (if (errorp vs)
(marked vs (default_value (base_type (type bv))))
      (GF_each (bound_id (subtree e 'opt_each_clause))
       vs bv (subtree e 'component_creation) c v n x)))
(GF_modifiers bv (subtree e 'component_creation) c v n x))

  (if (rule e (prodn (tag 'component_alterations 'as)
     (list (tag 'opt_each_clause 'e)
   (tag 'component_deletion 'd))))
      (if (each_clausep (subtree e 'opt_each_clause))
  (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
    (if (errorp vs)
(marked vs (default_value (base_type (type bv))))
      (GF_each (bound_id (subtree e 'opt_each_clause))
       vs bv (subtree e 'component_deletion) c v n x)))
(GF_modifiers bv (subtree e 'component_deletion) c v n x))

  (if (rule e (prodn (tag 'component_assignment 'a)
     (list (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (put_op bv
      (GF_selectors (subtree e 'selector_list) c v n x)
      (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'BEFORE (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
    (u (GF (subtree e 'expression) c v n x)))
```

```
(put_op bv (rcdr s)
(Gseq_insert_before (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'BEHIND (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
    (u (GF (subtree e 'expression) c v n x)))
(put_op bv (rcdr s)
(Gseq_insert_behind (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'INTO (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
    (u (GF (subtree e 'expression) c v n x)))
(put_op bv (rcdr s)
(Gmap_insert (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_deletion 'd)
     (list 'SEQOMIT (tag 'selector_list 's))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x)))
(put_op bv (rcdr s)
(Gseqomit (select_op bv (rcdr s)) (rcar s))))

  (if (rule e (prodn (tag 'component_deletion 'd)
     (list 'MAPOMIT (tag 'selector_list 's))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x)))
(put_op bv (rcdr s)
(Gmapomit (select_op bv (rcdr s)) (rcar s))))

    (marked (bad_value_modifiers_error e)
    (default_value (base_type (type bv)))))))))))))))))))))))))

  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler



; ***********************************
;  Name references and function calls
; ***********************************
```

```
(defn apply_fun (fn d sn n x)
  (let ((h (car (ref fn sn x)))  ; scope fn is declared in
(u (cdr (ref fn sn x)))) ; the function declaration
    (if (or (equal (kind u) 'function)
    (equal (kind u) 'constant))
(let ((ftype (type_desc (result_type u) h nil x))
      (fs (formal_dargs u)))                  ; formals
(let ((a (if (equal (length fs) 0) nil d))    ; actuals
      (s (if (equal (length fs) 0) d nil)))   ; selectors
  (if (zerop (fix n))
      (marked (n_too_small)
      (default_value ftype))
    (select_op
     (let ((arg_err (farg_check fs a h x)))
       (if (equal arg_err nil)
   (let ((v (add_to_map (bind_args fs a h x)
(mk_entry_name 'result)
(std_initial
 (list (marked 'type_descriptor
       ftype))))))
     (if (Gtruep (GF (GF_prec u) h v (sub1 n) x))
 (if (has_defn u ftype)
     (let ((r (GF (Gdefn u ftype)
  h v (sub1 n) x)))
       (if (and (determinate r)
(truep (in_type ftype r)))
   (marked nil
   (typed ftype (value r)))
 (marked
   (indeterminate_fn_result_error fn sn)
   (default_value ftype))))
   (marked (no_function_defn_error fn sn)
   (default_value ftype)))
       (marked (entry_not_true_error fn sn)
       (default_value ftype))))
 (marked arg_err (default_value ftype))))
     s))))
    (if (equal (kind u) 'error)
(marked u (default_value (integer_desc)))
      (marked (not_function_or_const_error fn sn)
      (default_value (integer_desc))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size fn)))
     (tree_size fn))) ))
```

```
(defn Gapply (fn ap sn v n x)
  (if (free_variablep fn v)
      (apply_var fn v ap)
  (if (equal fn 'false)
      (select_op (Gfalse) ap)
  (if (equal fn 'true)
      (select_op (Gtrue) ap)
  (if (type_name_expp fn ap sn x)
      (marked 'type_descriptor
      (type_desc (mk_identifier fn) sn nil x))
  (if (equal fn 'domain)
      (std_domain ap)
  (if (equal fn 'first)
      (std_first ap)
  (if (equal fn 'initial)
      (std_initial ap)
  (if (equal fn 'last)
      (std_last ap)
  (if (equal fn 'lower)
      (std_lower ap)
  (if (equal fn 'max)
      (std_max ap)
  (if (equal fn 'min)
      (std_min ap)
  (if (equal fn 'nonfirst)
      (std_nonfirst ap)
  (if (equal fn 'nonlast)
      (std_nonlast ap)
  (if (equal fn 'null)
      (std_null ap)
  (if (equal fn 'ord)
      (std_ord ap)
  (if (equal fn 'pred)
      (std_pred ap)
  (if (equal fn 'range)
      (std_range ap)
  (if (equal fn 'scale)
      (std_scale ap)
  (if (equal fn 'size)
      (std_size ap)
  (if (equal fn 'succ)
      (std_succ ap)
  (if (equal fn 'upper)
```

```
      (std_upper ap)
    (apply_fun fn ap sn n x))))))))))))))))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size fn)))
     (add1 (tree_size fn)))) ))



(defn GF (e c v n x)
  ; The meta-function GF(e,c,v,n,x) gives the value that results when the
  ; expression e is evaluated, in the context of scope c with free variables
  ; bound as determined in the environment v, by at most n applications of
  ; functions described by the Gypsy sentence x.
  ;
  ; The domain and range of GF(e,c,v,n,x) are as follows:
  ;
  ;   e is the parse tree representing the expression to be evaluated.
  ;   c is the (litatom) name of the Gypsy scope in which e is evaluated.
  ;   v is the name-value mapping that maps names of free variables
  ;     into their (marked typed) values.
  ;   n is the maximum allowed depth of Gypsy function calls and quantifiers
  ;   x is the parse tree representing the Gypsy sentence that is being
  ;     interpreted.  This is the complete sentence containing the list of all
  ;     available scopes.
  ;   GF(e,c,v,n,x) is the marked, typed value that results from evaluating e.
  ;


  ; ************************
  ;
  ;    <expression> ::= ...
  ;
  ; ************************

  (if (rule e (prodn (tag 'expression 'e)
     (tag 'modified_primary_value 'm)))
      (GF (subtree e 'modified_primary_value) c v n x)

  (if (rule e (prodn (tag 'expression 'e)
     (list 'ALL (tag 'bound_expression 'b))))
      (let ((vs (bound_values e c x)))
(if (errorp vs)
    (marked vs (default_value (boolean_desc)))
  (GF_all (bound_id (subtree e 'bound_expression))
```

```
  vs (cdr_quantified_exp e) c v n x)))

  (if (rule e (prodn (tag 'expression 'e)
     (list 'SOME (tag 'bound_expression 'b))))
      (let ((vs (bound_values e c x)))
(if (errorp vs)
    (marked vs (default_value (boolean_desc)))
  (GF_some (bound_id (subtree e 'bound_expression))
   vs (cdr_quantified_exp e) c v n x)))

  (if (rule e (prodn (tag 'expression 'e)
     (list (tag 'unary_operator 'op) (tag 'expression 'e2))))
      (apply_unary_op (subtree e 'unary_operator)
      (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'expression 'e)
     (list (tag 'expression 'e1) (tag 'binary_operator 'op)
   (tag 'expression 'e2))))
      (apply_binary_op (subtree e 'binary_operator)
       (GF (subtree_i e 'expression 1) c v n x)
       (GF (subtree_i e 'expression 2) c v n x))


  ; ********************************
  ;
  ;    <modified primary value> ::=
  ;
  ; ********************************

  (if (rule e (prodn (tag 'modified_primary_value 'm)
     (tag 'primary_value 'p)))
      (GF (subtree e 'primary_value) c v n x)

  (if (rule e (prodn (tag 'modified_primary_value 'm)
     (list (tag 'modified_primary_value 'm2)
   (tag 'value_modifiers 'vm))))
      (if (fn_call_formp e)
  (Gapply (object_name (subtree e 'modified_primary_value))
  (GF_adp (arg_list (subtree e 'value_modifiers))
  c v n x)
  c v n x)
(GF_modifiers (GF (subtree e 'modified_primary_value) c v n x)
      (subtree e 'value_modifiers) c v n x))
```

```
  (if (rule e (prodn (tag 'modified_primary_value 'm)
     (list (tag 'modified_primary_value 'm2)
   (tag 'actual_condition_parameters 'cp))))
      ; Condition parameters allowed only in <specification expression>
      ; with validation directive (p. 35, Gypsy 2.05 Report).
      ; **** Weed them out before calling GF in that case. ****
      (let ((r (GF (subtree e 'modified_primary_value) c v n x)))
(marked (condition_params_error e)
(default_value (type r))))


  ; ************************
  ;
  ;    <primary value> ::=
  ;
  ; ************************

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'literal_value 'l)))
      (GF (subtree e 'literal_value) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'set_or_sequence_value 's)))
      (GF (subtree e 'set_or_sequence_value) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'ENTRY_VALUE 'e)))
      (GF (subtree e 'ENTRY_VALUE) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'IDENTIFIER 'on)))
      (GF (subtree e 'IDENTIFIER) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'if_expression 'i)))
      (GF (subtree e 'if_expression) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (list 'OPEN_PAREN (tag 'expression 'e) 'CLOSE_PAREN)))
      (GF (subtree e 'expression) c v n x)


  ; ---------------------------------------------------------------------
  ;
```

```
  ;  From here down to parse tree leaves, clauses are in alphabetical order
  ;  by the left-hand side of the productions.  Everything that is a parse
  ;  tree for an expression should be covered.
  ;
  ; ------------------------------------------------------------------------


  ; ***********************
  ;
  ;    <constant body> ::=
  ;
  ; ***********************

  ; I don't think the following needs to be here.  siebert 9/Sept/90
  (if (rule e (prodn (tag 'constant_body 'b)
     (tag 'expression 'e)))
      (GF (subtree e 'expression) c v n x)


  ; ***********************
  ;
  ;    <if expression> ::=
  ;
  ; ***********************

  (if (rule e (prodn (tag 'if_expression 'i)
     (list 'IF (tag 'expression 'b) 'THEN
   (tag 'expression 'p)
   (tag 'if_expression_else_part 'e))))

      ; Note:  this does not require all potential value expressions to be the
      ; same type or all boolean expressions to be boolean.
      (let ((bv (GF (subtree_i e 'expression 1) c v n x)))
(if (indeterminate bv)
    (marked (mark bv)
    (default_value
      (type (GF (subtree_i e 'expression 2)
c v n x))))
        (if (truep (in_type (boolean_desc) bv))
    (if (Gtruep bv)
(GF (subtree_i e 'expression 2) c v n x)
      (GF (if_else_exp (subtree e 'if_expression_else_part))
  c v n x))
  (marked (if_test_not_boolean_error e c)
```

```
(default_value
  (type (GF (subtree_i e 'expression 2)
    c v n x)))))))


; ***********************
;
;    <literal value> ::=
;
; ***********************

(if (rule e (prodn (tag 'literal_value 'l)
   (tag 'CHARACTER_VALUE 'ch)))
    (GF (subtree e 'CHARACTER_VALUE) c v n x)

(if (rule e (prodn (tag 'literal_value 'l)
   (tag 'number 'n)))
    (GF (subtree e 'number) c v n x)

(if (rule e (prodn (tag 'literal_value 'l)
   (tag 'STRING_VALUE 's)))
    (GF (subtree e 'STRING_VALUE) c v n x)


; ****************
;
;    <number> ::=
;
; ****************

(if (rule e (prodn (tag 'number 'n)
   (tag 'DIGIT_LIST 's)))
    (minteger e)

(if (rule e (prodn (tag 'number 'n)
   (list (tag 'base 'b) (tag 'DIGIT_LIST 's))))
    (minteger e)


; ***************************************
;
;    <pre-computable label expression> ::=
;
; ***************************************
```

```
(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (tag 'number 'n)))
    (GF (subtree e 'number) c v n x)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (list 'MINUS (tag 'number 'n))))
    (apply_unary_op (mk_unary_operator 'MINUS)
    (GF (subtree e 'number) c v n x))

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (tag 'CHARACTER_VALUE 'ch)))
    (GF (subtree e 'CHARACTER_VALUE) c v n x)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (tag 'IDENTIFIER 'i)))
    (GF (subtree e 'IDENTIFIER) c v n x)


; ********************************
;
;    <set or sequence value> ::=
;
; ********************************

(if (rule e (prodn (tag 'set_or_sequence_value 's)
   (list 'OPEN_PAREN (tag 'set_or_seq_mark 'm)
 (tag 'element_list 'e) 'CLOSE_PAREN)))
    (Gset_or_seq (subtree e 'set_or_seq_mark)
 (GF_element_list (subtree e 'element_list) c v n x)
 (GF_element_type (subtree e 'element_list) c v n x))

(if (rule e (prodn (tag 'set_or_sequence_value 's)
   (tag 'range 'r)))
    (Gset_or_seq nil
 (GF_element_list (subtree e 'range) c v n x)
 (GF_element_type (subtree e 'range) c v n x))


; *********************
;
;    PARSE TREE LEAVES
;
; *********************
```

```
  (if (character_valuep e)
      (Gchar e)

  (if (digit_listp e)
      (minteger e)

  (if (entry_valuep e)
      (apply_var (entry_name e) v nil)

  (if (identifierp e)
      (Gapply (gname e) nil c v n x)

  (if (string_valuep e)
      (Gstring_seq e)

    (marked (not_expression_error e) (default_value (integer_desc)))

    ))))))))))))))))))))))))))))))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler

))
|#




#|
(do-mutual '(


; ***************************
;  Set/Sequence Constructors
; ***************************

(defn GF_element_list (e c v n x)

  (if (rule e (prodn (tag 'range 'r)
     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GF_element_list (subtree e 'range_limits) c v n x)
```

```
  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'value_list 'v)))
      (GF_element_list (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'range_limits 'r)))
      (GF_element_list (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'range_limits 'r)
     (list (tag 'expression 'lo) 'DOT_DOT
   (tag 'expression 'hi))))
      (Grange_elements (GF (subtree_i e 'expression 1) c v n x)
       (GF (subtree_i e 'expression 2) c v n x))

  (if (rule e (prodn (tag 'value_list 'v)
     (tag 'expression 'e)))
      (rcons nil (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'value_list 'v)
     (list (tag 'value_list 'v2) 'COMMA
   (tag 'expression 'e))))
      (rcons (GF_element_list (subtree e 'value_list) c v n x)
     (GF (subtree e 'expression) c v n x))

    nil))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler

(defn GF_element_type (e c v n x)

  (if (rule e (prodn (tag 'range 'r)
     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GF_element_type (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'value_list 'v)))
      (GF_element_type (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'range_limits 'r)))
      (GF_element_type (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'range_limits 'r)
```

```
     (list (tag 'expression 'lo) 'DOT_DOT
   (tag 'expression 'hi))))
      (base_type (type (GF (subtree_i e 'expression 1) c v n x)))

  (if (rule e (prodn (tag 'value_list 'v)
     (tag 'expression 'e)))
      (base_type (type (GF (subtree e 'expression) c v n x)))

  (if (rule e (prodn (tag 'value_list 'v)
     (list (tag 'value_list 'v2) 'COMMA
   (tag 'expression 'e))))
      (GF_element_type (subtree e 'value_list) c v n x)

    nil))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler



; ************************
;  Quantified expressions
; ************************

(defn GF_all (id vs e c v n x)
  (if (nlistp vs)
      (Gtrue)
  (if (new_namep id v)
      (if (zerop n)
  (marked (n_too_small)
  (default_value (boolean_desc)))
        (Gand (GF_all id (rcdr vs) e c v n x)
      (GF e c (add_to_map v id (rcar vs)) (sub1 n) x)))
    (marked (name_already_in_use_error id)
    (default_value (boolean_desc)))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size id)))
     (count vs))) ))

(defn GF_some (id vs e c v n x)
  (if (nlistp vs)
      (Gfalse)
    (if (new_namep id v)
      (if (zerop n)
```

```
  (marked (n_too_small)
  (default_value (boolean_desc)))
(Gor (GF_some id (rcdr vs) e c v n x)
     (GF e c (add_to_map v id (rcar vs)) (sub1 n) x)))
      (marked (name_already_in_use_error id)
      (default_value (boolean_desc)))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size id)))
     (count vs))) ))


; *********************
;  Value Modifications
; *********************

(defn GF_each (id vs bv e c v n x)
  ; e is the <component modification>
  (if (nlistp vs)
      bv
  (if (new_namep id v)
      (if (zerop n)
  (marked (n_too_small)
  (default_value (base_type (type bv))))
(GF_each id (cdr vs)
 (GF_modifiers bv e c (add_to_map v id (car vs)) (sub1 n) x)
 e c v n x))
    (marked (name_already_in_use_error id)
    (default_value (base_type (type bv))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count vs))) ))

(defn GF_adp (e c v n x)

  (if (rule e (prodn (tag 'arg_list 'as)
     (list 'OPEN_PAREN (tag 'value_list 'vs)
   'CLOSE_PAREN)))
      (GF_adp (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'value_list 'vs)
     (tag 'expression 'e)))
      (rcons nil (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'value_list 'vs)
```

```
     (list (tag 'value_list 'vs2)
   'COMMA (tag 'expression 'e))))
      (rcons (GF_adp (subtree e 'value_list) c v n x)
     (GF (subtree e 'expression) c v n x))
    nil)))

  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler

(defn GF_selectors (e c v n x)

  (if (rule e (prodn (tag 'selector_list 's)
     (tag 'component_selectors 's2)))
      (GF_selectors (subtree e 'component_selectors) c v n x)

  (if (rule e (prodn (tag 'selector_list 's)
     (list (tag 'selector_list 's2)
   (tag 'component_selectors 's3))))
      (append (GF_selectors (subtree e 'selector_list) c v n x)
      (GF_selectors (subtree e 'component_selectors) c v n x))

  (if (rule e (prodn (tag 'component_selectors 's)
     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (list (marked 'field_name
    (gname (subtree e 'IDENTIFIER))))

  (if (rule e (prodn (tag 'component_selectors 's)
     (tag 'arg_list 'd)))
      (GF_adp (subtree e 'arg_list) c v n x)

  (if (rule e (prodn (tag 'arg_list 'as)
     (list 'OPEN_PAREN (tag 'value_list 'vs)
   'CLOSE_PAREN)))
      (GF_adp (subtree e 'value_list) c v n x)

    nil)))))

  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler

(defn GF_modifiers (bv e c v n x)
  ; e is the <value modifiers>
```

```
(if (rule e (prodn (tag 'value_modifiers 'm)
   (tag 'component_selectors 's)))
    (GF_modifiers bv (subtree e 'component_selectors) c v n x)

(if (rule e (prodn (tag 'component_selectors 's)
   (list 'DOT (tag 'IDENTIFIER 'fn))))
    (record_get bv
(marked 'field_name
(gname (subtree e 'IDENTIFIER)))
(type bv))

(if (rule e (prodn (tag 'component_selectors 's)
   (tag 'arg_list 'd)))
    (select_op bv (GF_adp (subtree e 'arg_list) c v n x))

(if (rule e (prodn (tag 'value_modifiers 'm)
   (tag 'range 'r)))
    (GF_modifiers bv (subtree e 'range) c v n x)

(if (rule e (prodn (tag 'range 'r)
   (list 'OPEN_PAREN (tag 'range_limits 'r2)
 'CLOSE_PAREN)))
    (GF_modifiers bv (subtree e 'range_limits) c v n x)

(if (rule e (prodn (tag 'range_limits 'r)
   (list (tag 'expression 'lo) 'DOT_DOT
 (tag 'expression 'hi))))
    (subsequence_get bv
     (GF (subtree_i e 'expression 1) c v n x)
     (GF (subtree_i e 'expression 2) c v n x))

(if (rule e (prodn (tag 'value_modifiers 'm)
   (tag 'value_alterations 'a)))
    (GF_modifiers bv (subtree e 'value_alterations) c v n x)

(if (rule e (prodn (tag 'value_alterations 'a)
   (list 'WITH 'OPEN_PAREN
 (tag 'component_alterations_list 'al)
 'CLOSE_PAREN)))
    (GF_modifiers bv (subtree e 'component_alterations_list) c v n x)

(if (rule e (prodn (tag 'component_alterations_list 'al)
   (tag 'component_alterations 'a)))
```

```
      (GF_modifiers bv (subtree e 'component_alterations) c v n x)

  (if (rule e (prodn (tag 'component_alterations_list 'al)
     (list (tag 'component_alterations_list 'al2)
   'SEMI_COLON (tag 'component_alterations 'a))))
      (GF_modifiers (GF_modifiers bv (subtree e 'component_alterations_list)
  c v n x)
    (subtree e 'component_alterations) c v n x)

  (if (rule e (prodn (tag 'component_alterations 'as)
     (list (tag 'opt_each_clause 'e)
   (tag 'component_assignment 'a))))
      (if (each_clausep (subtree e 'opt_each_clause))
  (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
    (if (errorp vs)
(marked vs (default_value (base_type (type bv))))
      (GF_each (bound_id (subtree e 'opt_each_clause))
       vs bv (subtree e 'component_assignment) c v n x)))
(GF_modifiers bv (subtree e 'component_assignment) c v n x))

  (if (rule e (prodn (tag 'component_alterations 'as)
     (list (tag 'opt_each_clause 'e)
   (tag 'component_creation 'c))))
      (if (each_clausep (subtree e 'opt_each_clause))
  (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
    (if (errorp vs)
(marked vs (default_value (base_type (type bv))))
      (GF_each (bound_id (subtree e 'opt_each_clause))
       vs bv (subtree e 'component_creation) c v n x)))
(GF_modifiers bv (subtree e 'component_creation) c v n x))

  (if (rule e (prodn (tag 'component_alterations 'as)
     (list (tag 'opt_each_clause 'e)
   (tag 'component_deletion 'd))))
      (if (each_clausep (subtree e 'opt_each_clause))
  (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
    (if (errorp vs)
(marked vs (default_value (base_type (type bv))))
      (GF_each (bound_id (subtree e 'opt_each_clause))
       vs bv (subtree e 'component_deletion) c v n x)))
(GF_modifiers bv (subtree e 'component_deletion) c v n x))

  (if (rule e (prodn (tag 'component_assignment 'a)
     (list (tag 'selector_list 's)
```

```
   'COLON_EQUAL (tag 'expression 'e))))
      (put_op bv
      (GF_selectors (subtree e 'selector_list) c v n x)
      (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'BEFORE (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
    (u (GF (subtree e 'expression) c v n x)))
(put_op bv (rcdr s)
(Gseq_insert_before (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'BEHIND (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
    (u (GF (subtree e 'expression) c v n x)))
(put_op bv (rcdr s)
(Gseq_insert_behind (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'INTO (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
    (u (GF (subtree e 'expression) c v n x)))
(put_op bv (rcdr s)
(Gmap_insert (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_deletion 'd)
     (list 'SEQOMIT (tag 'selector_list 's))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x)))
(put_op bv (rcdr s)
(Gseqomit (select_op bv (rcdr s)) (rcar s))))

  (if (rule e (prodn (tag 'component_deletion 'd)
     (list 'MAPOMIT (tag 'selector_list 's))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x)))
(put_op bv (rcdr s)
(Gmapomit (select_op bv (rcdr s)) (rcar s))))

    (marked (bad_value_modifiers_error e)
    (default_value (base_type (type bv)))))))))))))))))))))))))
```

```
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler



; ***********************************
;  Name references and function calls
; ***********************************

(defn apply_fun (fn d sn n x)
  (let ((h (car (ref fn sn x)))  ; scope fn is declared in
(u (cdr (ref fn sn x)))) ; the function declaration
    (if (or (equal (kind u) 'function)
    (equal (kind u) 'constant))
(let ((ftype (type_desc (result_type u) h nil x))
      (fs (formal_dargs u)))                  ; formals
(let ((a (if (equal (length fs) 0) nil d))    ; actuals
      (s (if (equal (length fs) 0) d nil)))   ; selectors
  (if (zerop (fix n))
      (marked (n_too_small)
      (default_value ftype))
    (select_op
     (let ((arg_err (farg_check fs a h x)))
       (if (equal arg_err nil)
   (let ((v (add_to_map (bind_args fs a h x)
(mk_entry_name 'result)
(std_initial
 (list (marked 'type_descriptor
       ftype))))))
     (if (Gtruep (GF (GF_prec u) h v (sub1 n) x))
 (if (has_defn u ftype)
     (let ((r (GF (Gdefn u ftype)
  h v (sub1 n) x)))
       (if (and (determinate r)
(truep (in_type ftype r)))
   (marked nil
   (typed ftype (value r)))
 (marked
   (indeterminate_fn_result_error fn sn)
   (default_value ftype))))
   (marked (no_function_defn_error fn sn)
   (default_value ftype)))
       (marked (entry_not_true_error fn sn)
```

```
       (default_value ftype))))
 (marked arg_err (default_value ftype))))
     s))))
    (if (equal (kind u) 'error)
(marked u (default_value (integer_desc)))
      (marked (not_function_or_const_error fn sn)
      (default_value (integer_desc))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size fn)))
     (tree_size fn))) ))

(defn Gapply (fn ap sn v n x)
  (if (free_variablep fn v)
      (apply_var fn v ap)
  (if (equal fn 'false)
      (select_op (Gfalse) ap)
  (if (equal fn 'true)
      (select_op (Gtrue) ap)
  (if (type_name_expp fn ap sn x)
      (marked 'type_descriptor
      (type_desc (mk_identifier fn) sn nil x))
  (if (equal fn 'domain)
      (std_domain ap)
  (if (equal fn 'first)
      (std_first ap)
  (if (equal fn 'initial)
      (std_initial ap)
  (if (equal fn 'last)
      (std_last ap)
  (if (equal fn 'lower)
      (std_lower ap)
  (if (equal fn 'max)
      (std_max ap)
  (if (equal fn 'min)
      (std_min ap)
  (if (equal fn 'nonfirst)
      (std_nonfirst ap)
  (if (equal fn 'nonlast)
      (std_nonlast ap)
  (if (equal fn 'null)
      (std_null ap)
  (if (equal fn 'ord)
      (std_ord ap)
  (if (equal fn 'pred)
```

```
      (std_pred ap)
  (if (equal fn 'range)
      (std_range ap)
  (if (equal fn 'scale)
      (std_scale ap)
  (if (equal fn 'size)
      (std_size ap)
  (if (equal fn 'succ)
      (std_succ ap)
  (if (equal fn 'upper)
      (std_upper ap)
    (apply_fun fn ap sn n x))))))))))))))))))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size fn)))
     (add1 (tree_size fn)))) ))



(defn GF (e c v n x)
  ; The meta-function GF(e,c,v,n,x) gives the value that results when the
  ; expression e is evaluated, in the context of scope c with free variables
  ; bound as determined in the environment v, by at most n applications of
  ; functions described by the Gypsy sentence x.
  ;
  ; The domain and range of GF(e,c,v,n,x) are as follows:
  ;
  ;   e is the parse tree representing the expression to be evaluated.
  ;   c is the (litatom) name of the Gypsy scope in which e is evaluated.
  ;   v is the name-value mapping that maps names of free variables
  ;     into their (marked typed) values.
  ;   n is the maximum allowed depth of Gypsy function calls and quantifiers
  ;   x is the parse tree representing the Gypsy sentence that is being
  ;     interpreted.  This is the complete sentence containing the list of all
  ;     available scopes.
  ;   GF(e,c,v,n,x) is the marked, typed value that results from evaluating e.
  ;


  ; ************************
  ;
  ;    <expression> ::= ...
  ;
  ; ************************
```

```
  (if (rule e (prodn (tag 'expression 'e)
     (tag 'modified_primary_value 'm)))
      (GF (subtree e 'modified_primary_value) c v n x)

  (if (rule e (prodn (tag 'expression 'e)
     (list 'ALL (tag 'bound_expression 'b))))
      (let ((vs (bound_values e c x)))
(if (errorp vs)
    (marked vs (default_value (boolean_desc)))
  (GF_all (bound_id (subtree e 'bound_expression))
  vs (cdr_quantified_exp e) c v n x)))

  (if (rule e (prodn (tag 'expression 'e)
     (list 'SOME (tag 'bound_expression 'b))))
      (let ((vs (bound_values e c x)))
(if (errorp vs)
    (marked vs (default_value (boolean_desc)))
  (GF_some (bound_id (subtree e 'bound_expression))
   vs (cdr_quantified_exp e) c v n x)))

  (if (rule e (prodn (tag 'expression 'e)
     (list (tag 'unary_operator 'op) (tag 'expression 'e2))))
      (apply_unary_op (subtree e 'unary_operator)
      (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'expression 'e)
     (list (tag 'expression 'e1) (tag 'binary_operator 'op)
   (tag 'expression 'e2))))
      (apply_binary_op (subtree e 'binary_operator)
       (GF (subtree_i e 'expression 1) c v n x)
       (GF (subtree_i e 'expression 2) c v n x))


  ; ********************************
  ;
  ;    <modified primary value> ::=
  ;
  ; ********************************

  (if (rule e (prodn (tag 'modified_primary_value 'm)
     (tag 'primary_value 'p)))
      (GF (subtree e 'primary_value) c v n x)

  (if (rule e (prodn (tag 'modified_primary_value 'm)
```

```
     (list (tag 'modified_primary_value 'm2)
   (tag 'value_modifiers 'vm))))
      (if (fn_call_formp e)
  (Gapply (object_name (subtree e 'modified_primary_value))
  (GF_adp (arg_list (subtree e 'value_modifiers))
  c v n x)
  c v n x)
(GF_modifiers (GF (subtree e 'modified_primary_value) c v n x)
      (subtree e 'value_modifiers) c v n x))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
     (list (tag 'modified_primary_value 'm2)
   (tag 'actual_condition_parameters 'cp))))
      ; Condition parameters allowed only in <specification expression>
      ; with validation directive (p. 35, Gypsy 2.05 Report).
      ; **** Weed them out before calling GF in that case. ****
      (let ((r (GF (subtree e 'modified_primary_value) c v n x)))
(marked (condition_params_error e)
(default_value (type r))))


  ; ***********************
  ;
  ;    <primary value> ::=
  ;
  ; ***********************

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'literal_value 'l)))
      (GF (subtree e 'literal_value) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'set_or_sequence_value 's)))
      (GF (subtree e 'set_or_sequence_value) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'ENTRY_VALUE 'e)))
      (GF (subtree e 'ENTRY_VALUE) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'IDENTIFIER 'on)))
      (GF (subtree e 'IDENTIFIER) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
```

```
    (tag 'if_expression 'i)))
     (GF (subtree e 'if_expression) c v n x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (list 'OPEN_PAREN (tag 'expression 'e) 'CLOSE_PAREN)))
      (GF (subtree e 'expression) c v n x)


  ; ------------------------------------------------------------------------
  ;
  ;  From here down to parse tree leaves, clauses are in alphabetical order
  ;  by the left-hand side of the productions.  Everything that is a parse
  ;  tree for an expression should be covered.
  ;
  ; ------------------------------------------------------------------------


  ; ***********************
  ;
  ;    <constant body> ::=
  ;
  ; ***********************

  ; I don't think the following needs to be here.  siebert 9/Sept/90
  (if (rule e (prodn (tag 'constant_body 'b)
     (tag 'expression 'e)))
      (GF (subtree e 'expression) c v n x)


  ; ***********************
  ;
  ;    <if expression> ::=
  ;
  ; ***********************

  (if (rule e (prodn (tag 'if_expression 'i)
     (list 'IF (tag 'expression 'b) 'THEN
   (tag 'expression 'p)
   (tag 'if_expression_else_part 'e))))

      ; Note:  this does not require all potential value expressions to be the
      ; same type or all boolean expressions to be boolean.
      (let ((bv (GF (subtree_i e 'expression 1) c v n x)))
(if (indeterminate bv)
```

```
    (marked (mark bv)
    (default_value
      (type (GF (subtree_i e 'expression 2)
c v n x))))
        (if (truep (in_type (boolean_desc) bv))
    (if (Gtruep bv)
(GF (subtree_i e 'expression 2) c v n x)
      (GF (if_else_exp (subtree e 'if_expression_else_part))
  c v n x))
  (marked (if_test_not_boolean_error e c)
  (default_value
    (type (GF (subtree_i e 'expression 2)
      c v n x)))))))


  ; ************************
  ;
  ;    <literal value> ::=
  ;
  ; ************************

  (if (rule e (prodn (tag 'literal_value 'l)
     (tag 'CHARACTER_VALUE 'ch)))
      (GF (subtree e 'CHARACTER_VALUE) c v n x)

  (if (rule e (prodn (tag 'literal_value 'l)
     (tag 'number 'n)))
      (GF (subtree e 'number) c v n x)

  (if (rule e (prodn (tag 'literal_value 'l)
     (tag 'STRING_VALUE 's)))
      (GF (subtree e 'STRING_VALUE) c v n x)


  ; ****************
  ;
  ;    <number> ::=
  ;
  ; ****************

  (if (rule e (prodn (tag 'number 'n)
     (tag 'DIGIT_LIST 's)))
      (minteger e)
```

```
(if (rule e (prodn (tag 'number 'n)
   (list (tag 'base 'b) (tag 'DIGIT_LIST 's))))
    (minteger e)


; ***************************************
;
;    <pre-computable label expression> ::=
;
; ***************************************

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (tag 'number 'n)))
    (GF (subtree e 'number) c v n x)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (list 'MINUS (tag 'number 'n))))
    (apply_unary_op (mk_unary_operator 'MINUS)
    (GF (subtree e 'number) c v n x))

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (tag 'CHARACTER_VALUE 'ch)))
    (GF (subtree e 'CHARACTER_VALUE) c v n x)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (tag 'IDENTIFIER 'i)))
    (GF (subtree e 'IDENTIFIER) c v n x)


; ********************************
;
;    <set or sequence value> ::=
;
; ********************************

(if (rule e (prodn (tag 'set_or_sequence_value 's)
   (list 'OPEN_PAREN (tag 'set_or_seq_mark 'm)
 (tag 'element_list 'e) 'CLOSE_PAREN)))
    (Gset_or_seq (subtree e 'set_or_seq_mark)
 (GF_element_list (subtree e 'element_list) c v n x)
 (GF_element_type (subtree e 'element_list) c v n x))

(if (rule e (prodn (tag 'set_or_sequence_value 's)
   (tag 'range 'r)))
```

```
      (Gset_or_seq nil
   (GF_element_list (subtree e 'range) c v n x)
   (GF_element_type (subtree e 'range) c v n x))


  ; *********************
  ;
  ;    PARSE TREE LEAVES
  ;
  ; *********************

  (if (character_valuep e)
      (Gchar e)

  (if (digit_listp e)
      (minteger e)

  (if (entry_valuep e)
      (apply_var (entry_name e) v nil)

  (if (identifierp e)
      (Gapply (gname e) nil c v n x)

  (if (string_valuep e)
      (Gstring_seq e)

    (marked (not_expression_error e) (default_value (integer_desc)))

    ))))))))))))))))))))))))))))))))))
  ( (ord-lessp (cons (cons (add1 n)
   (add1 (tree_size e)))
     (count c))) )) ; (count c) is a place filler

))
|#

(DEFN MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                    (MUTUAL-FLG AP BV C D E FN ID N SN V VS X)
                                    (CASE MUTUAL-FLG
                                     (GF
                                      (COND
                                       ((RULE E
                                              (PRODN (TAG 'EXPRESSION 'E)
                                                     (TAG 'MODIFIED_PRIMARY_VALUE 'M)))
```

```
(MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                        'GF
                                        T
                                        T
                                        C
                                        T
                                        (SUBTREE E 'MODIFIED_PRIMARY_VALUE)
                                        T
                                        T
                                        N
                                        T
                                        V
                                        T
                                        X))
((RULE E
       (PRODN (TAG 'EXPRESSION 'E)
              (LIST 'ALL
                    (TAG 'BOUND_EXPRESSION 'B))))
 (IF
  (ERRORP (BOUND_VALUES E C X))
  (MARKED (BOUND_VALUES E C X)
          (DEFAULT_VALUE (BOOLEAN_DESC)))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                  'GF_ALL
                                  T T
                                  C T
                                  (CDR_QUANTIFIED_EXP E)
                                  T
                                  (BOUND_ID (SUBTREE E 'BOUND_EXPRESSION))
                                  N T
                                  V
                                  (BOUND_VALUES E C X)
                                  X)))
((RULE E
       (PRODN (TAG 'EXPRESSION 'E)
              (LIST 'SOME
                    (TAG 'BOUND_EXPRESSION 'B))))
 (IF
  (ERRORP (BOUND_VALUES E C X))
  (MARKED (BOUND_VALUES E C X)
          (DEFAULT_VALUE (BOOLEAN_DESC)))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                  'GF_SOME
                                  T T
```

```
                                    C T
                                    (CDR_QUANTIFIED_EXP E)
                                    T
                                    (BOUND_ID (SUBTREE E 'BOUND_EXPRESSION))
                                    N T
                                    V
                                    (BOUND_VALUES E C X)
                                    X)))
((RULE E
       (PRODN (TAG 'EXPRESSION 'E)
              (LIST (TAG 'UNARY_OPERATOR 'OP)
                    (TAG 'EXPRESSION 'E2))))
 (APPLY_UNARY_OP
  (SUBTREE E 'UNARY_OPERATOR)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'EXPRESSION)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
((RULE E
       (PRODN (TAG 'EXPRESSION 'E)
              (LIST (TAG 'EXPRESSION 'E1)
                    (TAG 'BINARY_OPERATOR 'OP)
                    (TAG 'EXPRESSION 'E2))))
 (APPLY_BINARY_OP
  (SUBTREE E 'BINARY_OPERATOR)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF
                                                  T
                                                  T
                                                  C
                                                  T
                                                  (SUBTREE_I E 'EXPRESSION 1)
                                                  T
                                                  T
```

```
                                                N
                                                T
                                                V
                                                T
                                                X)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                'GF
                                                T
                                                T
                                                C
                                                T
                                                (SUBTREE_I E 'EXPRESSION 2)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X)))
((RULE E
       (PRODN (TAG 'MODIFIED_PRIMARY_VALUE 'M)
              (TAG 'PRIMARY_VALUE 'P)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                 'GF
                                                 T
                                                 T
                                                 C
                                                 T
                                                 (SUBTREE E 'PRIMARY_VALUE)
                                                 T
                                                 T
                                                 N
                                                 T
                                                 V
                                                 T
                                                 X))
((RULE E
       (PRODN (TAG 'MODIFIED_PRIMARY_VALUE 'M)
              (LIST (TAG 'MODIFIED_PRIMARY_VALUE 'M2)
                    (TAG 'VALUE_MODIFIERS 'VM))))
 (IF
  (FN_CALL_FORMP E)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
   'GAPPLY
```

```
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                  'GF_ADP
                                  T T C
                                  T
                                  (ARG_LIST (SUBTREE E 'VALUE_MODIFIERS))
                                  T T N
                                  T V T
                                  X)
   T T T T
   (OBJECT_NAME (SUBTREE E 'MODIFIED_PRIMARY_VALUE))
   T N C V T X)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
   'GF_MODIFIERS
   T
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                    'GF
                                    T
                                    T
                                    C
                                    T
                                    (SUBTREE E 'MODIFIED_PRIMARY_VALUE)
                                    T
                                    T
                                    N
                                    T
                                    V
                                    T
                                    X)
   C T
   (SUBTREE E 'VALUE_MODIFIERS)
   T T N T V T X)))
((RULE E
       (PRODN (TAG 'MODIFIED_PRIMARY_VALUE 'M)
              (LIST (TAG 'MODIFIED_PRIMARY_VALUE 'M2)
                    (TAG 'ACTUAL_CONDITION_PARAMETERS
                         'CP))))
 (MARKED
  (CONDITION_PARAMS_ERROR E)
  (DEFAULT_VALUE
   (TYPE
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                    'GF
                                    T
                                    T
```

```
                                        C
                                        T
                                        (SUBTREE E 'MODIFIED_PRIMARY_VALUE)
                                        T
                                        T
                                        N
                                        T
                                        V
                                        T
                                        X)))))
((RULE E
       (PRODN (TAG 'PRIMARY_VALUE 'P)
              (TAG 'LITERAL_VALUE 'L)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                              'GF
                                              T
                                              T
                                              C
                                              T
                                              (SUBTREE E 'LITERAL_VALUE)
                                              T
                                              T
                                              N
                                              T
                                              V
                                              T
                                              X))
((RULE E
       (PRODN (TAG 'PRIMARY_VALUE 'P)
              (TAG 'SET_OR_SEQUENCE_VALUE 'S)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                             'GF
                                             T
                                             T
                                             C
                                             T
                                             (SUBTREE E 'SET_OR_SEQUENCE_VALUE)
                                             T
                                             T
                                             N
                                             T
                                             V
                                             T
                                             X))
```

```
((RULE E
       (PRODN (TAG 'PRIMARY_VALUE 'P)
              (TAG 'ENTRY_VALUE 'E)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'ENTRY_VALUE)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
((RULE E
       (PRODN (TAG 'PRIMARY_VALUE 'P)
              (TAG 'IDENTIFIER 'ON)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                     'GF
                                                     T
                                                     T
                                                     C
                                                     T
                                                     (SUBTREE E 'IDENTIFIER)
                                                     T
                                                     T
                                                     N
                                                     T
                                                     V
                                                     T
                                                     X))
((RULE E
       (PRODN (TAG 'PRIMARY_VALUE 'P)
              (TAG 'IF_EXPRESSION 'I)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                   'GF
                                                   T
                                                   T
                                                   C
                                                   T
                                                   (SUBTREE E 'IF_EXPRESSION)
```

```
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
((RULE E
       (PRODN (TAG 'PRIMARY_VALUE 'P)
              (LIST 'OPEN_PAREN
                    (TAG 'EXPRESSION 'E)
                    'CLOSE_PAREN)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'EXPRESSION)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
((RULE E
       (PRODN (TAG 'CONSTANT_BODY 'B)
              (TAG 'EXPRESSION 'E)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'EXPRESSION)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
((RULE E
```

```
      (PRODN (TAG 'IF_EXPRESSION 'I)
             (LIST 'IF
                   (TAG 'EXPRESSION 'B)
                   'THEN
                   (TAG 'EXPRESSION 'P)
                   (TAG 'IF_EXPRESSION_ELSE_PART 'E))))
(COND
 ((INDETERMINATE
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                              'GF
                                              T
                                              T
                                              C
                                              T
                                              (SUBTREE_I E 'EXPRESSION 1)
                                              T
                                              T
                                              N
                                              T
                                              V
                                              T
                                              X))
  (MARKED
   (MARK
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                              'GF
                                              T
                                              T
                                              C
                                              T
                                              (SUBTREE_I E 'EXPRESSION 1)
                                              T
                                              T
                                              N
                                              T
                                              V
                                              T
                                              X))
   (DEFAULT_VALUE
    (TYPE
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                              'GF
                                              T
                                              T
```

```
                                                C
                                                T
                                                (SUBTREE_I E 'EXPRESSION 2)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X)))))
        ((TRUEP
          (IN_TYPE
           (BOOLEAN_DESC)
           (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                'GF
                                                T
                                                T
                                                C
                                                T
                                                (SUBTREE_I E 'EXPRESSION 1)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X)))
         (IF
          (GTRUEP
           (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                'GF
                                                T
                                                T
                                                C
                                                T
                                                (SUBTREE_I E 'EXPRESSION 1)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X))
          (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
```

```
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE_I E 'EXPRESSION 2)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                        'GF
                        T T C T
                        (IF_ELSE_EXP (SUBTREE E 'IF_EXPRESSION_ELSE_PART))
                        T T N T V T X)))
  (T
   (MARKED
    (IF_TEST_NOT_BOOLEAN_ERROR E C)
    (DEFAULT_VALUE
     (TYPE
      (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE_I E 'EXPRESSION 2)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))))))
((RULE E
       (PRODN (TAG 'LITERAL_VALUE 'L)
              (TAG 'CHARACTER_VALUE 'CH)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
```

```
                                                C
                                                T
                                                (SUBTREE E 'CHARACTER_VALUE)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X))
((RULE E
       (PRODN (TAG 'LITERAL_VALUE 'L)
              (TAG 'NUMBER 'N)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                      'GF
                                                      T
                                                      T
                                                      C
                                                      T
                                                      (SUBTREE E 'NUMBER)
                                                      T
                                                      T
                                                      N
                                                      T
                                                      V
                                                      T
                                                      X))
((RULE E
       (PRODN (TAG 'LITERAL_VALUE 'L)
              (TAG 'STRING_VALUE 'S)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'STRING_VALUE)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
```

```
((RULE E
       (PRODN (TAG 'NUMBER 'N)
              (TAG 'DIGIT_LIST 'S)))
 (MINTEGER E))
((RULE E
       (PRODN (TAG 'NUMBER 'N)
              (LIST (TAG 'BASE 'B)
                    (TAG 'DIGIT_LIST 'S))))
 (MINTEGER E))
((RULE E
       (PRODN (TAG 'PRE_COMPUTABLE_LABEL_EXPRESSION
                   'P)
              (TAG 'NUMBER 'N)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                        'GF
                                                        T
                                                        T
                                                        C
                                                        T
                                                        (SUBTREE E 'NUMBER)
                                                        T
                                                        T
                                                        N
                                                        T
                                                        V
                                                        T
                                                        X))
((RULE E
       (PRODN (TAG 'PRE_COMPUTABLE_LABEL_EXPRESSION
                   'P)
              (LIST 'MINUS (TAG 'NUMBER 'N))))
 (APPLY_UNARY_OP
  (MK_UNARY_OPERATOR 'MINUS)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                        'GF
                                                        T
                                                        T
                                                        C
                                                        T
                                                        (SUBTREE E 'NUMBER)
                                                        T
                                                        T
                                                        N
                                                        T
```

```
                                                    V
                                                    T
                                                    X)))
((RULE E
       (PRODN (TAG 'PRE_COMPUTABLE_LABEL_EXPRESSION
                   'P)
              (TAG 'CHARACTER_VALUE 'CH)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                           'GF
                                           T
                                           T
                                           C
                                           T
                                           (SUBTREE E 'CHARACTER_VALUE)
                                           T
                                           T
                                           N
                                           T
                                           V
                                           T
                                           X))
((RULE E
       (PRODN (TAG 'PRE_COMPUTABLE_LABEL_EXPRESSION
                   'P)
              (TAG 'IDENTIFIER 'I)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                             'GF
                                             T
                                             T
                                             C
                                             T
                                             (SUBTREE E 'IDENTIFIER)
                                             T
                                             T
                                             N
                                             T
                                             V
                                             T
                                             X))
((RULE E
       (PRODN (TAG 'SET_OR_SEQUENCE_VALUE 'S)
              (LIST 'OPEN_PAREN
                    (TAG 'SET_OR_SEQ_MARK 'M)
                    (TAG 'ELEMENT_LIST 'E)
```

```
                    'CLOSE_PAREN)))
 (GSET_OR_SEQ
  (SUBTREE E 'SET_OR_SEQ_MARK)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_ELEMENT_LIST
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'ELEMENT_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_ELEMENT_TYPE
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'ELEMENT_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
((RULE E
       (PRODN (TAG 'SET_OR_SEQUENCE_VALUE 'S)
              (TAG 'RANGE 'R)))
 (GSET_OR_SEQ NIL
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                          'GF_ELEMENT_LIST
                                                          T
                                                          T
                                                          C
                                                          T
                                                          (SUBTREE E 'RANGE)
                                                          T
                                                          T
```

```
                                                                N
                                                                T
                                                                V
                                                                T
                                                                X)
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                                'GF_ELEMENT_TYPE
                                                                T
                                                                T
                                                                C
                                                                T
                                                                (SUBTREE E 'RANGE)
                                                                T
                                                                T
                                                                N
                                                                T
                                                                V
                                                                T
                                                                X)))
  ((CHARACTER_VALUEP E) (GCHAR E))
  ((DIGIT_LISTP E) (MINTEGER E))
  ((ENTRY_VALUEP E)
   (APPLY_VAR (ENTRY_NAME E) V NIL))
  ((IDENTIFIERP E)
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                                    'GAPPLY
                                                                    NIL
                                                                    T
                                                                    T
                                                                    T
                                                                    T
                                                                    (GNAME E)
                                                                    T
                                                                    N
                                                                    C
                                                                    V
                                                                    T
                                                                    X))
  ((STRING_VALUEP E) (GSTRING_SEQ E))
  (T (MARKED (NOT_EXPRESSION_ERROR E)
             (DEFAULT_VALUE (INTEGER_DESC))))))
(GAPPLY
 (COND
  ((FREE_VARIABLEP FN V)
```

```
 (APPLY_VAR FN V AP))
((EQUAL FN 'FALSE)
 (SELECT_OP (GFALSE) AP))
((EQUAL FN 'TRUE)
 (SELECT_OP (GTRUE) AP))
((TYPE_NAME_EXPP FN AP SN X)
 (MARKED 'TYPE_DESCRIPTOR
         (TYPE_DESC (MK_IDENTIFIER FN)
                    SN NIL X)))
((EQUAL FN 'DOMAIN) (STD_DOMAIN AP))
((EQUAL FN 'FIRST) (STD_FIRST AP))
((EQUAL FN 'INITIAL)
 (STD_INITIAL AP))
((EQUAL FN 'LAST) (STD_LAST AP))
((EQUAL FN 'LOWER) (STD_LOWER AP))
((EQUAL FN 'MAX) (STD_MAX AP))
((EQUAL FN 'MIN) (STD_MIN AP))
((EQUAL FN 'NONFIRST)
 (STD_NONFIRST AP))
((EQUAL FN 'NONLAST)
 (STD_NONLAST AP))
((EQUAL FN 'NULL) (STD_NULL AP))
((EQUAL FN 'ORD) (STD_ORD AP))
((EQUAL FN 'PRED) (STD_PRED AP))
((EQUAL FN 'RANGE) (STD_RANGE AP))
((EQUAL FN 'SCALE) (STD_SCALE AP))
((EQUAL FN 'SIZE) (STD_SIZE AP))
((EQUAL FN 'SUCC) (STD_SUCC AP))
((EQUAL FN 'UPPER) (STD_UPPER AP))
(T
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                                  'APPLY_FUN
                                                                  T
                                                                  T
                                                                  T
                                                                  AP
                                                                  T
                                                                  FN
                                                                  T
                                                                  N
                                                                  SN
                                                                  T
                                                                  T
                                                                  X))))
```

```
(APPLY_FUN
 (COND
  ((OR (EQUAL (KIND (CDR (REF FN SN X)))
              'FUNCTION)
       (EQUAL (KIND (CDR (REF FN SN X)))
              'CONSTANT))
   (IF
    (ZEROP (FIX N))
    (MARKED (N_TOO_SMALL)
            (DEFAULT_VALUE (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                      (CAR (REF FN SN X))
                                      NIL
                                      X)))
    (SELECT_OP
     (IF
      (EQUAL
           (FARG_CHECK (FORMAL_DARGS (CDR (REF FN SN X)))
                       (IF (EQUAL (LENGTH (FORMAL_DARGS (CDR (REF FN SN X))))
                                  0)
                           NIL D)
                       (CAR (REF FN SN X))
                       X)
           NIL)
      (IF
       (GTRUEP
        (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
         'GF
         T T
         (CAR (REF FN SN X))
         T
         (GF_PREC (CDR (REF FN SN X)))
         T T
         (SUB1 N)
         T
         (ADD_TO_MAP
          (BIND_ARGS (FORMAL_DARGS (CDR (REF FN SN X)))
                     (IF (EQUAL (LENGTH (FORMAL_DARGS (CDR (REF FN SN X))))
                                0)
                         NIL D)
                     (CAR (REF FN SN X))
                     X)
          (MK_ENTRY_NAME 'RESULT)
          (STD_INITIAL
                   (LIST (MARKED 'TYPE_DESCRIPTOR
```

```
                                (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                           (CAR (REF FN SN X))
                                           NIL
                                           X)))))
  T X))
(IF
 (HAS_DEFN (CDR (REF FN SN X))
           (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                      (CAR (REF FN SN X))
                      NIL X))
 (IF
  (AND
   (DETERMINATE
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
     'GF
     T T
     (CAR (REF FN SN X))
     T
     (GDEFN (CDR (REF FN SN X))
            (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                       (CAR (REF FN SN X))
                       NIL X))
     T T
     (SUB1 N)
     T
     (ADD_TO_MAP
      (BIND_ARGS
                (FORMAL_DARGS (CDR (REF FN SN X)))
                (IF (EQUAL (LENGTH (FORMAL_DARGS (CDR (REF FN SN X))))
                           0)
                    NIL D)
                (CAR (REF FN SN X))
                X)
      (MK_ENTRY_NAME 'RESULT)
      (STD_INITIAL
            (LIST (MARKED 'TYPE_DESCRIPTOR
                          (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                     (CAR (REF FN SN X))
                                     NIL
                                     X)))))
     T X))
   (TRUEP
    (IN_TYPE
     (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
```

```
                (CAR (REF FN SN X))
                NIL X)
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIS
     'GF
     T T
     (CAR (REF FN SN X))
     T
     (GDEFN (CDR (REF FN SN X))
            (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                       (CAR (REF FN SN X))
                       NIL X))
     T T
     (SUB1 N)
     T
     (ADD_TO_MAP
      (BIND_ARGS
                (FORMAL_DARGS (CDR (REF FN SN X)))
                (IF (EQUAL (LENGTH (FORMAL_DARGS (CDR (REF FN SN X))))
                           0)
                    NIL D)
                (CAR (REF FN SN X))
                X)
      (MK_ENTRY_NAME 'RESULT)
      (STD_INITIAL
           (LIST (MARKED 'TYPE_DESCRIPTOR
                         (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                    (CAR (REF FN SN X))
                                    NIL
                                    X)))))
     T X))))
   (MARKED NIL
    (TYPED
     (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                (CAR (REF FN SN X))
                NIL X)
     (VALUE
      (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIS
       'GF
       T T
       (CAR (REF FN SN X))
       T
       (GDEFN (CDR (REF FN SN X))
              (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                         (CAR (REF FN SN X))
```

```
                        NIL X))
       T T
       (SUB1 N)
       T
       (ADD_TO_MAP
        (BIND_ARGS
                (FORMAL_DARGS (CDR (REF FN SN X)))
                (IF (EQUAL (LENGTH (FORMAL_DARGS (CDR (REF FN SN X))))
                           0)
                    NIL D)
                (CAR (REF FN SN X))
                X)
        (MK_ENTRY_NAME 'RESULT)
        (STD_INITIAL
             (LIST (MARKED 'TYPE_DESCRIPTOR
                           (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                      (CAR (REF FN SN X))
                                      NIL
                                      X)))))
       T X))))
   (MARKED (INDETERMINATE_FN_RESULT_ERROR FN SN)
           (DEFAULT_VALUE (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                     (CAR (REF FN SN X))
                                     NIL
                                     X))))
  (MARKED (NO_FUNCTION_DEFN_ERROR FN SN)
          (DEFAULT_VALUE (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                    (CAR (REF FN SN X))
                                    NIL
                                    X))))
 (MARKED (ENTRY_NOT_TRUE_ERROR FN SN)
         (DEFAULT_VALUE (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                                   (CAR (REF FN SN X))
                                   NIL
                                   X))))
(MARKED
     (FARG_CHECK (FORMAL_DARGS (CDR (REF FN SN X)))
                 (IF (EQUAL (LENGTH (FORMAL_DARGS (CDR (REF FN SN X))))
                            0)
                     NIL D)
                 (CAR (REF FN SN X))
                 X)
     (DEFAULT_VALUE (TYPE_DESC (RESULT_TYPE (CDR (REF FN SN X)))
                               (CAR (REF FN SN X))
```

```
                                    NIL
                                    X))))
      (IF (EQUAL (LENGTH (FORMAL_DARGS (CDR (REF FN SN X))))
                 0)
          D NIL))))
  ((EQUAL (KIND (CDR (REF FN SN X)))
          'ERROR)
   (MARKED (CDR (REF FN SN X))
           (DEFAULT_VALUE (INTEGER_DESC))))
  (T (MARKED (NOT_FUNCTION_OR_CONST_ERROR FN SN)
             (DEFAULT_VALUE (INTEGER_DESC))))))
(GF_MODIFIERS
 (COND
  ((RULE E
         (PRODN (TAG 'VALUE_MODIFIERS 'M)
                (TAG 'COMPONENT_SELECTORS 'S)))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                          'GF_MODIFIERS
                                          T
                                          BV
                                          C
                                          T
                                          (SUBTREE E 'COMPONENT_SELECTORS)
                                          T
                                          T
                                          N
                                          T
                                          V
                                          T
                                          X))
  ((RULE E
         (PRODN (TAG 'COMPONENT_SELECTORS 'S)
                (LIST 'DOT (TAG 'IDENTIFIER 'FN))))
   (RECORD_GET BV
               (MARKED 'FIELD_NAME
                       (GNAME (SUBTREE E 'IDENTIFIER)))
               (TYPE BV)))
  ((RULE E
         (PRODN (TAG 'COMPONENT_SELECTORS 'S)
                (TAG 'ARG_LIST 'D)))
   (SELECT_OP BV
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_ADP
                                                    T
```

```
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'ARG_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
((RULE E
       (PRODN (TAG 'VALUE_MODIFIERS 'M)
              (TAG 'RANGE 'R)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                      'GF_MODIFIERS
                                                      T
                                                      BV
                                                      C
                                                      T
                                                      (SUBTREE E 'RANGE)
                                                      T
                                                      T
                                                      N
                                                      T
                                                      V
                                                      T
                                                      X))
((RULE E
       (PRODN (TAG 'RANGE 'R)
              (LIST 'OPEN_PAREN
                    (TAG 'RANGE_LIMITS 'R2)
                    'CLOSE_PAREN)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF_MODIFIERS
                                                  T
                                                  BV
                                                  C
                                                  T
                                                  (SUBTREE E 'RANGE_LIMITS)
                                                  T
                                                  T
                                                  N
                                                  T
```

```
                                                V
                                                T
                                                X))
((RULE E
       (PRODN (TAG 'RANGE_LIMITS 'R)
              (LIST (TAG 'EXPRESSION 'LO)
                    'DOT_DOT
                    (TAG 'EXPRESSION 'HI))))
 (SUBSEQUENCE_GET BV
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                               'GF
                                               T
                                               T
                                               C
                                               T
                                               (SUBTREE_I E 'EXPRESSION 1)
                                               T
                                               T
                                               N
                                               T
                                               V
                                               T
                                               X)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                               'GF
                                               T
                                               T
                                               C
                                               T
                                               (SUBTREE_I E 'EXPRESSION 2)
                                               T
                                               T
                                               N
                                               T
                                               V
                                               T
                                               X)))
((RULE E
       (PRODN (TAG 'VALUE_MODIFIERS 'M)
              (TAG 'VALUE_ALTERATIONS 'A)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                            'GF_MODIFIERS
                                            T
                                            BV
```

```
                                        C
                                        T
                                        (SUBTREE E 'VALUE_ALTERATIONS)
                                        T
                                        T
                                        N
                                        T
                                        V
                                        T
                                        X))
((RULE E
       (PRODN (TAG 'VALUE_ALTERATIONS 'A)
              (LIST 'WITH
                    'OPEN_PAREN
                    (TAG 'COMPONENT_ALTERATIONS_LIST
                         'AL)
                    'CLOSE_PAREN)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                    'GF_MODIFIERS
                                    T
                                    BV
                                    C
                                    T
                                    (SUBTREE E
                                             'COMPONENT_ALTERATIONS_LIST)
                                    T
                                    T
                                    N
                                    T
                                    V
                                    T
                                    X))
((RULE E
       (PRODN (TAG 'COMPONENT_ALTERATIONS_LIST
                   'AL)
              (TAG 'COMPONENT_ALTERATIONS 'A)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                      'GF_MODIFIERS
                                      T
                                      BV
                                      C
                                      T
                                      (SUBTREE E 'COMPONENT_ALTERATIONS)
                                      T
```

```
                                        T
                                        N
                                        T
                                        V
                                        T
                                        X))
((RULE E
       (PRODN (TAG 'COMPONENT_ALTERATIONS_LIST
                   'AL)
              (LIST (TAG 'COMPONENT_ALTERATIONS_LIST
                         'AL2)
                    'SEMI_COLON
                    (TAG 'COMPONENT_ALTERATIONS 'A))))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
  'GF_MODIFIERS
  T
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                    'GF_MODIFIERS
                                    T
                                    BV
                                    C T
                                    (SUBTREE E
                                             'COMPONENT_ALTERATIONS_LIST)
                                    T T
                                    N T
                                    V T
                                    X)
  C T
  (SUBTREE E 'COMPONENT_ALTERATIONS)
  T T N T V T X))
((RULE E
       (PRODN (TAG 'COMPONENT_ALTERATIONS 'AS)
              (LIST (TAG 'OPT_EACH_CLAUSE 'E)
                    (TAG 'COMPONENT_ASSIGNMENT 'A))))
 (IF
  (EACH_CLAUSEP (SUBTREE E 'OPT_EACH_CLAUSE))
  (IF
   (ERRORP (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                         C X))
   (MARKED (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                         C X)
           (DEFAULT_VALUE (BASE_TYPE (TYPE BV))))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                'GF_EACH
```

```
                                    T BV C
                                    T
                                    (SUBTREE E 'COMPONENT_ASSIGNMENT)
                                    T
                                    (BOUND_ID (SUBTREE E 'OPT_EACH_CLAUSE))
                                    N T V
                                    (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                                                  C
                                                  X)
                                    X))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                            'GF_MODIFIERS
                                            T
                                            BV
                                            C
                                            T
                                            (SUBTREE E 'COMPONENT_ASSIGNMENT)
                                            T
                                            T
                                            N
                                            T
                                            V
                                            T
                                            X)))
((RULE E
       (PRODN (TAG 'COMPONENT_ALTERATIONS 'AS)
              (LIST (TAG 'OPT_EACH_CLAUSE 'E)
                    (TAG 'COMPONENT_CREATION 'C))))
 (IF
  (EACH_CLAUSEP (SUBTREE E 'OPT_EACH_CLAUSE))
  (IF
   (ERRORP (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                         C X))
   (MARKED (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                         C X)
           (DEFAULT_VALUE (BASE_TYPE (TYPE BV))))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                    'GF_EACH
                                    T BV C
                                    T
                                    (SUBTREE E 'COMPONENT_CREATION)
                                    T
                                    (BOUND_ID (SUBTREE E 'OPT_EACH_CLAUSE))
                                    N T V
```

```
                                    (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                                                  C
                                                  X)
                                    X))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF_MODIFIERS
                                                  T
                                                  BV
                                                  C
                                                  T
                                                  (SUBTREE E 'COMPONENT_CREATION)
                                                  T
                                                  T
                                                  N
                                                  T
                                                  V
                                                  T
                                                  X)))
((RULE E
       (PRODN (TAG 'COMPONENT_ALTERATIONS 'AS)
              (LIST (TAG 'OPT_EACH_CLAUSE 'E)
                    (TAG 'COMPONENT_DELETION 'D))))
 (IF
  (EACH_CLAUSEP (SUBTREE E 'OPT_EACH_CLAUSE))
  (IF
   (ERRORP (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                         C X))
   (MARKED (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                         C X)
           (DEFAULT_VALUE (BASE_TYPE (TYPE BV))))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                    'GF_EACH
                                    T BV C
                                    T
                                    (SUBTREE E 'COMPONENT_DELETION)
                                    T
                                    (BOUND_ID (SUBTREE E 'OPT_EACH_CLAUSE))
                                    N T V
                                    (BOUND_VALUES (SUBTREE E 'OPT_EACH_CLAUSE)
                                                  C
                                                  X)
                                    X))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF_MODIFIERS
```

```
                                                T
                                                BV
                                                C
                                                T
                                                (SUBTREE E 'COMPONENT_DELETION)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X)))
((RULE E
       (PRODN (TAG 'COMPONENT_ASSIGNMENT 'A)
              (LIST (TAG 'SELECTOR_LIST 'S)
                    'COLON_EQUAL
                    (TAG 'EXPRESSION 'E))))
 (PUT_OP BV
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                      'GF
                                                      T
                                                      T
                                                      C
                                                      T
                                                      (SUBTREE E 'EXPRESSION)
                                                      T
                                                      T
                                                      N
                                                      T
                                                      V
```

```
                                                    T
                                                    X)))
((RULE E
       (PRODN (TAG 'COMPONENT_CREATION 'C)
              (LIST 'BEFORE
                    (TAG 'SELECTOR_LIST 'S)
                    'COLON_EQUAL
                    (TAG 'EXPRESSION 'E))))
 (PUT_OP BV
  (RCDR
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
  (GSEQ_INSERT_BEFORE
   (SELECT_OP BV
    (RCDR
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
   (RCAR
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
```

```
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                       'GF
                                                       T
                                                       T
                                                       C
                                                       T
                                                       (SUBTREE E 'EXPRESSION)
                                                       T
                                                       T
                                                       N
                                                       T
                                                       V
                                                       T
                                                       X))))
((RULE E
       (PRODN (TAG 'COMPONENT_CREATION 'C)
              (LIST 'BEHIND
                    (TAG 'SELECTOR_LIST 'S)
                    'COLON_EQUAL
                    (TAG 'EXPRESSION 'E))))
 (PUT_OP BV
  (RCDR
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
```

```
                                                T
                                                V
                                                T
                                                X))
(GSEQ_INSERT_BEHIND
 (SELECT_OP BV
  (RCDR
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                ’GF_SELECTORS
                                                T
                                                T
                                                C
                                                T
                                                (SUBTREE E ’SELECTOR_LIST)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X)))
  (RCAR
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                ’GF_SELECTORS
                                                T
                                                T
                                                C
                                                T
                                                (SUBTREE E ’SELECTOR_LIST)
                                                T
                                                T
                                                N
                                                T
                                                V
                                                T
                                                X))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                   ’GF
                                                   T
                                                   T
                                                   C
                                                   T
                                                   (SUBTREE E ’EXPRESSION)
                                                   T
```

```
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))))
((RULE E
       (PRODN (TAG 'COMPONENT_CREATION 'C)
              (LIST 'INTO
                    (TAG 'SELECTOR_LIST 'S)
                    'COLON_EQUAL
                    (TAG 'EXPRESSION 'E))))
 (PUT_OP BV
  (RCDR
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
  (GMAP_INSERT
   (SELECT_OP BV
    (RCDR
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
```

```
                                                    X)))
   (RCAR
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                      'GF
                                                      T
                                                      T
                                                      C
                                                      T
                                                      (SUBTREE E 'EXPRESSION)
                                                      T
                                                      T
                                                      N
                                                      T
                                                      V
                                                      T
                                                      X))))
((RULE E
       (PRODN (TAG 'COMPONENT_DELETION 'D)
              (LIST 'SEQOMIT
                    (TAG 'SELECTOR_LIST 'S))))
 (PUT_OP BV
  (RCDR
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
```

```
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
  (GSEQOMIT
   (SELECT_OP BV
    (RCDR
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
    (RCAR
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_SELECTORS
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'SELECTOR_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))))
((RULE E
       (PRODN (TAG 'COMPONENT_DELETION 'D)
              (LIST 'MAPOMIT
                    (TAG 'SELECTOR_LIST 'S))))
 (PUT_OP BV
  (RCDR
```

```
(MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                 'GF_SELECTORS
                                                 T
                                                 T
                                                 C
                                                 T
                                                 (SUBTREE E 'SELECTOR_LIST)
                                                 T
                                                 T
                                                 N
                                                 T
                                                 V
                                                 T
                                                 X))
(GMAPOMIT
 (SELECT_OP BV
  (RCDR
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                 'GF_SELECTORS
                                                 T
                                                 T
                                                 C
                                                 T
                                                 (SUBTREE E 'SELECTOR_LIST)
                                                 T
                                                 T
                                                 N
                                                 T
                                                 V
                                                 T
                                                 X)))
 (RCAR
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                 'GF_SELECTORS
                                                 T
                                                 T
                                                 C
                                                 T
                                                 (SUBTREE E 'SELECTOR_LIST)
                                                 T
                                                 T
                                                 N
                                                 T
                                                 V
```

```
                                                    T
                                                    X)))))
  (T (MARKED (BAD_VALUE_MODIFIERS_ERROR E)
             (DEFAULT_VALUE (BASE_TYPE (TYPE BV)))))))
(GF_SELECTORS
 (COND
  ((RULE E
         (PRODN (TAG 'SELECTOR_LIST 'S)
                (TAG 'COMPONENT_SELECTORS 'S2)))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                            'GF_SELECTORS
                                            T
                                            T
                                            C
                                            T
                                            (SUBTREE E 'COMPONENT_SELECTORS)
                                            T
                                            T
                                            N
                                            T
                                            V
                                            T
                                            X))
  ((RULE E
         (PRODN (TAG 'SELECTOR_LIST 'S)
                (LIST (TAG 'SELECTOR_LIST 'S2)
                      (TAG 'COMPONENT_SELECTORS 'S3))))
   (APPEND
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                              'GF_SELECTORS
                                              T
                                              T
                                              C
                                              T
                                              (SUBTREE E 'SELECTOR_LIST)
                                              T
                                              T
                                              N
                                              T
                                              V
                                              T
                                              X)
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                            'GF_SELECTORS
```

```
                                          T
                                          T
                                          C
                                          T
                                          (SUBTREE E 'COMPONENT_SELECTORS)
                                          T
                                          T
                                          N
                                          T
                                          V
                                          T
                                          X)))
((RULE E
       (PRODN (TAG 'COMPONENT_SELECTORS 'S)
              (LIST 'DOT (TAG 'IDENTIFIER 'FN))))
 (LIST (MARKED 'FIELD_NAME
               (GNAME (SUBTREE E 'IDENTIFIER)))))
((RULE E
       (PRODN (TAG 'COMPONENT_SELECTORS 'S)
              (TAG 'ARG_LIST 'D)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF_ADP
                                                  T
                                                  T
                                                  C
                                                  T
                                                  (SUBTREE E 'ARG_LIST)
                                                  T
                                                  T
                                                  N
                                                  T
                                                  V
                                                  T
                                                  X))
((RULE E
       (PRODN (TAG 'ARG_LIST 'AS)
              (LIST 'OPEN_PAREN
                    (TAG 'VALUE_LIST 'VS)
                    'CLOSE_PAREN)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                'GF_ADP
                                                T
                                                T
                                                C
```

```
                                                    T
                                                    (SUBTREE E 'VALUE_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
  (T NIL)))
(GF_ADP
 (COND
  ((RULE E
         (PRODN (TAG 'ARG_LIST 'AS)
                (LIST 'OPEN_PAREN
                      (TAG 'VALUE_LIST 'VS)
                      'CLOSE_PAREN)))
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_ADP
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'VALUE_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
  ((RULE E
         (PRODN (TAG 'VALUE_LIST 'VS)
                (TAG 'EXPRESSION 'E)))
   (RCONS NIL
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'EXPRESSION)
                                                    T
                                                    T
```

```
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
  ((RULE E
         (PRODN (TAG 'VALUE_LIST 'VS)
                (LIST (TAG 'VALUE_LIST 'VS2)
                      'COMMA
                      (TAG 'EXPRESSION 'E))))
   (RCONS
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF_ADP
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'VALUE_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)
    (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE E 'EXPRESSION)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
    (T NIL)))
  (GF_EACH
   (COND
    ((NLISTP VS) BV)
    ((NEW_NAMEP ID V)
```

```
    (IF
     (ZEROP N)
     (MARKED (N_TOO_SMALL)
             (DEFAULT_VALUE (BASE_TYPE (TYPE BV))))
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
      'GF_EACH
      T
      (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF_MODIFIERS
                                                  T
                                                  BV
                                                  C
                                                  T
                                                  E
                                                  T
                                                  T
                                                  (SUB1 N)
                                                  T
                                                  (ADD_TO_MAP V ID (CAR VS))
                                                  T
                                                  X)
      C T E T ID N T V
      (CDR VS)
      X)))
   (T (MARKED (NAME_ALREADY_IN_USE_ERROR ID)
              (DEFAULT_VALUE (BASE_TYPE (TYPE BV)))))))
(GF_SOME
 (COND
  ((NLISTP VS) (GFALSE))
  ((NEW_NAMEP ID V)
   (IF
    (ZEROP N)
    (MARKED (N_TOO_SMALL)
            (DEFAULT_VALUE (BOOLEAN_DESC)))
    (GOR
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                                  'GF_SOME
                                                                  T
                                                                  T
                                                                  C
                                                                  T
                                                                  E
                                                                  T
                                                                  ID
```

```
                                                                    N
                                                                    T
                                                                    V
                                                                    (RCDR VS)
                                                                    X)
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    E
                                                    T
                                                    T
                                                    (SUB1 N)
                                                    T
                                                    (ADD_TO_MAP V ID (RCAR VS))
                                                    T
                                                    X))))
   (T (MARKED (NAME_ALREADY_IN_USE_ERROR ID)
              (DEFAULT_VALUE (BOOLEAN_DESC))))))
(GF_ALL
 (COND
  ((NLISTP VS) (GTRUE))
  ((NEW_NAMEP ID V)
   (IF
    (ZEROP N)
    (MARKED (N_TOO_SMALL)
            (DEFAULT_VALUE (BOOLEAN_DESC)))
    (GAND
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                                    'GF_ALL
                                                                    T
                                                                    T
                                                                    C
                                                                    T
                                                                    E
                                                                    T
                                                                    ID
                                                                    N
                                                                    T
                                                                    V
                                                                    (RCDR VS)
                                                                    X)
```

```
      (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    E
                                                    T
                                                    T
                                                    (SUB1 N)
                                                    T
                                                    (ADD_TO_MAP V ID (RCAR VS))
                                                    T
                                                    X))))
    (T (MARKED (NAME_ALREADY_IN_USE_ERROR ID)
               (DEFAULT_VALUE (BOOLEAN_DESC))))))
  (GF_ELEMENT_TYPE
   (COND
    ((RULE E
           (PRODN (TAG 'RANGE 'R)
                  (LIST 'OPEN_PAREN
                        (TAG 'RANGE_LIMITS 'R2)
                        'CLOSE_PAREN)))
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                     'GF_ELEMENT_TYPE
                                                     T
                                                     T
                                                     C
                                                     T
                                                     (SUBTREE E 'RANGE_LIMITS)
                                                     T
                                                     T
                                                     N
                                                     T
                                                     V
                                                     T
                                                     X))
    ((RULE E
           (PRODN (TAG 'ELEMENT_LIST 'E)
                  (TAG 'VALUE_LIST 'V)))
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                       'GF_ELEMENT_TYPE
                                                       T
                                                       T
```

```
                                                    C
                                                    T
                                                    (SUBTREE E 'VALUE_LIST)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
((RULE E
       (PRODN (TAG 'ELEMENT_LIST 'E)
              (TAG 'RANGE_LIMITS 'R)))
 (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                   'GF_ELEMENT_TYPE
                                                   T
                                                   T
                                                   C
                                                   T
                                                   (SUBTREE E 'RANGE_LIMITS)
                                                   T
                                                   T
                                                   N
                                                   T
                                                   V
                                                   T
                                                   X))
((RULE E
       (PRODN (TAG 'RANGE_LIMITS 'R)
              (LIST (TAG 'EXPRESSION 'LO)
                    'DOT_DOT
                    (TAG 'EXPRESSION 'HI))))
 (BASE_TYPE
  (TYPE
   (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF
                                                  T
                                                  T
                                                  C
                                                  T
                                                  (SUBTREE_I E 'EXPRESSION 1)
                                                  T
                                                  T
                                                  N
```

```
                                                    T
                                                    V
                                                    T
                                                    X))))
    ((RULE E
           (PRODN (TAG 'VALUE_LIST 'V)
                  (TAG 'EXPRESSION 'E)))
     (BASE_TYPE
      (TYPE
       (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                      'GF
                                                      T
                                                      T
                                                      C
                                                      T
                                                      (SUBTREE E 'EXPRESSION)
                                                      T
                                                      T
                                                      N
                                                      T
                                                      V
                                                      T
                                                      X))))
    ((RULE E
           (PRODN (TAG 'VALUE_LIST 'V)
                  (LIST (TAG 'VALUE_LIST 'V2)
                        'COMMA
                        (TAG 'EXPRESSION 'E))))
     (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                      'GF_ELEMENT_TYPE
                                                      T
                                                      T
                                                      C
                                                      T
                                                      (SUBTREE E 'VALUE_LIST)
                                                      T
                                                      T
                                                      N
                                                      T
                                                      V
                                                      T
                                                      X))
    (T NIL)))
   (OTHERWISE
```

```
(COND
 ((RULE E
        (PRODN (TAG 'RANGE 'R)
               (LIST 'OPEN_PAREN
                     (TAG 'RANGE_LIMITS 'R2)
                     'CLOSE_PAREN)))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF_ELEMENT_LIST
                                                  T
                                                  T
                                                  C
                                                  T
                                                  (SUBTREE E 'RANGE_LIMITS)
                                                  T
                                                  T
                                                  N
                                                  T
                                                  V
                                                  T
                                                  X))
 ((RULE E
        (PRODN (TAG 'ELEMENT_LIST 'E)
               (TAG 'VALUE_LIST 'V)))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                   'GF_ELEMENT_LIST
                                                   T
                                                   T
                                                   C
                                                   T
                                                   (SUBTREE E 'VALUE_LIST)
                                                   T
                                                   T
                                                   N
                                                   T
                                                   V
                                                   T
                                                   X))
 ((RULE E
        (PRODN (TAG 'ELEMENT_LIST 'E)
               (TAG 'RANGE_LIMITS 'R)))
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                  'GF_ELEMENT_LIST
                                                  T
                                                  T
```

```
                                                    C
                                                    T
                                                    (SUBTREE E 'RANGE_LIMITS)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X))
((RULE E
       (PRODN (TAG 'RANGE_LIMITS 'R)
              (LIST (TAG 'EXPRESSION 'LO)
                    'DOT_DOT
                    (TAG 'EXPRESSION 'HI))))
 (GRANGE_ELEMENTS
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE_I E 'EXPRESSION 1)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                    'GF
                                                    T
                                                    T
                                                    C
                                                    T
                                                    (SUBTREE_I E 'EXPRESSION 2)
                                                    T
                                                    T
                                                    N
                                                    T
                                                    V
                                                    T
                                                    X)))
```

```
((RULE E
       (PRODN (TAG 'VALUE_LIST 'V)
              (TAG 'EXPRESSION 'E)))
 (RCONS NIL
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                        'GF
                                                        T
                                                        T
                                                        C
                                                        T
                                                        (SUBTREE E 'EXPRESSION)
                                                        T
                                                        T
                                                        N
                                                        T
                                                        V
                                                        T
                                                        X)))
((RULE E
       (PRODN (TAG 'VALUE_LIST 'V)
              (LIST (TAG 'VALUE_LIST 'V2)
                    'COMMA
                    (TAG 'EXPRESSION 'E))))
 (RCONS
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                        'GF_ELEMENT_LIST
                                                        T
                                                        T
                                                        C
                                                        T
                                                        (SUBTREE E 'VALUE_LIST)
                                                        T
                                                        T
                                                        N
                                                        T
                                                        V
                                                        T
                                                        X)
  (MUTUAL-GF-GAPPLY-APPLY_FUN-GF_MODIFIERS-GF_SELECTORS-GF_ADP-GF_EACH-GF_SOME-GF_ALL-GF_ELEMENT_TYPE-GF_ELEMENT_LIST
                                                        'GF
                                                        T
                                                        T
                                                        C
                                                        T
```

```
                                                                (SUBTREE E 'EXPRESSION)
                                                                T
                                                                T
                                                                N
                                                                T
                                                                V
                                                                T
                                                                X)))
   (T NIL))))
((ORD-LESSP (CASE MUTUAL-FLG
                  (GF (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE E)))
                            (COUNT C)))
                  (GAPPLY (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE FN)))
                                (ADD1 (TREE_SIZE FN))))
                  (APPLY_FUN (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE FN)))
                                   (TREE_SIZE FN)))
                  (GF_MODIFIERS (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE E)))
                                      (COUNT C)))
                  (GF_SELECTORS (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE E)))
                                      (COUNT C)))
                  (GF_ADP (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE E)))
                                (COUNT C)))
                  (GF_EACH (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE E)))
                                 (COUNT VS)))
                  (GF_SOME (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE ID)))
                                 (COUNT VS)))
                  (GF_ALL (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE ID)))
                                (COUNT VS)))
                  (GF_ELEMENT_TYPE (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE E)))
                                         (COUNT C)))
                  (OTHERWISE (CONS (CONS (ADD1 N) (ADD1 (TREE_SIZE E)))
                                   (COUNT C)))))))
```

DEFINITION:
gf $(e,\ c,\ v,\ n,\ x)$
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('gf,
t,
t,
c,
t,
e,
t,
t,
n,

t,
v,
t,
x)

DEFINITION:
gapply (fn, ap, sn, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('gapply,
ap,
t,
t,
t,
t,
fn,
t,
n,
sn,
v,
t,
x)

DEFINITION:
apply_fun (fn, d, sn, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('apply_fun,
t,
t,
t,
d,
t,
fn,
t,
n,
sn,
t,
t,
x)

DEFINITION:
gf_modifiers (bv, e, c, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('gf_modifiers,
t,
bv,
c,
t,
e,

t,
t,
n,
t,
v,
t,
x)

Definition:
gf_selectors (e, c, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('`gf_selectors`,
t,
t,
c,
t,
e,
t,
t,
n,
t,
v,
t,
x)

Definition:
gf_adp (e, c, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('`gf_adp`,
t,
t,
c,
t,
e,
t,
t,
n,
t,
v,
t,
x)

Definition:
gf_each (id, vs, bv, e, c, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('`gf_each`,
t,
bv,

c,
t,
e,
t,
id,
n,
t,
v,
vs,
x)

DEFINITION:
gf_some (id, vs, e, c, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list (`'gf_some`,
t,
t,
c,
t,
e,
t,
id,
n,
t,
v,
vs,
x)

DEFINITION:
gf_all (id, vs, e, c, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list (`'gf_all`,
t,
t,
c,
t,
e,
t,
id,
n,
t,
v,
vs,
x)

DEFINITION:
gf_element_type (e, c, v, n, x)

= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('gf_element_type,
t,
t,
c,
t,
e,
t,
t,
n,
t,
v,
t,
x)

DEFINITION:
gf_element_list (e, c, v, n, x)
= mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element_type-gf_element_list ('gf_element_list,
t,
t,
c,
t,
e,
t,
t,
n,
t,
v,
t,
x)

```
; *********************
;  The Meta Function F
; *********************
```

DEFINITION:
meta_f (e, c, v, n, x)
= **let** *pte* **be** pt (e, 'expression),
ptx **be** pt (x, 'program_description)
in
if *pte* = **nil**
then marked ('expression_syntax_error, **nil**)
elseif *ptx* = **nil**
then marked ('program_description_syntax_error, **nil**)
else gf (*pte*, c, v, n, *ptx*) **endif endlet**

EVENT: Make the library "gf".

Index

access, 46, 60, 177
actual_formal_type_error, 28, 147
add_to_map, 10, 101, 178
adjoin_args_error, 28, 158
alias_id_error, 29, 76
all_gypsy_type_members, 125
all_gypsy_types, 125, 126
all_matches, 10
all_scope_types, 125
all_scope_types_members, 125
all_scopes, 75, 76, 125, 126
all_scopes_car, 125
all_seqs_le_n, 105, 107
all_seqs_n, 105
all_subsets, 104, 107
all_type_units, 125
all_type_units_members, 125
all_units, 75, 76, 126
all_units_car, 126
append_args_error, 29, 161
apply_binary_op, 161
apply_fun, 271
apply_unary_op, 152
apply_var, 174
arg_check, 147–160, 162–174, 177
arg_list, 46, 47, 184
arg_listp, 47, 57
array_desc, 127, 138
array_descp, 82, 85, 95, 148, 163
array_get, 148, 150, 165
array_index_error, 29, 127
array_put, 163, 165
array_value_set, 105, 108
ascii_0, 3, 7, 8, 145
ascii_1, 3
ascii_2, 3
ascii_3, 3
ascii_4, 3
ascii_5, 3
ascii_6, 3
ascii_7, 3
ascii_8, 3
ascii_9, 3, 7, 145
ascii_a, 4, 7, 14, 145
ascii_ack, 1
ascii_and, 3, 12
ascii_at, 4, 12
ascii_b, 4
ascii_back_quote, 5
ascii_backslash, 5
ascii_bel, 1
ascii_bs, 1
ascii_c, 4
ascii_can, 2
ascii_caret, 5
ascii_character_listp, 7, 13, 180
ascii_characterp, 6, 7, 16, 180
ascii_close_brace, 6
ascii_close_bracket, 5, 14
ascii_close_paren, 3, 12, 14
ascii_colon, 3, 12, 13
ascii_comma, 3, 13
ascii_cr, 2
ascii_d, 4
ascii_dash, 3, 13
ascii_dc1, 2
ascii_dc2, 2
ascii_dc3, 2
ascii_dc4, 2
ascii_del, 6
ascii_dle, 2
ascii_dollar, 3
ascii_dot, 3, 13
ascii_double_quote, 3, 16, 147
ascii_e, 4
ascii_em, 2
ascii_enq, 1
ascii_eot, 1
ascii_equal, 4, 12, 13
ascii_esc, 2
ascii_etb, 2
ascii_etx, 1

ascii_exclamation_point, 2
ascii_f, 4, 14, 145
ascii_ff, 2
ascii_fs, 2
ascii_g, 4
ascii_gs, 2
ascii_gt, 4, 13
ascii_h, 4
ascii_ht, 2
ascii_i, 4
ascii_j, 4
ascii_k, 4
ascii_l, 4
ascii_lc_a, 5, 7, 145
ascii_lc_b, 5
ascii_lc_c, 5
ascii_lc_d, 5
ascii_lc_e, 5
ascii_lc_f, 5, 145
ascii_lc_g, 5
ascii_lc_h, 5
ascii_lc_i, 5
ascii_lc_j, 5
ascii_lc_k, 5
ascii_lc_l, 5
ascii_lc_m, 6
ascii_lc_n, 6
ascii_lc_o, 6
ascii_lc_p, 6
ascii_lc_q, 6
ascii_lc_r, 6
ascii_lc_s, 6
ascii_lc_t, 6
ascii_lc_u, 6
ascii_lc_v, 6
ascii_lc_w, 6
ascii_lc_x, 6
ascii_lc_y, 6
ascii_lc_z, 6, 7
ascii_lf, 2
ascii_lt, 4, 13
ascii_m, 4
ascii_n, 4
ascii_nak, 2
ascii_nul, 1
ascii_number_sign, 3
ascii_o, 4
ascii_open_brace, 6
ascii_open_bracket, 5, 14
ascii_open_paren, 3, 13, 14
ascii_p, 4
ascii_percent, 3
ascii_plus, 3, 13
ascii_q, 4
ascii_question, 4
ascii_r, 4
ascii_rs, 2
ascii_s, 4
ascii_semicolon, 4, 13
ascii_si, 2
ascii_single_quote, 3, 14, 15, 24, 25, 178
ascii_slash, 3, 13
ascii_so, 2
ascii_soh, 1
ascii_space, 2, 7
ascii_star, 3, 13
ascii_stx, 1
ascii_sub, 2
ascii_syn, 2
ascii_t, 5
ascii_tilde, 6, 7, 46
ascii_u, 5
ascii_underscore, 5, 15
ascii_us, 2
ascii_v, 5
ascii_vertical_bar, 6
ascii_vt, 2
ascii_w, 5
ascii_x, 5
ascii_y, 5
ascii_z, 5, 7
available_types, 126

bad_string_error, 29, 147
bad_value_modifiers_error, 29
band, 151, 153

base_type, 121, 124, 150–153, 157–161, 163, 164, 166, 167, 169, 172–174
bimp, 151, 154
bind_args, 177
bnot, 151, 152
boolean_desc, 83, 142, 145, 152–154, 157
boolean_typep, 83, 175
bor, 151, 153
bound_boolean_expression, 47, 48, 53
bound_id, 48, 49, 184
bound_id_type, 49, 53
bound_values, 179
bounded_index_typep, 84, 180
bounded_typep, 83, 84, 105, 115

case_exit_list, 50–52, 67
case_exit_list2, 49, 50, 52
cdr_quantified_exp, 53
cdr_quantified_names, 52, 53
char_digit, 145, 146
character_desc, 83, 142, 145, 147
character_error, 29, 145
character_typep, 83
character_value_lexemep, 14, 16
character_valuep, 16, 18, 145
close_bracket_lexeme, 14, 16
close_paren_lexeme, 14, 16
colon_gt_args_error, 30
component_assign_error, 30, 166
component_td, 80, 82, 93–95, 102–104, 108, 109, 113–115, 120, 121, 123, 124, 148, 149, 157–159, 163–166, 172
condition_params_error, 30
constant_value_exp, 53, 176
construct_scalar_desc, 132, 140
count_cons, 106
count_key_values, 110
count_keys, 110
crd, 80, 81, 120, 123

default_initial_value, 128, 141
default_value, 119, 145–174
derived_units, 55, 72
determinate, 117, 129, 145, 157, 164–167
difference_args_error, 30, 160
digit_list_lexemep, 15, 17
digit_listp, 17, 18, 24, 146
digit_valid, 146
digit_value, 146
domain_arg_error, 30, 167
dparam_name, 55, 177, 178
dparam_name_list, 55, 175, 177
dtype, 116, 117, 124, 127, 170, 171, 173
dtype_fields, 116
dtype_list, 116
dtype_size, 110, 114, 115
duplicate_field_names_error, 31, 128
duplicate_param_names_error, 31, 177

each_clausep, 55
each_id_type_error, 31, 180
elif_leafp, 181
empty_map, 9, 84, 177
empty_seq_error, 31, 170
empty_type_error, 31, 131
entry_name, 45
entry_not_true_error, 31
entry_value_lexemep, 15, 17, 25
entry_valuep, 17, 18, 45
eq_opp, 175
equality_typep, 84, 128, 152, 175
error_descp, 82, 124, 127, 128, 131, 143, 144
error_msg, 28, 81
errorp, 28, 63, 81, 82, 102–107, 109, 110, 113, 115, 118, 119, 122, 129–132, 143, 147–149, 168–171, 173
expression_from_spec, 56, 67

f_of_formals, 175
farg_check, 177
fformals_check, 177

field_base_types, 121
field_descs, 144
field_list_sets, 106
field_name_reserved_error, 32, 128
field_names, 84, 113, 123, 128
field_td, 80, 149, 164
field_tds, 80, 82, 95, 108, 113, 120, 123
field_tds_equal, 124
field_value_sets, 109
first_arg_error, 32, 168
fn_call_formp, 57, 58
foreign_name, 58, 76
foreign_scope_name, 58, 59, 76
formal_dargs, 59, 60, 175
formal_type, 61, 177, 178
formal_type_list, 176, 177
free_value, 174
free_variablep, 174
fselect, 85, 95, 101, 149
full_dargs, 59, 60
function_access_error, 32, 177
function_defn, 175, 176

gadjoin, 158, 162
gand, 153, 154, 161
gappend, 160–162, 166, 167
gapply, 271
gchar, 145
gcons, 161, 162
gdefn, 176
gdifference, 160, 162
gdiv, 156, 161
gequal, 152–154, 157, 161, 172, 174
gf, 270, 274
gf_adp, 272
gf_all, 273
gf_each, 272
gf_element_list, 274
gf_element_type, 273
gf_modifiers, 271
gf_prec, 175
gf_selectors, 272
gf_some, 273
gfalse, 145, 152–154, 157, 159
gge, 153, 161
ggt, 153, 161, 169
giff, 154, 161
gimp, 154, 161
gin, 157, 159, 162
gintersect, 159, 162
gione, 145, 166–168, 170
gitwo, 145, 170
gizero, 145, 154
gle, 153, 161, 162
glt, 153, 154, 161, 169
gmap_insert, 166
gmapomit, 166
gminus, 151, 152
gmod, 156, 162
gname, 45, 49, 50, 54, 55, 58, 59, 64–66, 70, 71, 142, 143, 183
gne, 152, 161
gnot, 152
gnull_map, 170, 171
gnull_seq, 170, 171
gnull_set, 171
gomit, 158, 162
gor, 153, 161
gplus, 156, 162, 166, 167
gpower, 154, 161
gquotient, 155, 161
grange_elements, 150
grcons, 161, 162
grzero, 145, 154
gseq, 150, 151, 161, 167
gseq_insert_before, 167
gseq_insert_behind, 167
gseqomit, 166
gset, 150, 151, 158
gset_or_seq, 151
gstring_seq, 147
gsub, 159, 162
gsubtract, 156, 162, 166, 167
gsymbol, 20
gtimes, 155, 161
gtrue, 145, 152–154, 157, 159

gtruep, 145, 154, 157, 159, 162, 169, 172, 174
gunion, 158, 162
gypsy_grammar, 17, 19, 20

has_defn, 176

ibase, 61, 146
identifier_lexeme_form, 15
identifier_lexemep, 15, 17, 24, 25
identifierp, 17, 18, 45, 49, 50, 55, 58, 59, 64–66, 70, 71, 142, 183
identifierp_imp_treep, 183
idifference, 7, 157
if_else_exp, 61, 183
if_exp_else_subtrees_car, 181
if_test_not_boolean_error, 32
ileq, 7, 115, 129, 131
ilessp, 153
in_arg_error, 33, 157
in_arg_type, 147
in_map, 10, 23, 85, 174
in_type, 124, 129, 130, 145, 147
indeterminate, 117, 130, 147, 162
indeterminate_fn_result_error, 33
index_set_value_set, 105, 107
index_typep, 84, 127
indexed_value_set, 105
ineg, 152
integer_desc, 83, 117, 119, 120, 123, 124, 129, 142, 145, 146, 148–152, 154–174
integer_descp, 81, 83
integer_typep, 83
integerp, 115, 131
intersect_args_error, 33, 160
intersection, 8
iplus, 7, 156
ipower, 1, 154
iquotient, 156
iremainder, 156
is_digit, 7, 14, 15
is_hexdigit, 14, 15
is_hexdigit_list, 15
is_letter, 7, 15
itimes, 1, 155
izerop, 155, 156

key_value_mapp, 10, 110, 112, 113
key_values, 10, 14, 84, 110
keys, 10, 13, 84, 110
kind, 62, 63, 76, 125, 126, 142, 176, 178

last_arg_error, 34, 168
leaf_equal, 21, 181
leaf_equal_imp_tree_size_equal, 181
leafp, 13, 16–18, 21, 23, 181
length, 8, 9, 14, 16, 55, 67, 76, 101, 104, 110, 118, 125, 126, 132, 144, 150
leq_arg_list_tree_size, 184
leq_bound_id_tree_size, 184
leq_if_exp_else_part, 183
leq_object_name_tree_size, 183
lessp_arg_list_tree_size, 184
lessp_available_types, 126
lessp_bound_id_tree_size, 184
lessp_car_imp_lessp_tree_size, 182
lessp_cdar_tree_size, 86
lessp_cdr_tree_size, 86
lessp_gname_tree_size, 183
lessp_gname_tree_size_0, 183
lessp_if_exp_else_part, 183
lessp_if_than_elif, 182
lessp_key_values, 10
lessp_keys, 10
lessp_list_subtree_size, 22
lessp_list_subtree_than_subtrees, 85
lessp_mapping_domain_tree_size, 85
ze_0, 85
lessp_mk_tree_car_imp_lessp_tree_size, 183

lessp_object_name_tree_size, 183
lessp_plus_cdr_caar_cddar, 106
lessp_rcdr, 9
lessp_remove_length, 9
lessp_subtree_body_size, 22
lessp_subtree_body_than_subtrees, 85
lessp_subtree_i_size, 22
lessp_subtree_size, 22
lessp_subtree_than_subtrees, 85
lessp_subtrees_imp_lessp_tree_size, 181
lexeme, 13, 16, 17, 21, 25, 45, 145–147, 181
lhs, 20
list_cons, 104–106
list_subtree, 21, 22, 85
list_tree_size_not_zero, 85
listp_array_desc_subtrees, 85
listp_cdar_imp_sub1_count_not_zero, 106
listp_if_exp_else_subtrees, 182
listp_imp_count_not_zero, 106
listp_mapping_desc_subtrees, 85
listp_record_desc_subtrees, 85
listp_sequence_desc_subtrees, 85
listp_set_desc_subtrees, 86
local_name, 64, 65, 75, 126
local_names, 75
local_unit_names, 75
lower_pred_error, 34, 172
lt_colon_args_error, 34

many_post_conditions_error, 34, 67
many_scope_error, 34, 76
many_unit_error, 34, 76
map_entry, 10, 101, 109, 119, 121
mapped_value, 10, 16, 23, 80, 85, 115, 116, 124, 174
mapping_desc, 128, 139
mapping_descp, 82, 85, 94, 148, 157–160, 164, 166, 167, 170, 172, 173
mapping_get, 148, 150, 157, 164–166
mapping_merge_arg_check, 157–160
mapping_merge_arg_check2, 157, 158
mapping_merge_error, 35, 103, 104, 158, 160
mapping_put, 164–166
mapping_selector_type_error, 35, 128
mapping_value_set, 106, 108
mark, 117, 129, 147, 166, 167
marked, 77, 117, 130, 131, 145–174, 178, 274
marked_typed, 117
marked_typed_list, 117, 118, 151, 158
marked_typed_value_set, 118, 180
markedp, 77, 117, 118
max_arg_error, 35, 169
max_size, 80, 82, 108, 109, 114, 115, 123, 124
mdigit_value, 146
member_append, 125
meta_f, 274
min_arg_error, 35, 169
minteger, 146
mk_actual, 26, 27
mk_actual_list, 27
mk_arg_list, 26, 27
mk_array_default, 118, 120, 127
mk_array_desc, 78, 82, 120, 127
mk_bound_expression, 27, 28
mk_component_selectors, 27
mk_digit_list, 24, 25
mk_empty, 25
mk_entry_name, 178
mk_entry_value, 25, 175
mk_entry_value_lexeme, 24, 25
mk_error, 28–45, 125, 129–131, 147, 152, 155–157
mk_error_decl, 28, 29, 34, 36, 42, 44
mk_error_kind, 125
mk_expression, 26, 175
mk_identifier, 24–27, 175
mk_identifier_lexeme, 24, 25
mk_identifier_list, 27, 53
mk_integer_desc, 78, 81, 83
mk_literal_value, 25

mk_mapping_desc, 79, 82, 85, 121, 128
mk_modified_primary_value, 26, 175
mk_named_unit, 26, 66
mk_number, 25
mk_pending_desc, 80, 82, 128
mk_primary_value, 25–27
mk_quantified_expression, 28, 53
mk_rational_desc, 78, 81, 83
mk_record_default, 118, 120, 128
mk_record_desc, 79, 82, 120, 128
mk_reserved_word, 23, 24, 26, 28, 62, 181, 182
mk_rhs, 17, 18, 181
mk_rhs_imp_root, 181
mk_rule, 18–20
mk_scalar_const_unit, 26, 54
mk_scalar_desc, 78, 81, 83, 120, 127
mk_sequence_desc, 79, 82, 121, 128
mk_set_desc, 79, 80, 82, 85, 94, 113, 121, 128
mk_single_formal_data_parameter, 26, 59
mk_special_symbol, 23, 24, 26, 27
mk_tree, 11, 23–28, 62, 78–80, 82, 83, 125, 175, 183
mk_true_expression, 27, 67, 68
mk_typed, 116, 117
mk_unary_operator, 24
mk_value_modifiers, 27, 175
mref, 75–77, 126
mref_result, 126
mutual-base_type-field_base_types, 120, 121
mutual-dtype-dtype_fields-dtype_list, 112–116
mutual-gf-gapply-apply_fun-gf_modifiers-gf_selectors-gf_adp-gf_each-gf_some-gf_all-gf_element..., 271–274
mutual-parse_tree_listp-parse_treep, 19, 20
mutual-type_desc-field_descs, 138–144
mutual-type_equal-field_tds_equal, 123, 124
mutual-value_set-field_value_sets, 108, 109
mutual-vmember-vsubsetp-vequal-vset_equal-vsequence_equal-vmapping_equal-vrecord_equal-vfields_e..., 90–101

n_too_small, 35
name_already_in_use_error, 35
named_unit, 66
named_unit_list, 66, 74
namelist_to_actuals, 27, 175
ncopies, 8, 118, 144, 150
negative_exponent_error, 35, 154
new_namep, 174
no_function_defn_error, 36
no_scope_error, 36, 76
no_such_component_error, 36, 85, 97, 101, 148, 149, 162–165
no_such_field_error, 36, 149
no_unit_error, 36, 76
non_rational_simple_typep, 84, 105, 151
non_simple_subrange_type_error, 36, 131
nonfirst_arg_error, 36, 170
nonlast_arg_error, 36, 170
not_array_error, 36, 148, 163
not_binary_op_error, 36, 162
not_defined_on_type_error, 36, 153
not_equality_type_error, 36, 152
not_expression_error, 37
not_function_or_const_error, 37
not_in_set_error, 37, 159
not_in_type_error, 37, 117, 170, 171
not_mapping_error, 37, 148, 164, 166
not_mapping_type_error, 37, 170
not_range_error, 37, 129, 130
not_record_error, 37, 149, 164
not_record_fields_error, 37, 144
not_root_equal_lhs_imp_not_rule, 125
not_selectable_error, 38, 101, 150

not_sequence_error, 38, 149, 163, 165
not_sequence_type_error, 38, 171
not_set_type_error, 38, 171
not_type_descriptor_error, 38, 101, 109, 115, 121
not_type_error, 38, 143
not_unary_op_error, 38, 152
nth, 8, 101
null_map, 84, 121, 128, 170
null_seq, 84, 121, 128, 163, 171
null_set, 84, 94, 113, 121, 128, 171
null_undefined_error, 38, 171
number_error, 38, 146
number_list, 7, 105, 151
number_list2, 7
number_to_char_list, 8, 24

object, 77
object_name, 66, 183
object_namep, 57
omit_args_error, 38, 159
open_bracket_lexeme, 14, 16
open_paren_lexeme, 14, 16
opt_default_value_error, 39, 128
opt_size_limit_error, 39, 129
ord_arg_error, 39, 172

pair_list_map, 10, 118, 144
param_reserved_error, 39, 177
parse_tree_leafp, 18, 19, 181
parse_tree_listp, 19
parse_treep, 20, 181
pending_default_value_error, 40, 128
pending_desc, 128, 140
pending_descp, 82
pending_in_type_error, 40, 115
pending_type_defnp, 67, 140
pending_type_value_set_error, 40, 109
postc, 67, 175
prec, 67, 68, 175
precomputable_f, 77, 128–130
pred_arg_error, 40, 172
printable_char_ordp, 7, 14
prodn, 11, 18, 20, 22, 46–74, 125, 128–130, 138–144, 146, 151, 152, 161, 162, 175, 179, 180, 182–184
prodnp, 20
pt, 20, 274
pt_intro, 20
put_op, 165

quantifier, 53

range_arg_error, 40, 172
range_limits_error, 40, 151
range_max, 129, 130, 142
range_max_setting, 130, 131
range_min, 129, 142
range_min_setting, 130, 131
rational, 1, 83, 145, 155
rational_desc, 83, 120, 142, 145, 151, 152, 154–157
rational_descp, 81, 83
rational_typep, 83, 84
rational_value_set_error, 41, 105
rationalp, 115, 131
rcar, 8, 9, 14–16, 24, 46, 146, 178
rcar_rcons, 9
rcdr, 8, 9, 15, 16, 24, 45, 46, 146, 147, 177
rcdr_rcons, 9
rcons, 9, 25, 46, 54, 55, 59, 66, 69–71, 178
rdifference, 157
record_desc, 128, 139
record_descp, 82, 85, 94, 148, 164
record_field_names, 69, 70, 144
record_get, 148, 150, 165
record_put, 163, 165
record_value_set, 106, 108
reduce, 155
ref, 77, 126, 131, 142, 143, 178
ref_scope, 77, 126, 142
ref_unit, 77, 126, 131, 142, 143, 178
refed_type, 126
remove, 9, 126

remove_larger, 104, 107
requal, 1, 93
reserved_idp, 74, 76, 174, 177
reserved_word_lexemep, 14
reserved_wordp, 16, 18
reserved_words, 12–16, 23, 24, 74
result_type, 70
rhs, 20
rleq, 1, 115, 131
rlessp, 1, 153
rneg, 152
root, 13, 16–19, 21–25, 27, 28, 81, 91, 101, 102, 108, 112, 120, 123, 125, 150, 153, 165, 171, 181
rplus, 156
rpower, 1, 155
rquotient, 155
rtimes, 1, 155
rule, 20, 22, 46–74, 125, 128–130, 138–144, 146, 151, 152, 161, 162, 175, 179, 180, 182–184
rule_imp_lessp_subtree_i_size, 22
rule_imp_lessp_subtree_size, 22
rule_imp_root_equal_lhs, 125
rule_imp_treep, 22
rzerop, 155

same_leaf_imp_same_tree_si
ze1, 181
ze2, 181
scalar_check, 131, 132
scalar_const_units, 54, 55
scalar_desc, 127, 132
scalar_descp, 81, 83
scalar_type_defnp, 70, 140
scalar_typep, 83, 172–174
scalar_value_list, 54, 55, 140
scale_int_arg_error, 41, 173
scale_type_arg_error, 41, 173
scope_id_error, 41, 76
scope_list, 70, 71, 76, 125, 126
scope_name, 71, 75, 125, 126
scope_reserved_error, 42, 76
select_op, 150, 174
selector_td, 81, 82, 94, 96, 101–103, 108, 113, 114, 120, 121, 123, 148, 158, 163, 164, 166, 167
sequence_desc, 128, 139, 147, 150
sequence_descp, 82, 85, 93, 149, 157, 159, 161, 162, 165, 168, 170, 171, 173
sequence_get, 149, 150, 162, 165–168
sequence_put, 164, 165
sequence_value_set, 107, 108
set_default_value, 131, 141
set_desc, 128, 140, 150, 167, 172
set_descp, 82, 86, 93, 157–160, 171, 173
set_difference, 9, 126
set_difference_member, 126
set_difference_remove, 126
set_equal, 9, 113, 123
set_range, 131, 142
set_tmax, 82, 127, 131
set_tmin, 83, 127, 131
set_udv, 83, 127, 131
set_value_set, 107, 109
setp, 113, 128
sid, 81, 82, 120, 123, 124
simple_descp, 81, 82
simple_typep, 83, 84, 131, 153, 169
simple_value_set, 105, 109
size_arg_error, 42, 173
size_limit, 129, 138–140
size_limit_error, 42, 129
some_reserved_idp, 74, 128
special_symbol_lexeme, 23, 24
special_symbol_lexemep, 14
special_symbol_lexemes, 14
special_symbol_map, 12–14, 16, 23
special_symbolp, 16, 18
special_symbols, 13, 16
standard_ids, 74
std_domain, 167
std_first, 168

std_initial, 168
std_last, 168
std_lower, 168, 172
std_max, 169
std_min, 169
std_nonfirst, 169
std_nonlast, 170
std_null, 171
std_ord, 171
std_pred, 172
std_range, 172
std_scale, 172
std_size, 173
std_succ, 174
std_upper, 173, 174
string_char_listp, 15, 16
string_char_seq, 147
string_value_lexemep, 16, 17
string_valuep, 17, 18, 147
sub_args_error, 42, 159
subrange_desc, 127, 131
subsequence_get, 162, 166, 167, 170
subsetp, 9
subst_tree, 23, 82, 83, 175, 176
subtree, 22, 46–74, 82, 83, 85, 128–130, 138–144, 146, 175, 179, 180, 183, 184
subtree_body, 22, 80, 81, 85
subtree_i, 22, 58, 65, 129, 130, 138, 139, 175, 176
subtreep, 23
subtrees, 13, 18, 19, 21–23, 28, 62, 85, 86, 181–183
succ_arg_error, 43, 174

t_or_f, 151
tag, 11, 46–74, 125, 128–130, 138–144, 146, 151, 152, 161, 162, 175, 179–182
taggedp, 20
tdigit_value, 146
tid, 81–83, 115, 120, 123, 124
tmax, 81, 83, 105, 115, 123, 173
tmin, 81, 83, 105, 115, 123, 168
tokenp, 13, 19
tokens, 13
tree_equal, 21, 23, 181, 182
tree_equal_imp_tree_size_equal, 181
tree_size, 20–22, 85, 86, 180–184
tree_size_not_zero, 22
treep, 13, 17–19, 21–23, 28, 86, 181, 183
treep_type_desc, 86
type, 117, 124, 150–153, 157–174
type_defn_cycle_error, 43, 142
type_desc, 144, 177, 178, 180
type_descp, 82, 84, 86, 91, 101, 108, 112, 119, 120, 123, 147
type_equal, 124
type_error, 43
type_error_msg, 81
type_in_all_types, 126
type_name_expp, 178
type_namep, 178
type_part, 117
type_vequal, 121–124
typed, 116, 117, 119, 145–150, 152, 154–161, 163–174, 178
typedp, 117, 118

uc_list, 7, 14–16, 21, 45, 74, 180, 181
uc_list_tree_size, 180
udv, 81, 82, 118, 119, 127, 168
unbounded_sequence_value_set_error, 43, 107
unbounded_type_error, 44, 169, 173
unbounded_value_set_error, 44, 105
union_arg_error, 44, 158
unit_list, 71–76, 125, 126
unit_name, 63–66, 140, 175
unit_reserved_error, 44, 76
unknown_name_error, 44, 174
unmark, 77, 117, 118, 130, 131, 147, 149, 164, 168, 171, 173
untag, 20, 125
upper_case, 7, 14, 180
upper_case_tree_size, 180

upper_succ_error, 45, 174
upper_undefined_error, 45

value, 117, 118, 124, 129, 130, 145, 148, 149, 151–161, 163–167, 169, 170, 172–174, 178
value_part, 118
value_set, 109, 113, 118
value_setting, 130, 131
values, 118, 150
varray_equal, 100
varray_put, 101, 163
vcomponents, 84, 85, 94, 95, 113
vdifference, 103, 160
vdifference_maps, 103, 160
vdomain, 84, 94, 114, 158, 167
vequal, 98, 101–104, 122, 152
vequal_list, 100
vfields_equal, 99
vindexes, 85, 95, 113
vintersect, 103, 159
vintersect_maps, 103, 104, 160
vmap_put, 101, 102
vmap_remove, 103, 166
vmapped_value, 100, 101
vmapped_value_list, 100
vmapping_equal, 99
vmapping_put, 102, 164, 166
vmember, 97, 102–104, 157
vrange, 85, 114, 172
vrecord_equal, 99
vrecord_put, 101, 164
vremove, 103, 159
vselect, 101–104, 148, 149
vselectors, 84, 85
vseq_select, 101
vsequence_equal, 98
vsequence_put, 102, 165
vset, 102, 150
vset_equal, 98, 113
vsetp, 102, 113, 114
vsize, 101, 166–168, 170, 173
vsubmapp, 102
vsubp, 102, 159
vsubseq_select, 101, 163
vsubseqp, 102
vsubsetp, 97, 98, 102
vunion, 104, 158
vunion_maps, 104, 158

zero_count_imp_not_list, 106
zero_divide_error, 45, 155, 156
zero_to_the_zero_power_error, 45, 154