```
;; 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.

;; See also the comment below the DEFN-SK event.
```

EVENT: Start with the library "gf".

```
; ********
;  Errors
; ********
```

DEFINITION:
entry_not_boolean_error (e, sn)
= mk_error (list ('the,
'entry,
'specification,
e,
'in,
'scope,
sn,
'is,
'not,
'a,
'boolean-valued,
'expression))

DEFINITION:
exit_label_error (e, sn)
= mk_error (list ('the,
'exit,
'specification,
e,
'in,
'scope,
sn,
'has,
'duplicate,
'or,
'unknown,
'exit,
'labels))

DEFINITION:
exit_not_boolean_error (e, sn)
= mk_error (list ('the,
 'exit,
 'specification,
 e,
 'in,
 'scope,
 sn,
 'is,
 'not,
 'a,
 'boolean-valued,
 'expression))

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

DEFINITION:
mk_opt_condition_handlers (x)
= mk_tree ('opt_condition_handlers,
 if x = **nil** **then** MK_EMPTY
 else x **endif**)

DEFINITION:
mk_signal_stmt (c)
= mk_tree ('signal_statement,
 list (mk_reserved_word ('signal), mk_identifier (c)))

DEFINITION:
mk_elif_into_if_statement (x)
= mk_tree ('if_composition,
 list (mk_reserved_word ('if),
 subtree (x, 'expression),
 mk_reserved_word ('then),
 subtree (x, 'opt_internal_statements),
 subtree (x, 'if_composition_else_part),
 mk_opt_condition_handlers (**nil**),
 mk_reserved_word ('end)))

DEFINITION:
component_selectors (m)
= **if** rule (m,

```
            prodn (tag ('value_modifiers, 'm),
                   tag ('component_selectors, 's)))
    then subtree (m, 'component_selectors)
    else nil endif
```

Definition:

```
extend_name_selectors (ne, cs)
=   if root (cs) ≠ 'component_selectors then nil
    elseif rule (ne,
                 prodn (tag ('name_expression, 'e),
                        tag ('identifier, 'i)))
    then mk_tree ('name_expression,
                  list (subtree (ne, 'identifier),
                        mk_tree ('selector_list, cs)))
    elseif rule (ne,
                 prodn (tag ('name_expression, 'e),
                        list (tag ('identifier, 'i),
                              tag ('selector_list, 'ss))))
    then mk_tree ('name_expression,
                  list (subtree (ne, 'identifier),
                        mk_tree ('selector_list,
                                 list (subtree (ne, 'selector_list), cs))))
    else nil endif
```

Definition:

```
mk_name_expression (e)
=   if root (e) = 'name_expression then e
    elseif rule (e,
                 prodn (tag ('expression, 'e),
                        tag ('modified_primary_value, 'm)))
    then mk_name_expression (subtree (e, 'modified_primary_value))
    elseif rule (e,
                 prodn (tag ('modified_primary_value, 'm),
                        tag ('primary_value, 'p)))
    then mk_name_expression (subtree (e, 'primary_value))
    elseif rule (e,
                 prodn (tag ('modified_primary_value, 'm),
                        list (tag ('modified_primary_value, 'm2),
                              tag ('value_modifiers, 'vm))))
    then extend_name_selectors (mk_name_expression (subtree (e,
                                                         'modified_primary_value)),
                                component_selectors (subtree (e,
                                                         'value_modifiers)))
    elseif rule (e,
```

```
            prodn (tag ('primary_value, 'p),
                   tag ('identifier, 'on)))
    then mk_name_expression (subtree (e, 'identifier))
    elseif identifierp (e)  then mk_tree ('name_expression, e)
    else nil endif

; ************************************************
;  Parse Tree Extraction and Recognizer Functions
; ************************************************
```

DEFINITION:

```
case_labels (m)
=   if rule (m,
           prodn (tag ('case_composition, 's),
                  list ('case,
                        tag ('expression, 'e),
                        tag ('case_composition_body, 'b),
                        tag ('opt_condition_handlers, 'c),
                        'end)))
    then case_labels (subtree (m, 'case_composition_body))
    elseif rule (m, prodn (tag ('case_composition_body, 'b), 'empty))
    then nil
    elseif rule (m,
             prodn (tag ('case_composition_body, 'b),
                    list ('else,
                          'colon,
                          tag ('opt_internal_statements, 'ss))))
    then nil
    elseif rule (m,
             prodn (tag ('case_composition_body, 'b),
                    list ('is,
                          tag ('case_labels, 'cs),
                          'colon,
                          tag ('opt_internal_statements, 'ss),
                          tag ('case_composition_body, 'b2))))
    then append (case_labels (subtree (m, 'case_labels)),
                 case_labels (subtree (m, 'case_composition_body)))
    elseif rule (m,
             prodn (tag ('case_labels, 'cs),
                    tag ('pre_computable_label_expression, 'e)))
    then rcons (nil, subtree (m, 'pre_computable_label_expression))
    elseif rule (m,
             prodn (tag ('case_labels, 'cs),
```

```
                        list (tag ('case_labels, 'cs2),
                             'comma,
                             tag ('pre_computable_label_expression,
                                  'e))))
    then rcons (case_labels (subtree (m, 'case_labels)),
                subtree (m, 'pre_computable_label_expression))
    else nil endif
```

DEFINITION:

```
constant_body (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 subtree (u, 'constant_body)
    else nil endif
```

DEFINITION:

```
id_list (d)
=   if rule (d,
            prodn (tag ('identifier_list, 'is),
                   list (tag ('identifier_list, 'is2),
                        'comma,
                        tag ('identifier, 'i))))
    then rcons (id_list (subtree (d, 'identifier_list)),
                id_list (subtree (d, 'identifier)))
    elseif rule (d,
                prodn (tag ('identifier_list, 'is),
                       tag ('identifier, 'i)))
    then rcons (nil, id_list (subtree (d, 'identifier)))
    elseif identifierp (d)  then gname (d)
    else nil endif
```

DEFINITION:

```
actual_cargs (m)
=   if rule (m,
            prodn (tag ('procedure_statement, 's),
                   list (tag ('identifier, 'pn),
                        tag ('arg_list, 'dp),
                        tag ('opt_actual_condition_parameters,
                             'cp))))
```

```
    then actual_cargs (subtree (m,
                               'opt_actual_condition_parameters))
    elseif rule (m,
                 prodn (tag ('modified_primary_value, 'm),
                        list (tag ('modified_primary_value, 'm2),
                              tag ('actual_condition_parameters, 'cp))))
    then actual_cargs (subtree (m, 'actual_condition_parameters))
    elseif rule (m,
                 prodn (tag ('opt_actual_condition_parameters, 'cp),
                        'empty)) then nil
    elseif rule (m,
                 prodn (tag ('opt_actual_condition_parameters, 'cp),
                        tag ('actual_condition_parameters, 'cp2)))
    then actual_cargs (subtree (m, 'actual_condition_parameters))
    elseif rule (m,
                 prodn (tag ('actual_condition_parameters, 'cp),
                        list ('unless,
                              'open_paren,
                              tag ('opt_group_name, 'g),
                              tag ('identifier_list, 'is),
                              'close_paren)))
    then id_list (subtree (m, 'identifier_list))
    else nil endif
```

DEFINITION:

```
actual_dargs (m)
=   if rule (m,
             prodn (tag ('procedure_statement, 's),
                    list (tag ('identifier, 'pn),
                          tag ('arg_list, 'dp),
                          tag ('opt_actual_condition_parameters,
                               'cp))))
    then actual_dargs (subtree (m, 'arg_list))
    elseif rule (m,
                 prodn (tag ('arg_list, 'as),
                        list ('open_paren,
                              tag ('value_list, 'vs),
                              'close_paren)))
    then actual_dargs (subtree (m, 'value_list))
    elseif rule (m,
                 prodn (tag ('value_list, 'vs), tag ('expression, 'e)))
    then rcons (nil, subtree (m, 'expression))
    elseif rule (m,
                 prodn (tag ('value_list, 'vs),
```

```
                    list (tag ('value_list, 'vs2),
                          'comma,
                          tag ('expression, 'e))))
    then rcons (actual_dargs (subtree (m, 'value_list)),
                subtree (m, 'expression))
    else nil endif
```

DEFINITION:

```
formal_cargs (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 formal_cargs (subtree (u, 'opt_external_conditions))
    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 formal_cargs (subtree (u, 'opt_external_conditions))
    elseif rule (u,
                 prodn (tag ('opt_external_conditions, 'c), 'empty))
    then nil
    elseif rule (u,
                 prodn (tag ('opt_external_conditions, 'c),
                        list ('unless,
                              'open_paren,
                              'cond,
                              tag ('identifier_list, 'is),
                              'close_paren)))
    then id_list (subtree (u, 'identifier_list))
    else nil endif
```

DEFINITION:

exit_labels (e)

```
=   if rule (e, prodn (tag ('opt_exit_specification, 'e), 'empty))
    then nil
    elseif rule (e,
                prodn (tag ('opt_exit_specification, 'e),
                       list ('exit,
                            tag ('non_validated_specification_expression,
                                'se),
                            'semi_colon)))  then nil
    elseif rule (e,
                prodn (tag ('opt_exit_specification, 'e),
                       list ('exit,
                            tag ('conditional_exit_specification,
                                'c),
                            'semi_colon)))
    then exit_labels (subtree (e, 'conditional_exit_specification))
    elseif rule (e,
                prodn (tag ('conditional_exit_specification, 'c),
                       list ('case,
                            'open_paren,
                            tag ('case_exit_body, 'e),
                            'close_paren)))
    then exit_labels (subtree (e, 'case_exit_body))
    elseif rule (e,
                prodn (tag ('case_exit_body, 'b), tag ('case_exit, 'c)))
    then exit_labels (subtree (e, 'case_exit))
    elseif rule (e,
                prodn (tag ('case_exit_body, 'b),
                       list (tag ('case_exit_body, 'b2),
                            'semi_colon,
                            tag ('case_exit, 'c))))
    then append (exit_labels (subtree (e, 'case_exit_body)),
                 exit_labels (subtree (e, 'case_exit)))
    elseif rule (e,
                prodn (tag ('case_exit, 'ce),
                       list ('is,
                            tag ('case_exit_labels, 'l),
                            'colon,
                            tag ('non_validated_specification_expression,
                                'e))))
    then exit_labels (subtree (e, 'case_exit_labels))
    elseif rule (e,
                prodn (tag ('case_exit_labels, 'ls),
                       list (tag ('case_exit_labels, 'ls2),
                            'comma,
```

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

DEFINITION:

```
exit_spec (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 exit_spec (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 exit_spec (subtree (u, 'procedure_body))
    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 exit_spec (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 subtree (u, 'opt_exit_specification)
    elseif rule (u, prodn (tag ('opt_exit_specification, 'e), 'empty))
    then u
    elseif rule (u,
             prodn (tag ('opt_exit_specification, 'e),
                  list ('exit,
                    tag ('non_validated_specification_expression,
                        'se),
                    'semi_colon))) then u
    elseif rule (u,
             prodn (tag ('opt_exit_specification, 'e),
                  list ('exit,
                    tag ('conditional_exit_specification,
                        'c),
                    'semi_colon))) then u
    else nil endif
```

DEFINITION:

```
handler (m, c)
=   if rule (m, prodn (tag ('opt_condition_handlers, 'c), 'empty))
    then nil
    elseif rule (m,
             prodn (tag ('opt_condition_handlers, 'c),
                  list ('when, tag ('opt_handler_list, 'hs))))
    then handler (subtree (m, 'opt_handler_list), c)
    elseif rule (m, prodn (tag ('opt_handler_list, 'hs), 'empty))
    then nil
    elseif rule (m,
             prodn (tag ('opt_handler_list, 'hs),
                  tag ('handler_list, 'hs2)))
    then handler (subtree (m, 'handler_list), c)
    elseif rule (m, prodn (tag ('handler_list, 'hs), tag ('handler, 'h)))
    then handler (subtree (m, 'handler), c)
    elseif rule (m,
```

```
                prodn (tag ('handler_list, 'hs),
                       list (tag ('handler_list, 'hs2),
                             tag ('handler, 'h))))
    then let r  be  handler (subtree (m, 'handler_list), c)
          in
          if r = nil  then handler (subtree (m, 'handler), c)
          else r endif endlet
    elseif rule (m,
                prodn (tag ('handler, 'h),
                       list ('is,
                             tag ('identifier_list, 'cs),
                             'colon,
                             tag ('opt_internal_statements, 's))))
    then if c ∈ id_list (subtree (m, 'identifier_list))
          then subtree (m, 'opt_internal_statements)
          else nil endif
    else nil endif
```

Definition:

```
handler_labels (m)
=   if rule (m, prodn (tag ('opt_condition_handlers, 'c), 'empty))
    then nil
    elseif rule (m,
                prodn (tag ('opt_condition_handlers, 'c),
                       list ('when, tag ('opt_handler_list, 'hs))))
    then handler_labels (subtree (m, 'opt_handler_list))
    elseif rule (m, prodn (tag ('opt_handler_list, 'hs), 'empty))
    then nil
    elseif rule (m,
                prodn (tag ('opt_handler_list, 'hs),
                       tag ('handler_list, 'hs2)))
    then handler_labels (subtree (m, 'handler_list))
    elseif rule (m, prodn (tag ('handler_list, 'hs), tag ('handler, 'h)))
    then handler_labels (subtree (m, 'handler))
    elseif rule (m,
                prodn (tag ('handler_list, 'hs),
                       list (tag ('handler_list, 'hs2),
                             tag ('handler, 'h))))
    then append (handler_labels (subtree (m, 'handler_list)),
                    handler_labels (subtree (m, 'handler)))
    elseif rule (m,
                prodn (tag ('handler, 'h),
                       list ('is,
                             tag ('identifier_list, 'cs),
```

```
                        'colon,
                        tag ('opt_internal_statements, 's))))
    then id_list (subtree (m, 'identifier_list))
    else nil endif
```

DEFINITION:

```
internal_initial_value_exp (m)
=   if rule (m,
           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 internal_initial_value_exp (subtree (m,
                                            'opt_internal_initial_value))
    elseif rule (m,
               prodn (tag ('opt_internal_initial_value, 'v),
                     'empty))  then nil
    elseif rule (m,
               prodn (tag ('opt_internal_initial_value, 'v),
                     list ('colon_equal, tag ('expression, 'e))))
    then subtree (m, 'expression)
    else nil endif
```

DEFINITION:

```
keep_spec (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 keep_spec (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 keep_spec (subtree (u, 'procedure_body))
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 keep_spec (subtree (u, 'opt_keep_specification))
elseif rule (u, prodn (tag ('opt_keep_specification, 'k), 'empty))
then MK_TRUE_EXPRESSION
elseif rule (u,
          prodn (tag ('opt_keep_specification, 'k),
                list ('keep,
                    tag ('non_validated_specification_expression,
                        'se),
                    'semi_colon)))
then keep_spec (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:

if_statement_else_part (s)
= **if** rule (s,
        prodn (tag (`'if_composition`, `'s`),
            list (`'if`,
                tag (`'expression`, `'b`),
                `'then`,
                tag (`'opt_internal_statements`, `'ss`),
                tag (`'if_composition_else_part`, `'ep`),
                tag (`'opt_condition_handlers`, `'cs`),
                `'end`)))
    then if_statement_else_part (subtree (s,
                `'if_composition_else_part`))
    elseif rule (s,
            prodn (tag (`'if_composition_else_part`, `'ep`),
                `'empty`)) **then nil**
    elseif rule (s,
            prodn (tag (`'if_composition_else_part`, `'ep`),
                list (`'else`,
                    tag (`'opt_internal_statements`, `'ss`))))
    then subtree (s, `'opt_internal_statements`)
    elseif rule (s,
            prodn (tag (`'if_composition_else_part`, `'ep`),
                list (`'elif`,
                    tag (`'expression`, `'b`),
                    `'then`,
                    tag (`'opt_internal_statements`, `'ss`),
                    tag (`'if_composition_else_part`, `'ep2`))))
    then mk_elif_into_if_statement (s)
    else nil endif

DEFINITION:
new_name_arg (m)
= **if** rule (m,
        prodn (tag (`'move_statement`, `'s`),
            list (`'move`,
                tag (`'removable_component`, `'c`),
                tag (`'component_destination`, `'d`))))
    then new_name_arg (subtree (m, `'component_destination`))
    elseif rule (m,
            prodn (tag (`'component_destination`, `'d`),
                tag (`'new_dynamic_variable_component`, `'dc`)))
    then new_name_arg (subtree (m, `'new_dynamic_variable_component`))
    elseif rule (m,
            prodn (tag (`'component_destination`, `'d`),

```
                  list ('to, tag ('name_expression, 'ne))))
then subtree (m, 'name_expression)
elseif rule (m,
           prodn (tag ('new_statement, 's),
                  list ('new,
                      tag ('expression, 'e),
                      tag ('new_dynamic_variable_component,
                          'dc))))
then new_name_arg (subtree (m, 'new_dynamic_variable_component))
elseif rule (m,
           prodn (tag ('new_dynamic_variable_component, 'dc),
                  list ('into, tag ('name_expression, 'ne))))
      ∨   rule (m,
               prodn (tag ('new_dynamic_variable_component,
                          'dc),
                      list ('into,
                          'set,
                          tag ('name_expression, 'ne))))
      ∨   rule (m,
               prodn (tag ('new_dynamic_variable_component,
                          'dc),
                      list ('before,
                          tag ('name_expression, 'ne))))
      ∨   rule (m,
               prodn (tag ('new_dynamic_variable_component,
                          'dc),
                      list ('before,
                          'seq,
                          tag ('name_expression, 'ne))))
      ∨   rule (m,
               prodn (tag ('new_dynamic_variable_component,
                          'dc),
                      list ('behind,
                          tag ('name_expression, 'ne))))
      ∨   rule (m,
               prodn (tag ('new_dynamic_variable_component,
                          'dc),
                      list ('behind,
                          'seq,
                          tag ('name_expression, 'ne))))
then subtree (m, 'name_expression)
else nil endif
```

DEFINITION:

```
remove_exp_arg (m)
=   if rule (m,
            prodn (tag ('move_statement, 's),
                   list ('move,
                       tag ('removable_component, 'c),
                       tag ('component_destination, 'd))))
    then remove_exp_arg (subtree (m, 'removable_component))
    elseif rule (m,
                prodn (tag ('remove_statement, 's),
                       list ('remove, tag ('removable_component, 'c))))
    then remove_exp_arg (subtree (m, 'removable_component))
    elseif rule (m,
                prodn (tag ('removable_component, 'c),
                       list ('element,
                           tag ('expression, 'e),
                           'from,
                           'set,
                           tag ('name_expression, 'ne))))
    then subtree (m, 'expression)
    elseif rule (m,
                prodn (tag ('removable_component, 'c),
                       tag ('name_expression, 'e)))  then nil
    else nil endif
```

Definition:

```
remove_name_arg (m)
=   if rule (m,
            prodn (tag ('move_statement, 's),
                   list ('move,
                       tag ('removable_component, 'c),
                       tag ('component_destination, 'd))))
    then remove_name_arg (subtree (m, 'removable_component))
    elseif rule (m,
                prodn (tag ('remove_statement, 's),
                       list ('remove, tag ('removable_component, 'c))))
    then remove_name_arg (subtree (m, 'removable_component))
    elseif rule (m,
                prodn (tag ('removable_component, 'c),
                       list ('element,
                           tag ('expression, 'e),
                           'from,
                           'set,
                           tag ('name_expression, 'ne))))
    then subtree (m, 'name_expression)
```

```
    elseif rule (m,
              prodn (tag ('removable_component, 'c),
                     tag ('name_expression, 'e)))
    then subtree (m, 'name_expression)
    else nil endif
```

DEFINITION:

```
procedure_body (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))))
      ∨   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, 'procedure_body)
    else nil endif
```

```
; *****************************************************************************
;
;                                  THE STATE
;
; *****************************************************************************

; The state is a marked object <mark , map>.  The mark is either NIL or 'IND,
; which indicates an indeterminate state.  The map is a name-value mapping.
; It maps variables (and constants) to their (marked typed) values.  In
; addition, it contains the special components:
;
;     entry - the (marked typed) value resulting from evaluation of the
;             entry specification on the initial state
;     exit - the (marked typed) value resulting from evaluation of the
;            exit specification on the final state
```

```
;     keep - the (marked typed) value that is the conjunction of results
;            from all evaluations of the keep specification
;     assert - the (marked typed) value that is the conjunction of results
;              from all evaluations of assert specifications
;     keep~ - the parse tree for the keep specification
;     cond~ - the currently active condition
;     result~ - the (marked typed) value resulting from expression evaluation
;     var - a name-value mapping from variable names, arguments and locals,
;           to local|formal
;     const - a name-value mapping from constant names, arguments and locals,
;             to local|formal
;     cond - a name-value mapping from condition names, arguments and locals,
;            to local|formal
```

DEFINITION:
DEFAULT_STATE
= marked (**nil**,
list (cons (`'entry`, GTRUE),
cons (`'exit`, GTRUE),
cons (`'keep`, GTRUE),
cons (`'assert`, GTRUE),
cons (`'keep~`, MK_TRUE_EXPRESSION),
cons (`'cond~`, `'normal`),
cons (`'result~`, GIZERO),
cons (`'var`, **nil**),
cons (`'const`, **nil**),
cons (`'cond`,
`'((routineerror . formal)`
`(spaceerror . formal)`))))

```
; ==========================================
;  Extraction of Components from the State
; ==========================================
```

DEFINITION: map (s) = object (s)

DEFINITION: state_componentp $(k,\ s)$ = in_map (map (s), k)

DEFINITION: state_component $(k,\ s)$ = mapped_value (map (s), k)

DEFINITION: entry (s) = mapped_value (map (s), `'entry`)

DEFINITION: exit (s) = mapped_value (map (s), `'exit`)

DEFINITION: $\text{keep}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'keep})$

DEFINITION: $\text{assert}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'assert})$

DEFINITION: $\text{keep\~{}}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'keep\~{}})$

DEFINITION: $\text{cond\~{}}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'cond\~{}})$

DEFINITION: $\text{condition\_normal}\,(s) = (\text{cond\~{}}\,(s) = \texttt{'normal})$

DEFINITION: $\text{condition\_non\_normal}\,(s) = (\neg\ \text{condition\_normal}\,(s))$

DEFINITION:
$\text{normal\_state}\,(s) = (\text{determinate}\,(s) \wedge \text{condition\_normal}\,(s))$

DEFINITION: $\text{result\~{}}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'result\~{}})$

DEFINITION:
$\text{result\~{}\_list}\,(ss)$
$= \ \textbf{if}\ ss \simeq \textbf{nil}\ \ \textbf{then nil}$
$\quad\ \textbf{else}\ \text{cons}\,(\text{result\~{}}\,(\text{car}\,(ss)),\ \text{result\~{}\_list}\,(\text{cdr}\,(ss)))\ \textbf{endif}$

DEFINITION: $\text{var}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'var})$

DEFINITION: $\text{const}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'const})$

DEFINITION: $\text{cond+}\,(s) = \text{mapped\_value}\,(\text{map}\,(s),\ \texttt{'cond})$

DEFINITION:
$\text{variablep}\,(id,\ s) = (\text{state\_componentp}\,(id,\ s) \wedge \text{in\_map}\,(\text{var}\,(s),\ id))$

DEFINITION: $\text{conditionp}\,(id,\ s) = \text{in\_map}\,(\text{cond+}\,(s),\ id)$

DEFINITION:
$\text{all\_conditionsp}\,(ids,\ s)$
$= \ \textbf{if}\ ids \simeq \textbf{nil}\ \ \textbf{then t}$
$\quad\ \textbf{else}\ \text{conditionp}\,(\text{car}\,(ids),\ s) \wedge \text{all\_conditionsp}\,(\text{cdr}\,(ids),\ s)\ \textbf{endif}$

DEFINITION: $\text{type\_of}\,(n,\ s) = \text{type}\,(\text{state\_component}\,(n,\ s))$

DEFINITION: $\text{mode}\,(td) = \text{root}\,(td)$

DEFINITION:
$\text{locals}\,(cs)$
$= \ \textbf{if}\ cs \simeq \textbf{nil}\ \ \textbf{then nil}$
$\quad\ \textbf{elseif}\ \text{cdar}\,(cs) = \texttt{'local}\ \ \textbf{then}\ \text{cons}\,(\text{car}\,(cs),\ \text{locals}\,(\text{cdr}\,(cs)))$
$\quad\ \textbf{else}\ \text{locals}\,(\text{cdr}\,(cs))\ \textbf{endif}$

DEFINITION: $\text{local\_vars}(s) = \text{locals}(\text{var}(s))$

DEFINITION: $\text{local\_consts}(s) = \text{locals}(\text{const}(s))$

DEFINITION: $\text{local\_conds}(s) = \text{locals}(\text{cond+}(s))$

```
; ================================
;  Setting Components of the State
; ================================
```

DEFINITION: $\text{mark\_state\_indeterminate}(s) = \text{marked}(\texttt{'ind}, \text{map}(s))$

DEFINITION:
$\text{store\_value}(k, v, s) = \text{marked}(\text{mark}(s), \text{add\_to\_map}(\text{map}(s), k, v))$

DEFINITION: $\text{set\_condition}(s, c) = \text{store\_value}(\texttt{'cond\~{}}, c, s)$

DEFINITION:
$\text{store\_result\~{}}(v, s)$
$=$ **if** $\text{determinate}(v)$ **then** $\text{store\_value}(\texttt{'result\~{}}, v, s)$
else $\text{set\_condition}(s, \texttt{'routineerror})$ **endif**

DEFINITION:
$\text{all\_determinate}(vs)$
$=$ **if** $vs \simeq \textbf{nil}$ **then** $vs = \textbf{nil}$
else $\text{determinate}(\text{car}(vs)) \wedge \text{all\_determinate}(\text{cdr}(vs))$ **endif**

DEFINITION:
$\text{reset\_leave\_to\_normal}(s)$
$=$ **if** $\text{cond\~{}}(s) = \texttt{'leave}$ **then** $\text{set\_condition}(s, \texttt{'normal})$
else s **endif**

DEFINITION:
$\text{record\_assertion}(a, s) = \text{store\_value}(\texttt{'assert}, \text{gand}(\text{assert}(s), a), s)$

DEFINITION:
$\text{store\_cond}(id, sc, s) = \text{store\_value}(\texttt{'cond}, \text{cons}(\text{cons}(id, sc), \text{cond+}(s)), s)$

DEFINITION:
$\text{note\_conds}(cs, sc, s)$
$=$ **if** $cs \simeq \textbf{nil}$ **then** s
else $\text{store\_cond}(\text{car}(cs), sc, \text{note\_conds}(\text{cdr}(cs), sc, s))$ **endif**

DEFINITION:
$\text{store\_const}(id, v, sc, s)$
$=$ $\text{store\_value}(\texttt{'const}, \text{cons}(\text{cons}(id, sc), \text{const}(s)), \text{store\_value}(id, v, s))$

DEFINITION:
store_var (*id*, *v*, *sc*, *s*)
= store_value ('var, cons (cons (*id*, *sc*), var (*s*)), store_value (*id*, *v*, *s*))

```
; ===================================
;  Deleting Components from the State
; ===================================
```

DEFINITION:
deallocate (*k*, *s*) = marked (mark (*s*), remove (assoc (*k*, map (*s*)), map (*s*)))

DEFINITION:
deallocate_vars (*vs*, *s*)
= **if** *vs* $\simeq$ **nil then** *s*
else deallocate_vars (cdr (*vs*),
deallocate (caar (*vs*),
store_value ('var,
remove (car (*vs*),
var (*s*)),
s))) **endif**

DEFINITION:
deallocate_consts (*cs*, *s*)
= **if** *cs* $\simeq$ **nil then** *s*
else deallocate_consts (cdr (*cs*),
deallocate (caar (*cs*),
store_value ('const,
remove (car (*cs*),
const (*s*)),
s))) **endif**

DEFINITION:
deallocate_conds (*cs*, *s*)
= **if** *cs* $\simeq$ **nil then** *s*
else deallocate_conds (cdr (*cs*),
store_value ('cond,
remove (car (*cs*), cond+ (*s*)),
s)) **endif**

```
; ================
;  State Equality
; ================
```

DEFINITION:
scomp_equal $(k,\ v1,\ v2)$
$=$ **if** $k \in$ `'(var const cond)` **then** set_equal $(v1,\ v2)$
elseif $k =$ `'cond~` **then** $v1 = v2$
elseif $k =$ `'keep~` **then** tree_equal $(v1,\ v2)$
else gtruep (gequal $(v1,\ v2)$) $\wedge$ (type $(v1)$ = type $(v2)$) **endif**

DEFINITION:
ssubmap $(m1,\ m2)$
$=$ **if** $m1 \simeq$ **nil** **then t**
else in_map $(m2$, caar $(m1))$
$\wedge$ scomp_equal (caar $(m1)$,
cdar $(m1)$,
mapped_value $(m2$, caar $(m1)))$
$\wedge$ ssubmap (cdr $(m1)$, $m2$) **endif**

DEFINITION:
smap_equal $(m1,\ m2)$ = (ssubmap $(m1,\ m2)$ $\wedge$ ssubmap $(m2,\ m1)$)

DEFINITION:
sequal $(s1,\ s2)$
$=$ **if** $s1 = s2$ **then t**
else (mark $(s1)$ = mark $(s2)$) $\wedge$ smap_equal (map $(s1)$, map $(s2)$) **endif**

```
; ********************
;  Constraint Support
; ********************

;; The following was a DEFN-SK+ form, but the three events that
;; would presumably be generated after the DEFN-SK have been
;; inserted below in capital letters.
```

DEFINITION:
implementation_constrained $(s1,\ s0)$
$\leftrightarrow$ $\forall\ k$ (($k \neq$ `'cond~`)
$\rightarrow$ (mapped_value (map $(s1)$, k) = mapped_value (map $(s0)$, k)))

EVENT: Enable implementation_constrained; name this event 'implementation_constrained-off'.

THEOREM: implementation_constrained-suff
((k $(s0,\ s1)$ $\neq$ `'cond~`)
$\rightarrow$ (mapped_value (map $(s1)$, k $(s0,\ s1)$) = mapped_value (map $(s0)$, k $(s0,\ s1)$)))
$\rightarrow$ implementation_constrained $(s1,\ s0)$

THEOREM: implementation_constrained-necc
$(\neg\ ((k \neq$ `'cond~`$)$
$\quad \rightarrow \quad (\text{mapped\_value}(\text{map}(s1),\ k) = \text{mapped\_value}(\text{map}(s0),\ k))))$
$\rightarrow \quad (\neg\ \text{implementation\_constrained}(s1,\ s0))$

DEFINITION:
$\text{state\_check}(ss,\ s0)$
$= \quad$ **if** $(ss \simeq$ **nil**$) \vee (\neg\ \text{normal\_state}(s0))$ **then** $s0$
$\quad$ **elseif** $\text{normal\_state}(\text{car}(ss))$ **then** $\text{state\_check}(\text{cdr}(ss),\ s0)$
$\quad$ **else** $\text{set\_condition}(\text{marked}(\text{mark}(\text{car}(ss))),\ \text{map}(s0)),\ \text{cond\~{}}(\text{car}(ss)))$ **endif**

EVENT: Disable sequal.

EVENT: Disable store_value.

EVENT: Disable cond~.

EVENT: Enable determinate.

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

CONSERVATIVE AXIOM: p_gfalse_intro
let $s1$ **be** $\text{p\_gfalse}(s0)$
in
$((\text{determinate}(s1) \wedge (\text{cond\~{}}(s1) =$ `'normal`$))$
$\rightarrow \quad (\text{sequal}(s1,\ \text{store\_value}($`'result~`$,\ \text{GFALSE},\ s0))$
$\quad \wedge \quad \text{determinate}(\text{GFALSE})))$
$\wedge \quad ((\text{determinate}(s1) \wedge (\text{cond\~{}}(s1) \neq$ `'normal`$))$
$\quad \rightarrow \quad (\text{implementation\_constrained}(s1,\ s0)$
$\quad\quad \wedge \quad (\text{cond\~{}}(s1)$
$\quad\quad\quad \in$ `'(routineerror spaceerror)`$)))$ **endlet**

Simultaneously, we introduce the new function symbol *p_gfalse*.

DEFINITION:
$\text{gpf\_false}(s0)$
$= \quad$ **if** $\text{normal\_state}(s0)$ **then** $\text{p\_gfalse}(s0)$
$\quad$ **else** $s0$ **endif**

Conservative Axiom: p_gtrue_intro

$$
\begin{array}{l}
\textbf{let } \mathit{s1} \textbf{ be } \mathrm{p\_gtrue}\,(\mathit{s0}) \\
\textbf{in} \\
((\mathrm{determinate}\,(\mathit{s1}) \land (\mathrm{cond\tilde{}}\,(\mathit{s1}) = \texttt{'normal}))) \\
\quad \rightarrow \quad (\mathrm{sequal}\,(\mathit{s1}, \mathrm{store\_value}\,(\texttt{'result\tilde{}}, \textsc{gtrue}, \mathit{s0})) \\
\qquad\quad \land \quad \mathrm{determinate}\,(\textsc{gtrue}))) \\
\land \quad ((\mathrm{determinate}\,(\mathit{s1}) \land (\mathrm{cond\tilde{}}\,(\mathit{s1}) \neq \texttt{'normal})) \\
\qquad \rightarrow \quad (\mathrm{implementation\_constrained}\,(\mathit{s1}, \mathit{s0}) \\
\qquad\qquad \land \quad (\mathrm{cond\tilde{}}\,(\mathit{s1}) \\
\qquad\qquad\qquad \in \quad \texttt{'(routineerror spaceerror)}))) \textbf{ endlet}
\end{array}
$$

Simultaneously, we introduce the new function symbol *p_gtrue*.

Definition:

$$
\begin{array}{l}
\mathrm{gpf\_true}\,(\mathit{s0}) \\
= \quad \textbf{if } \mathrm{normal\_state}\,(\mathit{s0}) \textbf{ then } \mathrm{p\_gtrue}\,(\mathit{s0}) \\
\qquad \textbf{else } \mathit{s0} \textbf{ endif}
\end{array}
$$

Conservative Axiom: p_minteger_intro

$$
\begin{array}{l}
\textbf{let } \mathit{s1} \textbf{ be } \mathrm{p\_minteger}\,(e, \mathit{s0}) \\
\textbf{in} \\
((\mathrm{determinate}\,(\mathit{s1}) \land (\mathrm{cond\tilde{}}\,(\mathit{s1}) = \texttt{'normal}))) \\
\quad \rightarrow \quad (\mathrm{sequal}\,(\mathit{s1}, \mathrm{store\_value}\,(\texttt{'result\tilde{}}, \mathrm{minteger}\,(e), \mathit{s0})) \\
\qquad\quad \land \quad \mathrm{determinate}\,(\mathrm{minteger}\,(e)))) \\
\land \quad ((\mathrm{determinate}\,(\mathit{s1}) \land (\mathrm{cond\tilde{}}\,(\mathit{s1}) \neq \texttt{'normal})) \\
\qquad \rightarrow \quad (\mathrm{implementation\_constrained}\,(\mathit{s1}, \mathit{s0}) \\
\qquad\qquad \land \quad (\mathrm{cond\tilde{}}\,(\mathit{s1}) \\
\qquad\qquad\qquad \in \quad \texttt{'(routineerror spaceerror)}))) \textbf{ endlet}
\end{array}
$$

Simultaneously, we introduce the new function symbol *p_minteger*.

Definition:

$$
\begin{array}{l}
\mathrm{gpf\_minteger}\,(e, s) \\
= \quad \textbf{if } \mathrm{normal\_state}\,(s) \textbf{ then } \mathrm{p\_minteger}\,(e, s) \\
\qquad \textbf{else } s \textbf{ endif}
\end{array}
$$

Conservative Axiom: p_gchar_intro

$$
\begin{array}{l}
\textbf{let } \mathit{s1} \textbf{ be } \mathrm{p\_gchar}\,(e, \mathit{s0}) \\
\textbf{in} \\
((\mathrm{determinate}\,(\mathit{s1}) \land (\mathrm{cond\tilde{}}\,(\mathit{s1}) = \texttt{'normal}))) \\
\quad \rightarrow \quad (\mathrm{sequal}\,(\mathit{s1}, \mathrm{store\_value}\,(\texttt{'result\tilde{}}, \mathrm{gchar}\,(e), \mathit{s0})) \\
\qquad\quad \land \quad \mathrm{determinate}\,(\mathrm{gchar}\,(e)))) \\
\land \quad ((\mathrm{determinate}\,(\mathit{s1}) \land (\mathrm{cond\tilde{}}\,(\mathit{s1}) \neq \texttt{'normal})) \\
\qquad \rightarrow \quad (\mathrm{implementation\_constrained}\,(\mathit{s1}, \mathit{s0}) \\
\qquad\qquad \land \quad (\mathrm{cond\tilde{}}\,(\mathit{s1}) \\
\qquad\qquad\qquad \in \quad \texttt{'(routineerror spaceerror)}))) \textbf{ endlet}
\end{array}
$$

Simultaneously, we introduce the new function symbol *p_gchar*.

DEFINITION:
gpf_gchar (e, s)
= **if** normal_state (s) **then** p_gchar (e, s)
else s **endif**

CONSERVATIVE AXIOM: p_gstring_seq_intro
let $s1$ **be** p_gstring_seq (e, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gstring_seq (e), $s0$))
$\wedge$ determinate (gstring_seq (e))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gstring_seq*.

DEFINITION:
gpf_gstring_seq (e, s)
= **if** normal_state (s) **then** p_gstring_seq (e, s)
else s **endif**

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

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

EVENT: Disable array_get.

CONSERVATIVE AXIOM: p_array_get_intro
let $s1$ **be** p_array_get (a, i, td, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, array_get (a, i, td), $s0$))
$\wedge$ determinate (array_get (a, i, td))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_array_get*.

DEFINITION:
gpf_array_get (sa, si, $s0$)
= **let** r **be** state_check (list (sa, si), $s0$)
in
if normal_state (r)
then p_array_get (result˜ (sa), result˜ (si), type (result˜ (sa)), $s0$)
else r **endif endlet**

EVENT: Disable mapping_get.

CONSERVATIVE AXIOM: p_mapping_get_intro
let $s1$ **be** p_mapping_get (m, d, td, $s0$)
in
((determinate ($s1$) $\land$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, mapping_get (m, d, td), $s0$))
$\land$ determinate (mapping_get (m, d, td))))
$\land$ ((determinate ($s1$) $\land$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\land$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_mapping_get*.

DEFINITION:
gpf_mapping_get (sm, sd, $s0$)
= **let** r **be** state_check (list (sm, sd), $s0$)
in
if normal_state (r)
then p_mapping_get (result˜ (sm),
result˜ (sd),
type (result˜ (sm)),
$s0$)
else r **endif endlet**

EVENT: Disable record_get.

CONSERVATIVE AXIOM: p_record_get_intro
let $s1$ **be** p_record_get (r, fn, td, $s0$)
in
((determinate ($s1$) $\land$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, record_get (r, fn, td), $s0$))
$\land$ determinate (record_get (r, fn, td))))

```
∧   ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
      →   (implementation_constrained (s1, s0)
             ∧   (cond˜ (s1)
                    ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_record_get*.

DEFINITION:

```
gpf_record_get (sr, sfn, s0)
=   let r  be  state_check (list (sr, sfn), s0)
    in
    if normal_state (r)
    then p_record_get (result˜ (sr),
                       result˜ (sfn),
                       type (result˜ (sr)),
                       s0)
    else r endif endlet
```

EVENT: Disable sequence_get.

CONSERVATIVE AXIOM: p_sequence_get_intro

```
let s1  be  p_sequence_get (s, i, td, s0)
in
((determinate (s1) ∧ (cond˜ (s1) = 'normal))
 →   (sequal (s1, store_value ('result˜, sequence_get (s, i, td), s0))
       ∧   determinate (sequence_get (s, i, td))))
∧   ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
      →   (implementation_constrained (s1, s0)
             ∧   (cond˜ (s1)
                    ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_sequence_get*.

DEFINITION:

```
gpf_sequence_get (ss, si, s0)
=   let r  be  state_check (list (ss, si), s0)
    in
    if normal_state (r)
    then p_sequence_get (result˜ (ss),
                         result˜ (si),
                         type (result˜ (ss)),
                         s0)
    else r endif endlet
```

EVENT: Disable state_check.

EVENT: Disable normal_state.

EVENT: Disable mode.

EVENT: Disable type.

EVENT: Disable result.

EVENT: Disable gpf_array_get.

EVENT: Disable gpf_record_get.

EVENT: Disable gpf_mapping_get.

EVENT: Disable gpf_sequence_get.

EVENT: Disable store_result.

EVENT: Disable not_selectable_error.

EVENT: Disable default_value.

EVENT: Disable integer_desc.

EVENT: Disable *1*integer_desc.

```
;; >> It might be better to just constrain this and forget the four defn's
;; above.
```

DEFINITION:
gpf_select_op (sv, ss, $s0$)
= **let** r **be** state_check (cons (sv, ss), $s0$)
in
if ¬ normal_state (r) **then** r

```
    elseif ss ≃ nil then sv
    else case on mode (type (result˜ (sv))):
         case = array
         then gpf_select_op (gpf_array_get (sv, car (ss), s0),
                             cdr (ss),
                             s0)
         case = record
          then gpf_select_op (gpf_record_get (sv, car (ss), s0),
                              cdr (ss),
                              s0)
         case = mapping
          then gpf_select_op (gpf_mapping_get (sv, car (ss), s0),
                              cdr (ss),
                              s0)
         case = sequence
          then gpf_select_op (gpf_sequence_get (sv, car (ss), s0),
                              cdr (ss),
                              s0)
         otherwise store_result˜ (marked (not_selectable_error (result˜ (sv)),
                                          default_value (INTEGER_DESC)),
                                  s0) endcase endif endlet
```

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

EVENT: Disable subsequence_get.

CONSERVATIVE AXIOM: p_subsequence_get_intro

```
 let s1  be p_subsequence_get (s, lo, hi, s0)
 in
 ((determinate (s1) ∧ (cond˜ (s1) = 'normal))
  →   (sequal (s1,
              store_value ('result˜, subsequence_get (s, lo, hi), s0))
        ∧   determinate (subsequence_get (s, lo, hi))))
 ∧   ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
       →   (implementation_constrained (s1, s0)
             ∧   (cond˜ (s1)
                   ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_subsequence_get*.

DEFINITION:

gpf_subsequence_get (ss, slo, shi, $s0$)
= **let** r **be** state_check (list (ss, slo, shi), $s0$)
in
if normal_state (r)
then p_subsequence_get (result˜ (ss),
result˜ (slo),
result˜ (shi),
$s0$)
else r **endif endlet**

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

Event: Disable array_put.

Conservative Axiom: p_array_put_intro
let $s1$ **be** p_array_put (a, i, v, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, array_put (a, i, v), $s0$))
$\wedge$ determinate (array_put (a, i, v))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_array_put*.

Definition:
gpf_array_put (sa, si, sv, $s0$)
= **let** r **be** state_check (list (sa, si, sv), $s0$)
in
if normal_state (r)
then p_array_put (result˜ (sa), result˜ (si), result˜ (sv), $s0$)
else r **endif endlet**

Event: Disable record_put.

Conservative Axiom: p_record_put_intro
let $s1$ **be** p_record_put (r, fn, v, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))

$\rightarrow$ (sequal (*s1*, store_value (`'result~`, record_put (*r*, *fn*, *v*), *s0*))
  $\wedge$ determinate (record_put (*r*, *fn*, *v*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_record_put*.

DEFINITION:
gpf_record_put (*sr*, *sfn*, *sv*, *s0*)
= **let** *r* **be** state_check (list (*sr*, *sfn*, *sv*), *s0*)
  in
  if normal_state (*r*)
  then p_record_put (result~ (*sr*), result~ (*sfn*), result~ (*sv*), *s0*)
  else *r* **endif endlet**

EVENT: Disable mapping_put.

CONSERVATIVE AXIOM: p_mapping_put_intro
let *s1* **be** p_mapping_put (*m*, *d*, *v*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = `'normal`))
 $\rightarrow$ (sequal (*s1*, store_value (`'result~`, mapping_put (*m*, *d*, *v*), *s0*))
  $\wedge$ determinate (mapping_put (*m*, *d*, *v*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_mapping_put*.

DEFINITION:
gpf_mapping_put (*sm*, *sd*, *sv*, *s0*)
= **let** *r* **be** state_check (list (*sm*, *sd*, *sv*), *s0*)
  in
  if normal_state (*r*)
  then p_mapping_put (result~ (*sm*), result~ (*sd*), result~ (*sv*), *s0*)
  else *r* **endif endlet**

EVENT: Disable sequence_put.

CONSERVATIVE AXIOM: p_sequence_put_intro
let *s1* **be** p_sequence_put (*s*, *i*, *v*, *s0*)

```
in
((determinate (s1) ∧ (cond˜ (s1) = 'normal))
 →   (sequal (s1, store_value ('result˜, sequence_put (s, i, v), s0))
        ∧   determinate (sequence_put (s, i, v))))
∧   ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
       →   (implementation_constrained (s1, s0)
              ∧   (cond˜ (s1)
                     ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_sequence_put*.

Definition:

```
gpf_sequence_put (ss, si, sv, s0)
=   let r  be  state_check (list (ss, si, sv), s0)
    in
    if normal_state (r)
    then p_sequence_put (result˜ (ss), result˜ (si), result˜ (sv), s0)
    else r endif endlet
```

Definition:

```
gpf_put_op (sbv, ss, sv, s0)
=   let r  be  state_check (cons (sbv, rcons (ss, sv)), s0)
    in
    if ¬ normal_state (r)  then  r
    elseif ss ≃ nil  then  store_result˜ (result˜ (sv), s0)
    else case on mode (type (result˜ (sbv))):
         case = array
         then gpf_array_put (sbv,
                             car (ss),
                             gpf_put_op (gpf_array_get (sbv,
                                                        car (ss),
                                                        s0),
                                         cdr (ss),
                                         sv,
                                         s0),
                             s0)
         case = record
          then gpf_record_put (sbv,
                               car (ss),
                               gpf_put_op (gpf_record_get (sbv,
                                                           car (ss),
                                                           s0),
                                           cdr (ss),
                                           sv,
                                           s0),
```

$s0$)
case = *mapping*
then gpf_mapping_put (sbv,
car (ss),
gpf_put_op (gpf_mapping_get (sbv,
car (ss),
$s0$),
cdr (ss),
sv,
$s0$),
$s0$)
case = *sequence*
then gpf_sequence_put (sbv,
car (ss),
gpf_put_op (gpf_sequence_get (sbv,
car (ss),
$s0$),
cdr (ss),
sv,
$s0$),
$s0$)
otherwise store_result˜ (marked (component_assign_error (result˜ (sbv)),
default_value (INTEGER_DESC)),
$s0$) **endcase endif endlet**

EVENT: Disable gmapomit.

CONSERVATIVE AXIOM: p_gmapomit_intro
let $s1$ **be** p_gmapomit (m, i, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gmapomit (m, i), $s0$))
$\wedge$ determinate (gmapomit (m, i))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gmapomit*.

DEFINITION:
gpf_gmapomit (sm, si, $s0$)
= **let** r **be** state_check (list (sm, si), $s0$)
in

if normal_state (r) **then** p_gmapomit (result˜ (sm), result˜ (si), $s0$)
else r **endif endlet**

EVENT: Disable gseqomit.

CONSERVATIVE AXIOM: p_gseqomit_intro
let $s1$ **be** p_gseqomit $(s,\ i,\ s0)$
in
((determinate ($s1$) $\land$ (cond˜ ($s1$) $=$ `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gseqomit $(s,\ i)$, $s0$))
$\land$ determinate (gseqomit $(s,\ i)$)))
$\land$ ((determinate ($s1$) $\land$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\land$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gseqomit*.

DEFINITION:
gpf_gseqomit $(ss,\ si,\ s0)$
$=$ **let** r **be** state_check (list $(ss,\ si)$, $s0$)
in
if normal_state (r) **then** p_gseqomit (result˜ (ss), result˜ (si), $s0$)
else r **endif endlet**

EVENT: Disable gmap_insert.

CONSERVATIVE AXIOM: p_gmap_insert_intro
let $s1$ **be** p_gmap_insert $(m,\ d,\ v,\ s0)$
in
((determinate ($s1$) $\land$ (cond˜ ($s1$) $=$ `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gmap_insert $(m,\ d,\ v)$, $s0$))
$\land$ determinate (gmap_insert $(m,\ d,\ v)$)))
$\land$ ((determinate ($s1$) $\land$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\land$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gmap_insert*.

DEFINITION:
gpf_gmap_insert $(sm,\ sd,\ sv,\ s0)$
$=$ **let** r **be** state_check (list $(sm,\ sd,\ sv)$, $s0$)
in

if normal_state (r)
then p_gmap_insert (result˜ (sm), result˜ (sd), result˜ (sv), $s0$)
else r **endif endlet**

EVENT: Disable gseq_insert_before.

CONSERVATIVE AXIOM: p_gseq_insert_before_intro
let $s1$ **be** p_gseq_insert_before (s, i, v, $s0$)
in
((determinate ($s1$) $\land$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$,
store_value (`'result˜`, gseq_insert_before (s, i, v), $s0$))
$\land$ determinate (gseq_insert_before (s, i, v))))
$\land$ ((determinate ($s1$) $\land$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\land$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gseq_insert_before*.

DEFINITION:
gpf_gseq_insert_before (ss, si, sv, $s0$)
= **let** r **be** state_check (list (ss, si, sv), $s0$)
in
if normal_state (r)
then p_gseq_insert_before (result˜ (ss),
result˜ (si),
result˜ (sv),
$s0$)
else r **endif endlet**

EVENT: Disable gseq_insert_behind.

CONSERVATIVE AXIOM: p_gseq_insert_behind_intro
let $s1$ **be** p_gseq_insert_behind (s, i, v, $s0$)
in
((determinate ($s1$) $\land$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$,
store_value (`'result˜`, gseq_insert_behind (s, i, v), $s0$))
$\land$ determinate (gseq_insert_behind (s, i, v))))
$\land$ ((determinate ($s1$) $\land$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\land$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gseq_insert_behind*.

DEFINITION:
gpf_gseq_insert_behind (*ss*, *si*, *sv*, *s0*)
= **let** *r* **be** state_check (list (*ss*, *si*, *sv*), *s0*)
in
if normal_state (*r*)
then p_gseq_insert_behind (result~ (*ss*),
result~ (*si*),
result~ (*sv*),
s0)
else *r* **endif** **endlet**

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

EVENT: Disable gseq.

CONSERVATIVE AXIOM: p_gseq_intro
let *s1* **be** p_gseq (*es*, *td*, *s0*)
in
((determinate (*s1*) $\land$ (cond~ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*, store_value (`'result~`, gseq (*es*, *td*), *s0*))
$\land$ determinate (gseq (*es*, *td*))))
$\land$ ((determinate (*s1*) $\land$ (cond~ (*s1*) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained (*s1*, *s0*)
$\land$ (cond~ (*s1*)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gseq*.

DEFINITION:
gpf_gseq (*ses*, *td*, *s0*)
= **let** *r* **be** state_check (*ses*, *s0*)
in
if normal_state (*r*) **then** p_gseq (result~_list (*ses*), *td*, *s0*)
else *r* **endif** **endlet**

EVENT: Disable gset.

CONSERVATIVE AXIOM: p_gset_intro
let *s1* **be** p_gset (*es*, *td*, *s0*)

in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
 $\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gset (es, td), $s0$))
  $\wedge$ determinate (gset (es, td))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
   $\wedge$ (cond˜ ($s1$)
    $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gset*.

DEFINITION:
gpf_gset (ses, td, $s0$)
= **let** r **be** state_check (ses, $s0$)
 in
 if normal_state (r) **then** p_gset (result˜_list (ses), td, $s0$)
 else r **endif** **endlet**

EVENT: Disable grange_elements.

CONSERVATIVE AXIOM: p_grange_elements_intro
let $s1$ **be** p_grange_elements (lo, hi, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
 $\rightarrow$ (sequal ($s1$, store_value (`'result˜`, grange_elements (lo, hi), $s0$))
  $\wedge$ all_determinate (grange_elements (lo, hi))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
   $\wedge$ (cond˜ ($s1$)
    $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_grange_elements*.

DEFINITION:
range_element_state_list2 (es, s)
= **if** $es \simeq$ **nil** **then nil**
 else cons (store_result˜ (car (es), s),
  range_element_state_list2 (cdr (es), s)) **endif**

DEFINITION:
range_element_state_list (s)
= **if** normal_state (s) **then** range_element_state_list2 (result˜ (s), s)
 else list (s) **endif**

DEFINITION:
gpf_grange_elements (*slo*, *shi*, *s0*)
= **let** *r* **be** state_check (list (*slo*, *shi*), *s0*)
  in
  if normal_state (*r*)
  then range_element_state_list (p_grange_elements (result˜ (*slo*),
                                                        result˜ (*shi*),
                                                        s0))
  else *r* **endif endlet**

DEFINITION:
gpf_gset_or_seq (*m*, *ses*, *td*, *s0*)
= **if** rule (*m*, prodn (tag (`'set_or_seq_mark`, `'m`), list (`'set`, `'colon`)))
  then gpf_gset (*ses*, *td*, *s0*)
  elseif rule (*m*,
            prodn (tag (`'set_or_seq_mark`, `'m`), list (`'seq`, `'colon`)))
  then gpf_gseq (*ses*, *td*, *s0*)
  else gpf_gseq (*ses*, *td*, *s0*) **endif**

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

EVENT: Disable gminus.

CONSERVATIVE AXIOM: p_gminus_intro
let *s1* **be** p_gminus (*v*, *s0*)
in
((determinate (*s1*) $\land$ (cond˜ (*s1*) = `'normal`))
 $\rightarrow$ (sequal (*s1*, store_value (`'result˜`, gminus (*v*), *s0*))
     $\land$ determinate (gminus (*v*))))
$\land$ ((determinate (*s1*) $\land$ (cond˜ (*s1*) $\neq$ `'normal`))
   $\rightarrow$ (implementation_constrained (*s1*, *s0*)
       $\land$ (cond˜ (*s1*)
          $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gminus*.

DEFINITION:
gpf_gminus (*sv*, *s0*)
= **let** *r* **be** state_check (list (*sv*), *s0*)
  in
  if normal_state (*r*) **then** p_gminus (result˜ (*sv*), *s0*)
  else *r* **endif endlet**

EVENT: Disable gnot.

CONSERVATIVE AXIOM: p_gnot_intro
let *s1* **be** p_gnot (*v*, *s0*)
in
((determinate (*s1*) ∧ (cond˜ (*s1*) = `'normal`))
→ (sequal (*s1*, store_value (`'result˜`, gnot (*v*), *s0*))
∧ determinate (gnot (*v*))))
∧ ((determinate (*s1*) ∧ (cond˜ (*s1*) ≠ `'normal`))
→ (implementation_constrained (*s1*, *s0*)
∧ (cond˜ (*s1*)
∈ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gnot*.

DEFINITION:
gpf_gnot (*sv*, *s0*)
= **let** *r* **be** state_check (list (*sv*), *s0*)
in
if normal_state (*r*) **then** p_gnot (result˜ (*sv*), *s0*)
else *r* **endif endlet**

DEFINITION:
gpf_apply_unary_op (*op*, *sv*, *s0*)
= **if** rule (*op*, prodn (tag (`'unary_operator`, `'op`), `'minus`))
then gpf_gminus (*sv*, *s0*)
elseif rule (*op*, prodn (tag (`'unary_operator`, `'op`), `'not`))
then gpf_gnot (*sv*, *s0*)
else mark_state_indeterminate (*s0*) **endif**

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

EVENT: Disable gequal.

CONSERVATIVE AXIOM: p_gequal_intro
let *s1* **be** p_gequal (*v1*, *v2*, *s0*)
in
((determinate (*s1*) ∧ (cond˜ (*s1*) = `'normal`))
→ (sequal (*s1*, store_value (`'result˜`, gequal (*v1*, *v2*), *s0*))
∧ determinate (gequal (*v1*, *v2*))))
∧ ((determinate (*s1*) ∧ (cond˜ (*s1*) ≠ `'normal`))

$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gequal*.

DEFINITION:
gpf_gequal ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gequal (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gne.

CONSERVATIVE AXIOM: p_gne_intro
let $s1$ **be** p_gne ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gne ($v1$, $v2$), $s0$))
$\wedge$ determinate (gne ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gne*.

DEFINITION:
gpf_gne ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gne (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable glt.

CONSERVATIVE AXIOM: p_glt_intro
let $s1$ **be** p_glt ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, glt ($v1$, $v2$), $s0$))
$\wedge$ determinate (glt ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))

$\rightarrow$ (implementation_constrained (*s1*, *s0*)
$\wedge$ (cond˜ (*s1*)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_glt*.

DEFINITION:
gpf_glt (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
in
if normal_state (*r*) **then** p_glt (result˜ (*sv1*), result˜ (*sv2*), *s0*)
else *r* **endif endlet**

EVENT: Disable gor.

CONSERVATIVE AXIOM: p_gor_intro
let *s1* **be** p_gor (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*, store_value (`'result˜`, gor (*v1*, *v2*), *s0*))
$\wedge$ determinate (gor (*v1*, *v2*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond˜ (*s1*) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained (*s1*, *s0*)
$\wedge$ (cond˜ (*s1*)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gor*.

DEFINITION:
gpf_gor (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
in
if normal_state (*r*) **then** p_gor (result˜ (*sv1*), result˜ (*sv2*), *s0*)
else *r* **endif endlet**

EVENT: Disable gle.

CONSERVATIVE AXIOM: p_gle_intro
let *s1* **be** p_gle (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*, store_value (`'result˜`, gle (*v1*, *v2*), *s0*))
$\wedge$ determinate (gle (*v1*, *v2*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond˜ (*s1*) $\neq$ `'normal`))

$\rightarrow$ (implementation_constrained (*s1*, *s0*)
 $\wedge$ (cond˜ (*s1*)
 $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gle*.

DEFINITION:
gpf_gle (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
 in
 if normal_state (*r*) **then** p_gle (result˜ (*sv1*), result˜ (*sv2*), *s0*)
 else *r* **endif endlet**

EVENT: Disable ggt.

CONSERVATIVE AXIOM: p_ggt_intro
let *s1* **be** p_ggt (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
 $\rightarrow$ (sequal (*s1*, store_value (`'result˜`, ggt (*v1*, *v2*), *s0*))
 $\wedge$ determinate (ggt (*v1*, *v2*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond˜ (*s1*) $\neq$ `'normal`))
 $\rightarrow$ (implementation_constrained (*s1*, *s0*)
 $\wedge$ (cond˜ (*s1*)
 $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_ggt*.

DEFINITION:
gpf_ggt (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
 in
 if normal_state (*r*) **then** p_ggt (result˜ (*sv1*), result˜ (*sv2*), *s0*)
 else *r* **endif endlet**

EVENT: Disable gge.

CONSERVATIVE AXIOM: p_gge_intro
let *s1* **be** p_gge (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
 $\rightarrow$ (sequal (*s1*, store_value (`'result˜`, gge (*v1*, *v2*), *s0*))
 $\wedge$ determinate (gge (*v1*, *v2*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond˜ (*s1*) $\neq$ `'normal`))

$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gge*.

DEFINITION:
gpf_gge ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gge (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gand.

CONSERVATIVE AXIOM: p_gand_intro
let $s1$ **be** p_gand ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gand ($v1$, $v2$), $s0$))
$\wedge$ determinate (gand ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gand*.

DEFINITION:
gpf_gand ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gand (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gimp.

CONSERVATIVE AXIOM: p_gimp_intro
let $s1$ **be** p_gimp ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gimp ($v1$, $v2$), $s0$))
$\wedge$ determinate (gimp ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))

$\rightarrow$ (implementation_constrained (*s1*, *s0*)
 $\wedge$ (cond˜ (*s1*)
 $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gimp*.

DEFINITION:
gpf_gimp (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
in
if normal_state (*r*) **then** p_gimp (result˜ (*sv1*), result˜ (*sv2*), *s0*)
else *r* **endif endlet**

EVENT: Disable giff.

CONSERVATIVE AXIOM: p_giff_intro
let *s1* **be** p_giff (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
 $\rightarrow$ (sequal (*s1*, store_value (`'result˜`, giff (*v1*, *v2*), *s0*))
 $\wedge$ determinate (giff (*v1*, *v2*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond˜ (*s1*) $\neq$ `'normal`))
 $\rightarrow$ (implementation_constrained (*s1*, *s0*)
 $\wedge$ (cond˜ (*s1*)
 $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_giff*.

DEFINITION:
gpf_giff (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
in
if normal_state (*r*) **then** p_giff (result˜ (*sv1*), result˜ (*sv2*), *s0*)
else *r* **endif endlet**

EVENT: Disable gpower.

CONSERVATIVE AXIOM: p_gpower_intro
let *s1* **be** p_gpower (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
 $\rightarrow$ (sequal (*s1*, store_value (`'result˜`, gpower (*v1*, *v2*), *s0*))
 $\wedge$ determinate (gpower (*v1*, *v2*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond˜ (*s1*) $\neq$ `'normal`))

$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gpower*.

DEFINITION:
gpf_gpower ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gpower (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gtimes.

CONSERVATIVE AXIOM: p_gtimes_intro
let $s1$ **be** p_gtimes ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gtimes ($v1$, $v2$), $s0$))
$\wedge$ determinate (gtimes ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gtimes*.

DEFINITION:
gpf_gtimes ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gtimes (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gquotient.

CONSERVATIVE AXIOM: p_gquotient_intro
let $s1$ **be** p_gquotient ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gquotient ($v1$, $v2$), $s0$))
$\wedge$ determinate (gquotient ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))

$\rightarrow$ (implementation_constrained (*s1*, *s0*)
$\wedge$ (cond˜ (*s1*)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gquotient*.

DEFINITION:
gpf_gquotient (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
in
if normal_state (*r*)
then p_gquotient (result˜ (*sv1*), result˜ (*sv2*), *s0*)
else *r* **endif endlet**

EVENT: Disable gdiv.

CONSERVATIVE AXIOM: p_gdiv_intro
let *s1* **be** p_gdiv (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*, store_value (`'result˜`, gdiv (*v1*, *v2*), *s0*))
$\wedge$ determinate (gdiv (*v1*, *v2*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond˜ (*s1*) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained (*s1*, *s0*)
$\wedge$ (cond˜ (*s1*)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gdiv*.

DEFINITION:
gpf_gdiv (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
in
if normal_state (*r*) **then** p_gdiv (result˜ (*sv1*), result˜ (*sv2*), *s0*)
else *r* **endif endlet**

EVENT: Disable gmod.

CONSERVATIVE AXIOM: p_gmod_intro
let *s1* **be** p_gmod (*v1*, *v2*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond˜ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*, store_value (`'result˜`, gmod (*v1*, *v2*), *s0*))
$\wedge$ determinate (gmod (*v1*, *v2*))))

$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond˜ ($s1$)
      $\in$ `'(routineerror spaceerror)`)))) **endlet**

Simultaneously, we introduce the new function symbol *p_gmod*.

DEFINITION:
gpf_gmod ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
  in
  if normal_state (r) **then** p_gmod (result˜ ($sv1$), result˜ ($sv2$), $s0$)
  else r **endif endlet**

EVENT: Disable gplus.

CONSERVATIVE AXIOM: p_gplus_intro
let $s1$ **be** p_gplus ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
 $\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gplus ($v1$, $v2$), $s0$))
   $\wedge$ determinate (gplus ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond˜ ($s1$)
      $\in$ `'(routineerror spaceerror)`)))) **endlet**

Simultaneously, we introduce the new function symbol *p_gplus*.

DEFINITION:
gpf_gplus ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
  in
  if normal_state (r) **then** p_gplus (result˜ ($sv1$), result˜ ($sv2$), $s0$)
  else r **endif endlet**

EVENT: Disable gsubtract.

CONSERVATIVE AXIOM: p_gsubtract_intro
let $s1$ **be** p_gsubtract ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
 $\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gsubtract ($v1$, $v2$), $s0$))
   $\wedge$ determinate (gsubtract ($v1$, $v2$))))

∧ ((determinate (*s1*) ∧ (cond˜ (*s1*) ≠ `'normal`))
  → (implementation_constrained (*s1*, *s0*)
    ∧ (cond˜ (*s1*)
      ∈ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gsubtract*.

DEFINITION:
gpf_gsubtract (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
  in
  if normal_state (*r*)
  then p_gsubtract (result˜ (*sv1*), result˜ (*sv2*), *s0*)
  else *r* **endif endlet**

EVENT: Disable gin.

CONSERVATIVE AXIOM: p_gin_intro
let *s1* **be** p_gin (*v1*, *v2*, *s0*)
in
((determinate (*s1*) ∧ (cond˜ (*s1*) = `'normal`))
 → (sequal (*s1*, store_value (`'result˜`, gin (*v1*, *v2*), *s0*))
   ∧ determinate (gin (*v1*, *v2*))))
∧ ((determinate (*s1*) ∧ (cond˜ (*s1*) ≠ `'normal`))
  → (implementation_constrained (*s1*, *s0*)
    ∧ (cond˜ (*s1*)
      ∈ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gin*.

DEFINITION:
gpf_gin (*sv1*, *sv2*, *s0*)
= **let** *r* **be** state_check (list (*sv1*, *sv2*), *s0*)
  in
  if normal_state (*r*) **then** p_gin (result˜ (*sv1*), result˜ (*sv2*), *s0*)
  else *r* **endif endlet**

EVENT: Disable gunion.

CONSERVATIVE AXIOM: p_gunion_intro
let *s1* **be** p_gunion (*v1*, *v2*, *s0*)
in
((determinate (*s1*) ∧ (cond˜ (*s1*) = `'normal`))
 → (sequal (*s1*, store_value (`'result˜`, gunion (*v1*, *v2*), *s0*))

$\wedge$ determinate (gunion ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
→ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gunion*.

DEFINITION:
gpf_gunion ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gunion (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gadjoin.

CONSERVATIVE AXIOM: p_gadjoin_intro
let $s1$ **be** p_gadjoin ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
→ (sequal ($s1$, store_value (`'result˜`, gadjoin ($v1$, $v2$), $s0$))
$\wedge$ determinate (gadjoin ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
→ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gadjoin*.

DEFINITION:
gpf_gadjoin ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r)
then p_gadjoin (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gomit.

CONSERVATIVE AXIOM: p_gomit_intro
let $s1$ **be** p_gomit ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))

$\rightarrow$ (sequal ($s1$, store_value ('result~, gomit ($v1$, $v2$), $s0$))
$\wedge$ determinate (gomit ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ 'normal))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond~ ($s1$)
$\in$ '(routineerror spaceerror)))) **endlet**

Simultaneously, we introduce the new function symbol *p_gomit*.

DEFINITION:
gpf_gomit ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gomit (result~ ($sv1$), result~ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gsub.

CONSERVATIVE AXIOM: p_gsub_intro
let $s1$ **be** p_gsub ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = 'normal))
$\rightarrow$ (sequal ($s1$, store_value ('result~, gsub ($v1$, $v2$), $s0$))
$\wedge$ determinate (gsub ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ 'normal))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond~ ($s1$)
$\in$ '(routineerror spaceerror)))) **endlet**

Simultaneously, we introduce the new function symbol *p_gsub*.

DEFINITION:
gpf_gsub ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gsub (result~ ($sv1$), result~ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gintersect.

CONSERVATIVE AXIOM: p_gintersect_intro
let $s1$ **be** p_gintersect ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = 'normal))

$\rightarrow$ (sequal ($s1$, store_value (`'result~`, gintersect ($v1$, $v2$), $s0$))
  $\wedge$ determinate (gintersect ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond~ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gintersect*.

DEFINITION:
gpf_gintersect ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
  in
  if normal_state (r)
  then p_gintersect (result~ ($sv1$), result~ ($sv2$), $s0$)
  else r **endif endlet**

EVENT: Disable gdifference.

CONSERVATIVE AXIOM: p_gdifference_intro
let $s1$ **be** p_gdifference ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result~`, gdifference ($v1$, $v2$), $s0$))
  $\wedge$ determinate (gdifference ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond~ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gdifference*.

DEFINITION:
gpf_gdifference ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
  in
  if normal_state (r)
  then p_gdifference (result~ ($sv1$), result~ ($sv2$), $s0$)
  else r **endif endlet**

EVENT: Disable gappend.

CONSERVATIVE AXIOM: p_gappend_intro
let $s1$ **be** p_gappend ($v1$, $v2$, $s0$)

in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gappend ($v1$, $v2$), $s0$))
$\wedge$ determinate (gappend ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gappend*.

DEFINITION:
gpf_gappend ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r)
then p_gappend (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable gcons.

CONSERVATIVE AXIOM: p_gcons_intro
let $s1$ **be** p_gcons ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, gcons ($v1$, $v2$), $s0$))
$\wedge$ determinate (gcons ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_gcons*.

DEFINITION:
gpf_gcons ($sv1$, $sv2$, $s0$)
= **let** r **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_gcons (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else r **endif endlet**

EVENT: Disable grcons.

CONSERVATIVE AXIOM: p_grcons_intro

let *s1* **be** p_grcons ($v1$, $v2$, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, grcons ($v1$, $v2$), $s0$))
$\wedge$ determinate (grcons ($v1$, $v2$))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`)))) **endlet**

Simultaneously, we introduce the new function symbol *p_grcons*.

Definition:
gpf_grcons ($sv1$, $sv2$, $s0$)
= **let** *r* **be** state_check (list ($sv1$, $sv2$), $s0$)
in
if normal_state (r) **then** p_grcons (result˜ ($sv1$), result˜ ($sv2$), $s0$)
else *r* **endif** **endlet**

Definition:
gpf_apply_binary_op (op, $sv1$, $sv2$, $s0$)
= **if** rule (op, prodn (tag (`'binary_operator`, `'op`), `'eq`))
$\vee$ rule (op, prodn (tag (`'binary_operator`, `'op`), `'equal`))
then gpf_gequal ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'ne`))
then gpf_gne ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'lt`))
then gpf_glt ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'le`))
then gpf_gle ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'gt`))
then gpf_ggt ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'ge`))
then gpf_gge ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'and`))
then gpf_gand ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'or`))
then gpf_gor ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'imp`))
then gpf_gimp ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'iff`))
then gpf_giff ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'star_star`))
then gpf_gpower ($sv1$, $sv2$, $s0$)
elseif rule (op, prodn (tag (`'binary_operator`, `'op`), `'star`))

```
    then gpf_gtimes (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'slash))
    then gpf_gquotient (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'div))
    then gpf_gdiv (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'mod))
    then gpf_gmod (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'plus))
    then gpf_gplus (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'minus))
    then gpf_gsubtract (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'in))
    then gpf_gin (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'adjoin))
    then gpf_gadjoin (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'omit))
    then gpf_gomit (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'sub))
    then gpf_gsub (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'union))
    then gpf_gunion (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'intersect))
    then gpf_gintersect (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'difference))
    then gpf_gdifference (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'colon_gt))
    then gpf_gcons (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'lt_colon))
    then gpf_grcons (sv1, sv2, s0)
    elseif rule (op, prodn (tag ('binary_operator, 'op), 'append))
    then gpf_gappend (sv1, sv2, s0)
    else mark_state_indeterminate (s0) endif

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

Event: Disable std_domain.

Conservative Axiom: p_std_domain_intro

```
let s1  be  p_std_domain (d, s0)
in
((determinate (s1) ∧ (cond˜ (s1) = 'normal))
```

$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, std_domain (d), $s0$))
  $\wedge$ determinate (std_domain (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond˜ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_domain*.

DEFINITION:
gpf_std_domain (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
  in
  if normal_state (r) **then** p_std_domain (result˜_list (sd), $s0$)
  else r **endif endlet**

EVENT: Disable std_first.

CONSERVATIVE AXIOM: p_std_first_intro
let $s1$ **be** p_std_first (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result˜`, std_first (d), $s0$))
  $\wedge$ determinate (std_first (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond˜ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_first*.

DEFINITION:
gpf_std_first (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
  in
  if normal_state (r) **then** p_std_first (result˜_list (sd), $s0$)
  else r **endif endlet**

EVENT: Disable std_initial.

CONSERVATIVE AXIOM: p_std_initial_intro
let $s1$ **be** p_std_initial (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))

$\to$ (sequal ($s1$, store_value (**'result~**, std_initial (d), $s0$))
    $\wedge$ determinate (std_initial (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ **'normal**))
    $\to$ (implementation_constrained ($s1$, $s0$)
        $\wedge$ (cond~ ($s1$)
            $\in$ **'(routineerror spaceerror)**))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_initial*.

DEFINITION:
gpf_std_initial (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_initial (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_last.

CONSERVATIVE AXIOM: p_std_last_intro
let $s1$ **be** p_std_last (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = **'normal**))
$\to$ (sequal ($s1$, store_value (**'result~**, std_last (d), $s0$))
    $\wedge$ determinate (std_last (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ **'normal**))
    $\to$ (implementation_constrained ($s1$, $s0$)
        $\wedge$ (cond~ ($s1$)
            $\in$ **'(routineerror spaceerror)**))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_last*.

DEFINITION:
gpf_std_last (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_last (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_lower.

CONSERVATIVE AXIOM: p_std_lower_intro
let $s1$ **be** p_std_lower (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = **'normal**))

$\rightarrow$ (sequal ($s1$, store_value ('result~, std_lower (d), $s0$))
$\wedge$ determinate (std_lower (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ 'normal))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond~ ($s1$)
$\in$ '(routineerror spaceerror)))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_lower*.

DEFINITION:
gpf_std_lower (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_lower (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_max.

CONSERVATIVE AXIOM: p_std_max_intro
let $s1$ **be** p_std_max (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = 'normal))
$\rightarrow$ (sequal ($s1$, store_value ('result~, std_max (d), $s0$))
$\wedge$ determinate (std_max (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ 'normal))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond~ ($s1$)
$\in$ '(routineerror spaceerror)))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_max*.

DEFINITION:
gpf_std_max (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_max (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_min.

CONSERVATIVE AXIOM: p_std_min_intro
let $s1$ **be** p_std_min (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = 'normal))

$\rightarrow$ (sequal (*s1*, store_value (`'result~`, std_min (*d*), *s0*))
  $\wedge$ determinate (std_min (*d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`)))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_min*.

DEFINITION:
gpf_std_min (*sd*, *s0*)
= **let** *r* **be** state_check (*sd*, *s0*)
  in
  if normal_state (*r*) **then** p_std_min (result~_list (*sd*), *s0*)
  else *r* **endif endlet**

EVENT: Disable std_nonfirst.

CONSERVATIVE AXIOM: p_std_nonfirst_intro
let *s1* **be** p_std_nonfirst (*d*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*, store_value (`'result~`, std_nonfirst (*d*), *s0*))
  $\wedge$ determinate (std_nonfirst (*d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`)))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_nonfirst*.

DEFINITION:
gpf_std_nonfirst (*sd*, *s0*)
= **let** *r* **be** state_check (*sd*, *s0*)
  in
  if normal_state (*r*) **then** p_std_nonfirst (result~_list (*sd*), *s0*)
  else *r* **endif endlet**

EVENT: Disable std_nonlast.

CONSERVATIVE AXIOM: p_std_nonlast_intro
let *s1* **be** p_std_nonlast (*d*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = `'normal`))

$\rightarrow$ (sequal ($s1$, store_value (`'result~`, std_nonlast (d), $s0$))
  $\wedge$ determinate (std_nonlast (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond~ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_nonlast*.

DEFINITION:
gpf_std_nonlast (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_nonlast (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_null.

CONSERVATIVE AXIOM: p_std_null_intro
let $s1$ **be** p_std_null (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result~`, std_null (d), $s0$))
  $\wedge$ determinate (std_null (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond~ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_null*.

DEFINITION:
gpf_std_null (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_null (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_ord.

CONSERVATIVE AXIOM: p_std_ord_intro
let $s1$ **be** p_std_ord (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = `'normal`))

$\rightarrow$ (sequal ($s1$, store_value (`'result~`, std_ord (d), $s0$))
  $\wedge$ determinate (std_ord (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond~ ($s1$)
      $\in$ `'(routineerror spaceerror)`)))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_ord*.

DEFINITION:
gpf_std_ord (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_ord (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_pred.

CONSERVATIVE AXIOM: p_std_pred_intro
let $s1$ **be** p_std_pred (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = `'normal`))
$\rightarrow$ (sequal ($s1$, store_value (`'result~`, std_pred (d), $s0$))
  $\wedge$ determinate (std_pred (d))))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond~ ($s1$)
      $\in$ `'(routineerror spaceerror)`)))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_pred*.

DEFINITION:
gpf_std_pred (sd, $s0$)
= **let** r **be** state_check (sd, $s0$)
in
if normal_state (r) **then** p_std_pred (result~_list (sd), $s0$)
else r **endif endlet**

EVENT: Disable std_range.

CONSERVATIVE AXIOM: p_std_range_intro
let $s1$ **be** p_std_range (d, $s0$)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = `'normal`))

$\rightarrow$ (sequal (*s1*, store_value (`'result~`, std_range (*d*), *s0*))
  $\wedge$ determinate (std_range (*d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_range*.

DEFINITION:
gpf_std_range (*sd*, *s0*)
= **let** *r* **be** state_check (*sd*, *s0*)
  in
  if normal_state (*r*) **then** p_std_range (result~_list (*sd*), *s0*)
  else *r* **endif endlet**

EVENT: Disable std_scale.

CONSERVATIVE AXIOM: p_std_scale_intro
let *s1* **be** p_std_scale (*d*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*, store_value (`'result~`, std_scale (*d*), *s0*))
  $\wedge$ determinate (std_scale (*d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_scale*.

DEFINITION:
gpf_std_scale (*sd*, *s0*)
= **let** *r* **be** state_check (*sd*, *s0*)
  in
  if normal_state (*r*) **then** p_std_scale (result~_list (*sd*), *s0*)
  else *r* **endif endlet**

EVENT: Disable std_size.

CONSERVATIVE AXIOM: p_std_size_intro
let *s1* **be** p_std_size (*d*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = `'normal`))

$\rightarrow$ (sequal (*s1*, store_value ('`result~`, std_size (*d*), *s0*))
  $\wedge$ determinate (std_size (*d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ '`normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ '`(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_size*.

DEFINITION:
gpf_std_size (*sd*, *s0*)
= **let** *r* **be** state_check (*sd*, *s0*)
  in
  if normal_state (*r*) **then** p_std_size (result~_list (*sd*), *s0*)
  else *r* **endif endlet**

EVENT: Disable std_succ.

CONSERVATIVE AXIOM: p_std_succ_intro
let *s1* **be** p_std_succ (*d*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = '`normal`))
$\rightarrow$ (sequal (*s1*, store_value ('`result~`, std_succ (*d*), *s0*))
  $\wedge$ determinate (std_succ (*d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ '`normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ '`(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_succ*.

DEFINITION:
gpf_std_succ (*sd*, *s0*)
= **let** *r* **be** state_check (*sd*, *s0*)
  in
  if normal_state (*r*) **then** p_std_succ (result~_list (*sd*), *s0*)
  else *r* **endif endlet**

EVENT: Disable std_upper.

CONSERVATIVE AXIOM: p_std_upper_intro
let *s1* **be** p_std_upper (*d*, *s0*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = '`normal`))

$\rightarrow$ (sequal (*s1*, store_value (`'result~`, std_upper (*d*), *s0*))
  $\wedge$ determinate (std_upper (*d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_std_upper*.

DEFINITION:
gpf_std_upper (*sd*, *s0*)
= **let** *r* **be** state_check (*sd*, *s0*)
  in
  if normal_state (*r*) **then** p_std_upper (result~_list (*sd*), *s0*)
  else *r* **endif endlet**

```
; ***********
;  Variables
; ***********
```

CONSERVATIVE AXIOM: p_apply_var_intro
let *s1* **be** p_apply_var (*fn*, *s0*, *d*)
in
((determinate (*s1*) $\wedge$ (cond~ (*s1*) = `'normal`))
$\rightarrow$ (sequal (*s1*,
      store_value (`'result~`, apply_var (*fn*, map (*s0*), *d*), *s0*))
  $\wedge$ determinate (apply_var (*fn*, map (*s0*), *d*))))
$\wedge$ ((determinate (*s1*) $\wedge$ (cond~ (*s1*) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained (*s1*, *s0*)
    $\wedge$ (cond~ (*s1*)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_apply_var*.

DEFINITION:
gpf_apply_var (*fn*, *s*, *d*)
= **if** normal_state (*s*)
  then if state_componentp (*fn*, *s*)
    then let *r* **be** state_check (*d*, *s*)
      in
      if normal_state (*r*)
      then p_apply_var (*fn*, *s*, result~_list (*d*))
      else *r* **endif endlet**
    else set_condition (*s*, `'routineerror`) **endif**
  else *s* **endif**

```
; **************
;  Bound Values
; **************
```

Event: Disable bound_values.

Conservative Axiom: gpf_bound_values_intro

let *s1* **be** gpf_bound_values$(e,\ c,\ s0,\ x)$
in
((determinate$(s1)$ $\wedge$ (cond~$(s1)$ = `'normal`))
 $\rightarrow$ (sequal$(s1$, store_value(`'result~`, bound_values$(e,\ c,\ x)$, $s0)$)
   $\wedge$ all_determinate(bound_values$(e,\ c,\ x)$)))
$\wedge$ ((determinate$(s1)$ $\wedge$ (cond~$(s1)$ $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained$(s1,\ s0)$
   $\wedge$ (cond~$(s1)$
    $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *gpf_bound_values*.

```
; **********************************
;  Space Allocation and Deallocation
; **********************************
```

Conservative Axiom: allocate_intro

let *s1* **be** allocate$(k,\ v,\ s0)$
in
((determinate$(s1)$ $\wedge$ (cond~$(s1)$ = `'normal`))
 $\rightarrow$ sequal$(s1$, store_value$(k,\ v,\ s0)$))
$\wedge$ ((determinate$(s1)$ $\wedge$ (cond~$(s1)$ $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained$(s1,\ s0)$
   $\wedge$ (cond~$(s1)$
    $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *allocate*.

Conservative Axiom: allocate_const_intro

let *s1* **be** allocate_const$(k,\ v,\ sc,\ s0)$
in
((determinate$(s1)$ $\wedge$ (cond~$(s1)$ = `'normal`))
 $\rightarrow$ sequal$(s1$, store_const$(k,\ v,\ sc,\ s0)$))
$\wedge$ ((determinate$(s1)$ $\wedge$ (cond~$(s1)$ $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained$(s1,\ s0)$
   $\wedge$ (cond~$(s1)$
    $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *allocate_const*.

DEFINITION:
gp_deallocate_locals (s)
= deallocate_vars (local_vars (s),
    deallocate_consts (local_consts (s),
        deallocate_conds (local_conds (s), s)))

```
; ******************
;  Name Expressions
; ******************

; A name expression is a marked object, with
;     mark = 'name_expression
;     object = (id . <evaluated selector_list>)
```

DEFINITION:
name_exp (id, ss) = marked (`'name_expression`, cons (id, ss))

DEFINITION: namep (v) = (mark (v) = `'name_expression`)

DEFINITION:
ne_name (v)
= **if** namep (v) **then** car (object (v))
  else f endif

DEFINITION:
ne_selectors (v)
= **if** namep (v) **then** cdr (object (v))
  else nil endif

```
; ******************
;  Retyping Result~
; ******************

; This is used in constant interpretation.
```

DEFINITION:
retype_result~ (s, td)
= **if** determinate (result~ (s)) $\land$ truep (in_type (td, result~ (s)))
  then store_value (`'result~`,
      marked (**nil**, typed (td, value (result~ (s)))),
      s)
  else set_condition (s, `'routineerror`) **endif**

CONSERVATIVE AXIOM: p_retype_result˜_intro

let *s1* **be** p_retype_result˜ (*s0*, *td*)
in
((determinate (*s1*) ∧ (cond˜ (*s1*) = `'normal`))
 → sequal (*s1*, retype_result˜ (*s0*, *td*)))
∧ ((determinate (*s1*) ∧ (cond˜ (*s1*) ≠ `'normal`))
  → (implementation_constrained (*s1*, *s0*)
   ∧ (cond˜ (*s1*)
    ∈ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_retype_result˜*.

DEFINITION:
gpf_retype_result˜ (*s*, *td*)
= **if** normal_state (*s*) **then** p_retype_result˜ (*s*, *td*)
  else *s* **endif**

```
; ******************************
;  Functions on Type Descriptors
; ******************************
```

DEFINITION:
subtype_irange (*t1*, *t2*)
= **if** bounded_typep (*t1*)
  then if bounded_typep (*t2*)
   then if integerp (tmin (*t1*))
     ∧ integerp (tmax (*t1*))
     ∧ integerp (tmin (*t2*))
     ∧ integerp (tmax (*t2*))
    then ileq (tmin (*t2*), tmin (*t1*)) ∧ ileq (tmax (*t1*), tmax (*t2*))
    else f endif
   else t endif
  elseif bounded_typep (*t2*) **then f**
  else t endif

DEFINITION:
subtype_rrange (*t1*, *t2*)
= **if** bounded_typep (*t1*)
  then if bounded_typep (*t2*)
   then if rationalp (tmin (*t1*))
     ∧ rationalp (tmax (*t1*))
     ∧ rationalp (tmin (*t2*))
     ∧ rationalp (tmax (*t2*))
    then rleq (tmin (*t2*), tmin (*t1*)) ∧ rleq (tmax (*t1*), tmax (*t2*))

else f endif
else t endif
elseif bounded_typep ($t2$) **then f**
else t endif

DEFINITION:
subtype_size ($s1$, $s2$)
= **if** $s1$ = **nil** **then** $s2$ = **nil**
elseif $s2$ = **nil** **then** $s1 \in \mathbf{N}$
elseif $(s1 \in \mathbf{N}) \wedge (s2 \in \mathbf{N})$ **then** $s1 \leq s2$
else f endif

```
#|
(do-mutual '(

(defn subtype_fields (t1 t2)
  (if (nlistp t1)
      T
    (and (subtype (cdar t1) (mapped_value t2 (caar t1)))
 (subtype_fields (cdr t1) t2)))
  ( (lessp (tree_size t1)) ))

(defn subtype (t1 t2)
  (if (and (type_descp t1) (type_descp t2)
   (equal (mode t1) (mode t2)))
      (case (mode t1)
(integer (subtype_irange t1 t2))
(rational (subtype_rrange t1 t2))
(scalar (and (equal (tid t1) (tid t2))
     (equal (sid t1) (sid t2))
     (type_vequal (crd t1) (crd t2) (integer_desc))
     (subtype_irange t1 t2)))
(array (and (type_equal (selector_td t1) (selector_td t2))
    (subtype (component_td t1) (component_td t2))))
(record (and (set_equal (field_names t1) (field_names t2))
     (subtype_fields (field_tds t1) (field_tds t2))))
(mapping (and (subtype_size (max_size t1) (max_size t2))
      (subtype (selector_td t1) (selector_td t2))
      (subtype (component_td t1) (component_td t2))))
(sequence (and (subtype_size (max_size t1) (max_size t2))
       (subtype (component_td t1) (component_td t2))))
(set (and (subtype_size (max_size t1) (max_size t2))
  (subtype (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-subtype-subtype_fields ($mutual\text{-}flg$, $t1$, $t2$)
= **if** $mutual\text{-}flg$ = `'subtype`
then if type_descp ($t1$)
∧ type_descp ($t2$)
∧ (mode ($t1$) = mode ($t2$))
then case on mode ($t1$):
case = *integer*
then subtype_irange ($t1$, $t2$)
case = *rational*
then subtype_rrange ($t1$, $t2$)
case = *scalar*
then (tid ($t1$) = tid ($t2$))
∧ (sid ($t1$) = sid ($t2$))
∧ type_vequal (crd ($t1$), crd ($t2$), INTEGER_DESC)
∧ subtype_irange ($t1$, $t2$)
case = *array*
then type_equal (selector_td ($t1$), selector_td ($t2$))
∧ mutual-subtype-subtype_fields (`'subtype`, component_td ($t1$), component_td ($t2$))
case = *record*
then set_equal (field_names ($t1$), field_names ($t2$))
∧ mutual-subtype-subtype_fields (`'subtype_fields`, field_tds ($t1$), field_tds ($t2$))
case = *mapping*
then subtype_size (max_size ($t1$), max_size ($t2$))
∧ mutual-subtype-subtype_fields (`'subtype`, selector_td ($t1$), selector_td ($t2$))
∧ mutual-subtype-subtype_fields (`'subtype`, component_td ($t1$), component_td ($t2$))
case = *sequence*

then subtype_size (max_size ($t1$), max_size ($t2$))
$\wedge$ mutual-subtype-subtype_fields (`'subtype`,
component_td ($t1$),
component_td ($t2$))
case = *set*
then subtype_size (max_size ($t1$), max_size ($t2$))
$\wedge$ mutual-subtype-subtype_fields (`'subtype`,
component_td ($t1$),
component_td ($t2$))
case = *pending*
then (tid ($t1$) = tid ($t2$)) $\wedge$ (sid ($t1$) = sid ($t2$))
otherwise f endcase
else f endif
elseif $t1 \simeq$ **nil then t**
else mutual-subtype-subtype_fields (`'subtype`,
cdar ($t1$),
mapped_value ($t2$, caar ($t1$)))
$\wedge$ mutual-subtype-subtype_fields (`'subtype_fields`,
cdr ($t1$),
$t2$) **endif**

DEFINITION:
subtype ($t1$, $t2$) = mutual-subtype-subtype_fields (`'subtype`, $t1$, $t2$)

DEFINITION:
subtype_fields ($t1$, $t2$)
= mutual-subtype-subtype_fields (`'subtype_fields`, $t1$, $t2$)

DEFINITION:
type_check (td, s)
= **if** determinate (result˜ (s)) $\wedge$ truep (in_type (td, result˜ (s))) **then** s
else set_condition (s, `'routineerror`) **endif**

CONSERVATIVE AXIOM: p_type_check_intro
let $s1$ **be** p_type_check (td, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
$\rightarrow$ sequal ($s1$, type_check (td, $s0$)))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond˜ ($s1$)
$\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_type_check*.

Definition:
gpf_type_check (td, s)
= **if** normal_state (s) **then** p_type_check (td, s)
else s **endif**

```
; ********************
;  Type Name Arguments
; ********************
```

Definition:
type_name_arg (tn, sn, s, x)
= store_value (**'result~**,
marked (**'type_descriptor**,
type_desc (mk_identifier (tn), sn, **nil**, x)),
s)

Conservative Axiom: p_type_name_arg_intro
let $s1$ **be** p_type_name_arg (tn, sn, $s0$, x)
in
((determinate ($s1$) $\wedge$ (cond~ ($s1$) = **'normal**))
$\rightarrow$ sequal ($s1$, type_name_arg (tn, sn, $s0$, x)))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond~ ($s1$) $\neq$ **'normal**))
$\rightarrow$ (implementation_constrained ($s1$, $s0$)
$\wedge$ (cond~ ($s1$)
$\in$ **'(routineerror spaceerror)**))) **endlet**

Simultaneously, we introduce the new function symbol *p_type_name_arg*.

Definition:
gpf_type_name_arg (tn, sn, s, x)
= **if** normal_state (s) **then** p_type_name_arg (tn, sn, s, x)
else s **endif**

```
; *********************
;  Specification Values
; *********************
```

Definition:
set_entry (e, c, s, n, x)
= **let** v **be** gf (e, c, map (s), n, x)
in
if boolean_typep (type (v)) **then** store_value (**'entry**, v, s)
else store_value (**'entry**,

```
                    marked (entry_not_boolean_error (e, c),
                            default_value (BOOLEAN_DESC)),
                    s) endif endlet
```

CONSERVATIVE AXIOM: p_set_entry_intro

```
let s1  be  p_set_entry (e, c, s0, n, x)
in
((determinate (s1) ∧ (cond˜ (s1) = 'normal))
 →   sequal (s1, set_entry (e, c, s0, n, x)))
∧   ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
      →   (implementation_constrained (s1, s0)
            ∧   (cond˜ (s1)
                  ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_set_entry*.

DEFINITION:

```
gp_set_entry (e, c, s, n, x)
=   if normal_state (s)  then  p_set_entry (e, c, s, n, x)
    else s endif
```

DEFINITION:

```
exit_labels_ok (cs, s)
=   (setp (cs) ∧ all_conditionsp (cs, store_cond ('normal, 'formal, s)))
```

DEFINITION:

```
set_exit (e, c, s, n, x)
=   if exit_labels_ok (exit_labels (e), s)
    then let v  be  gf (postc (e, cond˜ (s)), c, map (s), n, x)
         in
         if boolean_typep (type (v))  then  store_value ('exit, v, s)
         else store_value ('exit,
                           marked (exit_not_boolean_error (e, c),
                                   default_value (BOOLEAN_DESC)),
                           s) endif endlet
    else store_value ('exit,
                      marked (exit_label_error (e, c),
                              default_value (BOOLEAN_DESC)),
                      s) endif
```

CONSERVATIVE AXIOM: p_set_exit_intro

```
let s1  be  p_set_exit (e, c, s0, n, x)
in
((determinate (s1) ∧ (cond˜ (s1) = 'normal))
 →   sequal (s1, set_exit (e, c, s0, n, x)))
```

$\land$ ((determinate (*s1*) $\land$ (cond˜ (*s1*) $\neq$ `'normal`))
 $\rightarrow$ (implementation_constrained (*s1*, *s0*)
 $\land$ (cond˜ (*s1*)
 $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_set_exit*.

DEFINITION:
gp_set_exit (e, c, s, n, x)
= **if** normal_state (s) **then** p_set_exit (e, c, s, n, x)
 else s **endif**

DEFINITION:
update_keep (s, c, n, x)
= store_value (`'keep`, gand (keep (s), gf (keep˜ (s), c, map (s), n, x)), s)

CONSERVATIVE AXIOM: p_update_keep_intro
let *s1* **be** p_update_keep (*s0*, c, n, x)
in
((determinate (*s1*) $\land$ (cond˜ (*s1*) = `'normal`))
 $\rightarrow$ sequal (*s1*, update_keep (*s0*, c, n, x)))
$\land$ ((determinate (*s1*) $\land$ (cond˜ (*s1*) $\neq$ `'normal`))
 $\rightarrow$ (implementation_constrained (*s1*, *s0*)
 $\land$ (cond˜ (*s1*)
 $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_update_keep*.

DEFINITION:
gp_update_keep (s, c, n, x)
= **if** normal_state (s) **then** p_update_keep (s, c, n, x)
 else s **endif**

DEFINITION:
gp_set_keep (k, s, c, n, x)
= **if** normal_state (s) **then** gp_update_keep (allocate (`'keep˜`, k, s), c, n, x)
 else s **endif**

DEFINITION:
update_assert (e, c, s, n, x)
= store_value (`'assert`, gand (assert (s), gf (e, c, map (s), n, x)), s)

CONSERVATIVE AXIOM: p_update_assert_intro
let *s1* **be** p_update_assert (e, c, *s0*, n, x)
in
((determinate (*s1*) $\land$ (cond˜ (*s1*) = `'normal`))

$\rightarrow$ sequal ($s1$, update_assert (e, c, $s0$, n, x)))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond˜ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_update_assert*.

DEFINITION:
gp_update_assert (e, c, s, n, x)
= **if** normal_state (s) **then** p_update_assert (e, c, s, n, x)
  else s **endif**

DEFINITION:
record_assert (v, s) = store_value (`'assert`, gand (assert (s), v), s)

CONSERVATIVE AXIOM: p_record_assert_intro
let $s1$ **be** p_record_assert (v, $s0$)
in
((determinate ($s1$) $\wedge$ (cond˜ ($s1$) = `'normal`))
 $\rightarrow$ sequal ($s1$, record_assert (v, $s0$)))
$\wedge$ ((determinate ($s1$) $\wedge$ (cond˜ ($s1$) $\neq$ `'normal`))
  $\rightarrow$ (implementation_constrained ($s1$, $s0$)
    $\wedge$ (cond˜ ($s1$)
      $\in$ `'(routineerror spaceerror)`))) **endlet**

Simultaneously, we introduce the new function symbol *p_record_assert*.

DEFINITION:
gp_record_assert (v, s)
= **if** normal_state (s) **then** p_record_assert (v, s)
  else s **endif**

```
; ************
;  Assignment
; ************
```

DEFINITION:
mapping_selectionp (ss, td)
= **if** $ss \simeq$ **nil then f**
  else case on mode (td):
    case = *array*
    then mapping_selectionp (cdr (ss), component_td (td))
    case = *record*

```
        then mapping_selectionp (cdr (ss),
                                 field_td (value (car (ss)), td))
       case = mapping
        then t
       case = sequence
        then mapping_selectionp (cdr (ss), component_td (td))
       otherwise f endcase endif
```

DEFINITION:

```
mapping_element_lhsp (n, s)
=   if namep (n)
    then mapping_selectionp (ne_selectors (n), type_of (ne_name (n), s))
    else f endif
```

DEFINITION:

```
gassign0 (ne, v, s)
=   if namep (ne)
        ∧  variablep (ne_name (ne), s)
        ∧  (¬ mapping_element_lhsp (ne, s))
    then let id  be  ne_name (ne),
             ss  be  ne_selectors (ne)
         in
         if state_componentp (id, s)
         then let td  be  type_of (id, s),
                  v2  be  put_op (state_component (id, s), ss, v)
              in
              if determinate (v2) ∧ truep (in_type (td, v2))
              then store_value (id,
                                marked (nil,
                                        typed (td, value (v2))),
                                s)
              else set_condition (s, 'routineerror) endif endlet
         else set_condition (s, 'routineerror) endif endlet
    else set_condition (s, 'routineerror) endif
```

DEFINITION:

```
gassign (ne, v, s, c, n, x) = gp_update_keep (gassign0 (ne, v, s), c, n, x)
```

CONSERVATIVE AXIOM: p_assign_intro

```
 let s1  be  p_assign (ne, v, s0, c, n, x)
 in
 ((determinate (s1) ∧ (cond˜ (s1) = 'normal))
  →   sequal (s1, gassign (ne, v, s0, c, n, x)))
∧   ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
      →   (implementation_constrained (s1, s0)
```

```
            ∧   (cond~ (s1)
                  ∈  '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_assign*.

DEFINITION:

```
gp_assign (sne, sv, s0, c, n, x)
=   let r  be  state_check (list (sne, sv), s0)
    in
    if normal_state (r)
    then p_assign (result~ (sne), result~ (sv), s0, c, n, x)
    else r endif endlet
```

```
; ***************
;  New Statement
; ***************
```

DEFINITION:

```
gnew0 (dc, v, ne, s)
=   let id  be  ne_name (ne),
        ss  be  ne_selectors (ne)
    in
    if rule (dc,
             prodn (tag ('new_dynamic_variable_component,
                         'dc),
                    list ('into, tag ('name_expression, 'ne))))
    then gassign0 (name_exp (id, rcdr (ss)),
                   gmap_insert (apply_var (id, map (s), rcdr (ss)),
                                rcar (ss),
                                v),
                   s)
    elseif rule (dc,
                 prodn (tag ('new_dynamic_variable_component,
                             'dc),
                        list ('into,
                              'set,
                              tag ('name_expression, 'ne))))
    then gassign0 (ne, gadjoin (apply_var (id, map (s), ss), v), s)
    elseif rule (dc,
                 prodn (tag ('new_dynamic_variable_component,
                             'dc),
                        list ('before,
                              tag ('name_expression, 'ne))))
    then gassign0 (name_exp (id, rcdr (ss)),
```

```
                    gseq_insert_before (apply_var (id, map (s), rcdr (ss)),
                                        rcar (ss),
                                        v),
                    s)
    elseif rule (dc,
                 prodn (tag ('new_dynamic_variable_component,
                             'dc),
                        list ('before,
                              'seq,
                              tag ('name_expression, 'ne))))
    then gassign0 (ne, gcons (v, apply_var (id, map (s), ss)), s)
    elseif rule (dc,
                 prodn (tag ('new_dynamic_variable_component,
                             'dc),
                        list ('behind,
                              tag ('name_expression, 'ne))))
    then gassign0 (name_exp (id, rcdr (ss)),
                   gseq_insert_behind (apply_var (id, map (s), rcdr (ss)),
                                       rcar (ss),
                                       v),
                   s)
    elseif rule (dc,
                 prodn (tag ('new_dynamic_variable_component,
                             'dc),
                        list ('behind,
                              'seq,
                              tag ('name_expression, 'ne))))
    then gassign0 (ne, grcons (apply_var (id, map (s), ss), v), s)
    else mark_state_indeterminate (s) endif endlet
```

DEFINITION:
gnew (dc, v, ne, c, s, n, x) = gp_update_keep (gnew0 (dc, v, ne, s), $c, n, x)$

CONSERVATIVE AXIOM: p_new_intro

```
let s1 be p_new (dc, v, ne, c, s0, n, x)
in
((determinate (s1) ∧ (cond˜ (s1) = 'normal))
 →  sequal (s1, gnew (dc, v, ne, c, s0, n, x)))
∧   ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
     →  (implementation_constrained (s1, s0)
         ∧  (cond˜ (s1)
             ∈  '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_new*.

DEFINITION:
gp_new (*dc*, *sv*, *sne*, *c*, *s0*, *n*, *x*)
```
=   let r  be  state_check (list (sv, sne), s0)
    in
    if normal_state (r)
    then p_new (dc, result~ (sv), result~ (sne), c, s0, n, x)
    else r endif endlet
```

```
; *****************
;  Remove Statement
; *****************
```

DEFINITION:
gremove0 (*v*, *ne*, *s*)
```
=   let id  be  ne_name (ne),
        ss  be  ne_selectors (ne)
    in
    if v = nil
    then if mapping_descp (type (apply_var (id, map (s), rcdr (ss))))
         then gassign0 (name_exp (id, rcdr (ss)),
                        gmapomit (apply_var (id, map (s), rcdr (ss)),
                                  rcar (ss)),
                        s)
         else gassign0 (name_exp (id, rcdr (ss)),
                        gseqomit (apply_var (id, map (s), rcdr (ss)),
                                  rcar (ss)),
                        s) endif
    else gassign0 (ne, gomit (apply_var (id, map (s), ss), v), s) endif endlet
```

DEFINITION:
gremove (*v*, *ne*, *c*, *s*, *n*, *x*) = gp_update_keep (gremove0 (*v*, *ne*, *s*), *c*, *n*, *x*)

CONSERVATIVE AXIOM: p_remove_intro
```
let s1  be  p_remove (v, ne, c, s0, n, x)
in
((determinate (s1) ∧ (cond~ (s1) = 'normal))
 →   sequal (s1, gremove (v, ne, c, s0, n, x)))
∧   ((determinate (s1) ∧ (cond~ (s1) ≠ 'normal))
     →   (implementation_constrained (s1, s0)
          ∧   (cond~ (s1)
               ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_remove*.

```
DEFINITION:
gp_remove (sv, sne, c, s0, n, x)
=   let r  be  if sv = nil  then state_check (list (sne), s0)
               else state_check (list (sv, sne), s0) endif
    in
    if normal_state (r)
    then p_remove (if sv = nil  then nil
                   else result˜ (sv) endif,
                   result˜ (sne),
                   c,
                   s0,
                   n,
                   x)
    else r endif endlet

; ****************
;  Move Statement
; ****************


DEFINITION:
stored_value (n, s) = apply_var (ne_name (n), map (s), ne_selectors (n))

DEFINITION:
gmove_assign (cd, v, ne, s)
=   if rule (cd,
            prodn (tag ('component_destination, 'd),
                   tag ('new_dynamic_variable_component, 'dc)))
    then gnew0 (subtree (cd, 'new_dynamic_variable_component),
                v,
                ne,
                s)
    elseif rule (cd,
                 prodn (tag ('component_destination, 'd),
                        list ('to, tag ('name_expression, 'ne))))
    then if sequence_descp (type (stored_value (name_exp (ne_name (ne),
                                                          rcdr (ne_selectors (ne))),
                                                s))) then gassign0 (ne, v, s)
         else set_condition (s, 'routineerror) endif
    else mark_state_indeterminate (s) endif

DEFINITION:
same_selectors (s1, s2)
=   if s1 ≃ nil  then s2 ≃ nil
    elseif s2 ≃ nil  then f
```

```
    else ((car (s1) = car (s2)) ∨ gtruep (gequal (car (s1), car (s2))))
           ∧   same_selectors (cdr (s1), cdr (s2)) endif
```

DEFINITION:

```
same_names (n1, n2)
=   (namep (n1)
      ∧   namep (n2)
      ∧   (ne_name (n1) = ne_name (n2))
      ∧   same_selectors (ne_selectors (n1), ne_selectors (n2)))
```

DEFINITION:

```
remove_dynamic_name (rv, rne)
=   if rv = nil then name_exp (ne_name (rne), rcdr (ne_selectors (rne)))
    else rne endif
```

DEFINITION:

```
assign_dynamic_name (cd, nne)
=   if rule (cd,
             prodn (tag ('component_destination, 'd),
                    tag ('new_dynamic_variable_component, 'dc)))
    then assign_dynamic_name (subtree (cd,
                                       'new_dynamic_variable_component),
                              nne)
    elseif rule (cd,
                 prodn (tag ('new_dynamic_variable_component, 'dc),
                        list ('into, tag ('name_expression, 'ne))))
           ∨   rule (cd,
                     prodn (tag ('new_dynamic_variable_component,
                                 'dc),
                            list ('before,
                                  tag ('name_expression, 'ne))))
           ∨   rule (cd,
                     prodn (tag ('new_dynamic_variable_component,
                                 'dc),
                            list ('behind,
                                  tag ('name_expression, 'ne))))
           ∨   rule (cd,
                     prodn (tag ('component_destination, 'd),
                            list ('to, tag ('name_expression, 'ne))))
    then name_exp (ne_name (nne), rcdr (ne_selectors (nne)))
    else nne endif
```

DEFINITION:

```
gmove (rv, rne, cd, nne, c, s, n, x)
=   if same_names (remove_dynamic_name (rv, rne), assign_dynamic_name (cd, nne))
```

```
    then set_condition (s, 'routineerror)
    else let r1  be  gmove_assign (cd,
                                   if rv = nil
                                   then stored_value (rne, s)
                                   else rv endif,
                                   nne,
                                   s)
         in
         if normal_state (r1)
         then let r2  be  gremove0 (rv, rne, s)
                in
                if normal_state (r2)
                then gp_update_keep (r2, c, n, x)
                else marked (mark (r2),
                              map (set_condition (s,
                                                  cond~ (r2)))) endif endlet
         else r1 endif endlet endif
```

CONSERVATIVE AXIOM: p_move_intro

```
let s1  be  p_move (rv, rne, cd, nne, c, s0, n, x)
in
((determinate (s1) ∧ (cond~ (s1) = 'normal))
 →  sequal (s1, gmove (rv, rne, cd, nne, c, s0, n, x)))
∧  ((determinate (s1) ∧ (cond~ (s1) ≠ 'normal))
     →  (implementation_constrained (s1, s0)
          ∧  (cond~ (s1)
               ∈  '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_move*.

DEFINITION:

```
gp_move (srv, srne, cd, snne, c, s, n, x)
=   let r  be  if srv = nil  then state_check (list (srne, snne), s)
               else state_check (list (srv, srne, snne), s) endif
    in
    if normal_state (r)
    then p_move (if srv = nil  then nil
                 else result~ (srv) endif,
                 result~ (srne),
                 cd,
                 result~ (snne),
                 c,
                 s,
                 n,
                 x)
```

else r **endif endlet**

```
; ****************
;  Procedure Calls
; ****************

; =====================================
;  Argument Checking for Procedure Calls
; =====================================

; ------------
;  Formals OK
; ------------
```

DEFINITION:
pformals_ok (fs)
$=$ **if** $fs \simeq$ **nil then t**
elseif reserved_idp (car (fs)) **then f**
elseif car (fs) $\in$ cdr (fs) **then f**
else pformals_ok (cdr (fs)) **endif**

```
; ----------------
;  Aliasing Checks
; ----------------
```

DEFINITION:
selectors_aliasedp ($s1$, $s2$)
$=$ **if** ($s1 \simeq$ **nil**) $\vee$ ($s2 \simeq$ **nil**) **then t**
else ((car ($s1$) $=$ car ($s2$)) $\vee$ gtruep (gequal (car ($s1$), car ($s2$))))
$\wedge$ selectors_aliasedp (cdr ($s1$), cdr ($s2$)) **endif**

DEFINITION:
harmful_aliasp ($a1$, $a2$, $f1$, $f2$)
$=$ **if** (access ($f1$) $=$ `'var`) $\vee$ (access ($f2$) $=$ `'var`)
then if namep ($a1$) $\wedge$ namep ($a2$)
then (ne_name ($a1$) $=$ ne_name ($a2$))
$\wedge$ selectors_aliasedp (ne_selectors ($a1$), ne_selectors ($a2$))
else f endif
else f endif

DEFINITION:
harmfully_aliasedp (ap, as, fp, fs)
$=$ **if** $as \simeq$ **nil then f**
else harmful_aliasp (ap, car (as), fp, car (fs))
$\vee$ harmfully_aliasedp (ap, cdr (as), fp, cdr (fs)) **endif**

DEFINITION:

```
no_harmful_aliasing (as, fs)
=  if as ≃ nil then t
   elseif namep (car (as))
   then (¬ harmfully_aliasedp (car (as), cdr (as), car (fs), cdr (fs)))
        ∧  no_harmful_aliasing (cdr (as), cdr (fs))
   else no_harmful_aliasing (cdr (as), cdr (fs)) endif
```

```
; ------------------------
;  Type and Access Checks
; ------------------------
```

DEFINITION:

```
one_parg_check (fp, ap, fsn, s, x)
=  let ft  be type_desc (formal_type (fp), fsn, nil, x),
       av  be if namep (ap)
              then apply_var (ne_name (ap), map (s), ne_selectors (ap))
              else ap endif
   in
   if indeterminate (av)  then set_condition (s, 'routineerror)
   elseif access (fp) = 'var
   then if namep (ap)
           ∧  variablep (ne_name (ap), s)
           ∧  (¬ mapping_element_lhsp (ap, s))
        then if subtype (ft, type (av)) ∧ truep (in_type (ft, av))
             then s
             else set_condition (s, 'routineerror) endif
        else set_condition (s, 'routineerror) endif
   elseif truep (in_type (ft, av))  then s
   else set_condition (s, 'routineerror) endif endlet
```

EVENT: Disable set_condition.

EVENT: Disable one_parg_check.

EVENT: Disable normal_state.

DEFINITION:

```
parg_check2 (fs, as, fsn, s, x)
=  if as ≃ nil
   then if fs ≃ nil then s
        else set_condition (s, 'routineerror) endif
```

```
    elseif fs ≃ nil then set_condition (s, 'routineerror)
    else let s2 be one_parg_check (car (fs), car (as), fsn, s, x)
         in
         if normal_state (s2)
         then parg_check2 (cdr (fs), cdr (as), fsn, s, x)
         else s2 endif endlet endif

; ---------------------------------------------
;  The Full Argument Check for Procedure Calls
; ---------------------------------------------
```

DEFINITION:

```
cond_arg_check (fcs, acs, s)
=   if length (fcs) = length (acs)
    then if all_conditionsp (acs, s) then s
         else mark_state_indeterminate (s) endif
    else set_condition (s, 'routineerror) endif
```

DEFINITION:

```
parg_check (u, usn, ads, acs, s, x)
=   let fds be formal_dargs (u),
        fcs be formal_cargs (u)
    in
    case on kind (u):
    case = function
    then if pformals_ok (append (cons ('result,
                                       dparam_name_list (fds)),
                                 append (fcs,
                                         '(routineerror
                                           spaceerror))))
            ∧  (farg_check (fds, ads, usn, x) = nil)
         then cond_arg_check (fcs, acs, s)
         else set_condition (s, 'routineerror) endif
    case = procedure
     then if pformals_ok (append (dparam_name_list (fds),
                                  append (fcs,
                                          '(routineerror
                                            spaceerror))))
           then let s2 be parg_check2 (fds, ads, usn, s, x)
                in
                if normal_state (s2)
                then if no_harmful_aliasing (ads, fds)
                     then cond_arg_check (fcs, acs, s)
                     else set_condition (s,
```

```
                                        'routineerror) endif
                    else s2 endif endlet
                else set_condition (s, 'routineerror) endif
    otherwise set_condition (s, 'routineerror) endcase endlet
```

```
; ==============================
;  New State for Procedure Call
; ==============================
```

DEFINITION:

```
padd_darg (fp, ap, fsn, sa, s, x)
=   let ft  be  type_desc (formal_type (fp), fsn, nil, x),
        av  be  if namep (ap)
                then apply_var (ne_name (ap), map (sa), ne_selectors (ap))
                else ap endif
    in
    let fv  be  marked (nil, typed (ft, value (av)))
    in
    if access (fp) = 'var
    then store_const (mk_entry_name (dparam_name (fp)),
                      fv,
                      'formal,
                      store_var (dparam_name (fp),
                                 fv,
                                 'formal,
                                 s))
    else store_const (dparam_name (fp), fv, 'formal, s) endif endlet endlet
```

DEFINITION:

```
pbind_dargs (fs, as, fsn, s, x)
=   if as ≃ nil  then DEFAULT_STATE
    else padd_darg (rcar (fs),
                    rcar (as),
                    fsn,
                    s,
                    pbind_dargs (rcdr (fs), rcdr (as), fsn, s, x),
                    x) endif
```

DEFINITION:

```
padd_result (s, ftype)
=   let v  be  std_initial (list (marked ('type_descriptor, ftype)))
    in
    if determinate (v)
    then store_const (mk_entry_name ('result),
```

```
                     v,
                     'formal,
                     store_var ('result, v, 'formal, s))
    else set_condition (s, 'routineerror) endif endlet
```

DEFINITION:

```
call_state (u, usn, ads, acs, s, x)
=   let r1  be parg_check (u, usn, ads, acs, s, x)
    in
    if normal_state (r1)
    then let r2  be note_conds (formal_cargs (u),
                                'formal,
                                pbind_dargs (formal_dargs (u),
                                             ads,
                                             usn,
                                             s,
                                             x))
         in
         if kind (u) = 'function
         then let ftype  be type_desc (result_type (u),
                                       usn,
                                       nil,
                                       x)
              in
              padd_result (r2, ftype) endlet
         else r2 endif endlet
    else r1 endif endlet
```

EVENT: Disable call_state.

EVENT: Disable determinate.

EVENT: Enable sequal.

CONSERVATIVE AXIOM: p_call_state_intro

```
 (determinate (p_call_state (u, usn, ads, acs, s0, x))
  ∧   (¬ sequal (p_call_state (u, usn, ads, acs, s0, x),
                 call_state (u, usn, ads, acs, s0, x))))
 →   (cond~ (p_call_state (u, usn, ads, acs, s0, x))
       ∈  '(routineerror spaceerror))
```

Simultaneously, we introduce the new function symbol *p_call_state*.
EVENT: Enable determinate.

EVENT: Disable sequal.

DEFINITION:
gp_call_state (u, usn, $sads$, acs, s, x)
= **let** r **be** state_check ($sads$, s)
in
if normal_state (r)
then p_call_state (u, usn, result~_list ($sads$), acs, s, x)
else r **endif endlet**

```
; ==============================
;  State Update for Call Effects
; ==============================
```

DEFINITION:
map_cond_effects (fc, fcs, acs, s)
= **if** fc = `'normal` **then** s
elseif $fc \in$ `'(routineerror spaceerror)`
then set_condition (s, fc)
elseif listp (fcs) $\wedge$ listp (acs)
then if fc = car (fcs) **then** set_condition (s, car (acs))
else map_cond_effects (fc, cdr (fcs), cdr (acs), s) **endif**
else mark_state_indeterminate (s) **endif**

DEFINITION:
map_var_effects (fs, as, $s2$, $s1$)
= **if** $fs \simeq$ **nil then** $s1$
elseif access (car (fs)) = `'var`
then map_var_effects (cdr (fs),
cdr (as),
$s2$,
gassign0 (car (as),
state_component (dparam_name (car (fs)), $s2$),
$s1$))
else map_var_effects (cdr (fs), cdr (as), $s2$, $s1$) **endif**

DEFINITION:
map_call_effects ($s2$, u, ads, acs, c, $s1$, n, x)
= **if** indeterminate ($s2$) **then** mark_state_indeterminate ($s1$)
elseif kind (u) = `'function`
then if condition_normal ($s2$)
then allocate (`'result~`, state_component (`'result`, $s2$), $s1$)
else map_cond_effects (cond~ ($s2$), formal_cargs (u), acs, $s1$) **endif**

```
     else gp_update_keep (map_cond_effects (cond~ (s2),
                                            formal_cargs (u),
                                            acs,
                                            map_var_effects (formal_dargs (u),
                                                             ads,
                                                             s2,
                                                             s1)),
                          c,
                          n,
                          x) endif
```

EVENT: Enable sequal.

CONSERVATIVE AXIOM: gp_map_call_effects_intro

```
 sequal (gp_map_call_effects (s2, u, ads, acs, c, s1, n, x),
         map_call_effects (s2, u, ads, acs, c, s1, n, x))
```

Simultaneously, we introduce the new function symbol *gp_map_call_effects*.
EVENT: Disable sequal.

```
; ============
;  Local Names
; ============
```

DEFINITION:

```
gp_new_namep (id, s)
=   ((¬ in_map (map (s), id))
     ∧  (¬ in_map (cond+ (s), id))
     ∧  (¬ reserved_idp (id)))
```

```
; ================
;  Local Variables
; ================
```

DEFINITION:

```
bind_local (a, id, td, iv, s)
=   if gp_new_namep (id, s)
    then let v be if iv = nil
                  then std_initial (list (marked ('type_descriptor,
                                                  td)))
                  else iv endif
         in
```

```
        if determinate (v) ∧ truep (in_type (td, v))
        then if a = 'var
             then store_var (id,
                             marked (nil, typed (td, value (v))),
                             'local,
                             s)
             else store_const (id,
                               marked (nil, typed (td, value (v))),
                               'local,
                               s) endif
        else set_condition (s, 'routineerror) endif endlet
   else set_condition (s, 'routineerror) endif
```

CONSERVATIVE AXIOM: p_bind_local_intro

```
let s1 be p_bind_local (a, id, td, iv, s0)
in
((determinate (s1) ∧ (cond~ (s1) = 'normal))
 →   sequal (s1, bind_local (a, id, td, iv, s0)))
∧   ((determinate (s1) ∧ (cond~ (s1) ≠ 'normal))
     →   (implementation_constrained (s1, s0)
          ∧   (cond~ (s1)
               ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_bind_local*.

DEFINITION:

```
gp_bind_local (a, id, td, iv, s)
=    let r be if iv = nil then s
              else state_check (list (iv), s) endif
     in
     if normal_state (r)
     then p_bind_local (a,
                        id,
                        td,
                        if iv = nil then iv
                        else result~ (iv) endif,
                        s)
     else r endif endlet
```

DEFINITION:

```
gp_bind_locals (a, ids, td, iv, s)
=    if ids ≃ nil then s
     else gp_bind_locals (a,
                          cdr (ids),
                          td,
```

```
                         iv,
                         gp_bind_local (a, car (ids), td, iv, s)) endif
```

DEFINITION:

```
gp_local_conds (ids, s)
=  if (ids ≃ nil) ∨ (¬ normal_state (s))  then s
   elseif gp_new_namep (car (ids), s)
   then gp_local_conds (cdr (ids), store_cond (car (ids), 'local, s))
   else set_condition (s, 'routineerror) endif
```

```
; ****************
;  Case Statement
; ****************
```

DEFINITION:

```
case_label_check (k, cs, s)
=  let td  be  base_type (type (k))
   in
   if non_rational_simple_typep (td)
      ∧   (arg_check (cs, ncopies (length (cs), td)) = nil)
      ∧   vsetp (values (cs), td)  then s
   else set_condition (s, 'routineerror) endif endlet
```

CONSERVATIVE AXIOM: p_case_label_check_intro

```
 let s1  be  p_case_label_check (k, cs, s0)
 in
 ((determinate (s1) ∧ (cond˜ (s1) = 'normal))
  →    sequal (s1, case_label_check (k, cs, s0)))
 ∧    ((determinate (s1) ∧ (cond˜ (s1) ≠ 'normal))
        →    (implementation_constrained (s1, s0)
               ∧    (cond˜ (s1)
                      ∈   '(routineerror spaceerror)))) endlet
```

Simultaneously, we introduce the new function symbol *p_case_label_check*.

DEFINITION:

```
gp_case_label_check (sk, scs, s0)
=  let r  be  state_check (cons (sk, scs), s0)
   in
   if normal_state (r)
   then p_case_label_check (result˜ (sk), result˜_list (scs), s0)
   else r endif endlet
```

DEFINITION:

```
condition_labels_ok (cs, s) = (setp (cs) ∧ all_conditionsp (cs, s))
```

```
; *********************
;  The Meta-Function GP
; *********************
```

EVENT: Disable *1*boolean_desc.

EVENT: Disable gpf_gand.

EVENT: Disable gpf_gchar.

EVENT: Disable gpf_gin.

EVENT: Disable gpf_gmap_insert.

EVENT: Disable gpf_gmapomit.

EVENT: Disable gpf_gor.

EVENT: Disable gpf_grange_elements.

EVENT: Disable gpf_gseq_insert_before.

EVENT: Disable gpf_gseq_insert_behind.

EVENT: Disable gpf_gseqomit.

EVENT: Disable gpf_gset.

EVENT: Disable gpf_gset_or_seq.

EVENT: Disable gpf_gstring_seq.

EVENT: Disable gpf_apply_binary_op.

EVENT: Disable gpf_apply_unary_op.

EVENT: Disable gpf_apply_var.

EVENT: Disable gpf_bound_values.

EVENT: Disable gpf_false.

EVENT: Disable gpf_minteger.

EVENT: Disable gpf_put_op.

EVENT: Disable gpf_record_get.

EVENT: Disable gpf_retype_result.

EVENT: Disable gpf_select_op.

EVENT: Disable gpf_std_domain.

EVENT: Disable gpf_std_first.

EVENT: Disable gpf_std_initial.

EVENT: Disable gpf_std_last.

EVENT: Disable gpf_std_lower.

EVENT: Disable gpf_std_max.

EVENT: Disable gpf_std_min.

EVENT: Disable gpf_std_nonfirst.

EVENT: Disable gpf_std_nonlast.

EVENT: Disable gpf_std_null.

EVENT: Disable gpf_std_ord.

EVENT: Disable gpf_std_pred.

EVENT: Disable gpf_std_range.

EVENT: Disable gpf_std_scale.

EVENT: Disable gpf_std_size.

EVENT: Disable gpf_std_succ.

EVENT: Disable gpf_std_upper.

EVENT: Disable gpf_subsequence_get.

EVENT: Disable gpf_true.

EVENT: Disable gpf_type_check.

EVENT: Disable gpf_type_name_arg.

EVENT: Disable gp_assign.

EVENT: Disable gp_bind_locals.

EVENT: Disable gp_call_state.

EVENT: Disable gp_case_label_check.

EVENT: Disable gp_deallocate_locals.

EVENT: Disable gp_local_conds.

EVENT: Disable gp_map_call_effects.

EVENT: Disable gp_move.

EVENT: Disable gp_new.

EVENT: Disable gp_new_namep.

EVENT: Disable gp_record_assert.

EVENT: Disable gp_remove.

EVENT: Disable gp_set_entry.

EVENT: Disable gp_set_exit.

EVENT: Disable gp_set_keep.

EVENT: Disable gp_update_assert.

EVENT: Disable gtruep.

EVENT: Disable access.

EVENT: Disable actual_cargs.

EVENT: Disable actual_dargs.

EVENT: Disable allocate.

EVENT: Disable allocate_const.

EVENT: Disable arg_list.

EVENT: Disable base_type.

EVENT: Disable boolean_desc.

EVENT: Disable bound_id.

EVENT: Disable case_labels.

EVENT: Disable cdr_quantified_exp.

EVENT: Disable character_valuep.

EVENT: Disable condition_labels_ok.

EVENT: Disable condition_non_normal.

EVENT: Disable condition_normal.

EVENT: Disable conditionp.

EVENT: Disable cond~.

EVENT: Disable constant_body.

EVENT: Disable digit_listp.

EVENT: Disable each_clausep.

EVENT: Disable entry_name.

EVENT: Disable entry_valuep.

EVENT: Disable exit_spec.

EVENT: Disable expression_from_spec.

EVENT: Disable fn_call_formp.

EVENT: Disable formal_cargs.

EVENT: Disable formal_dargs.

EVENT: Disable gname.

EVENT: Disable handler.

EVENT: Disable handler_labels.

EVENT: Disable id_list.

EVENT: Disable identifierp.

EVENT: Disable if_else_exp.

EVENT: Disable if_statement_else_part.

EVENT: Disable indeterminate.

EVENT: Disable internal_initial_value_exp.

EVENT: Disable keep_spec.

EVENT: Disable kind.

EVENT: Disable length.

EVENT: Disable map.

EVENT: Disable map_cond_effects.

EVENT: Disable mark_state_indeterminate.

EVENT: Disable mk_name_expression.

EVENT: Disable mk_signal_stmt.

EVENT: Disable mk_unary_operator.

EVENT: Disable name_exp.

EVENT: Disable new_name_arg.

EVENT: Disable normal_state.

EVENT: Disable object_name.

EVENT: Disable prec.

EVENT: Disable procedure_body.

EVENT: Disable rcar.

EVENT: Disable rcdr.

EVENT: Disable ref.

EVENT: Disable remove_exp_arg.

EVENT: Disable remove_name_arg.

EVENT: Disable reset_leave_to_normal.

EVENT: Disable result_type.

EVENT: Disable result~.

EVENT: Disable result~_list.

EVENT: Disable set_condition.

EVENT: Disable state_componentp.

EVENT: Disable string_valuep.

EVENT: Disable type.

EVENT: Disable type_desc.

EVENT: Disable type_name_expp.

```
#|
(do-mutual '(

; ****************************************************************************
;
;                       THE EXPRESSION INTERPRETER
;
; ****************************************************************************

; ***************************
;  Set/Sequence Constructors
; ***************************

;; This returns a list of states rather than a single state.

(defn GPF_element_list (e c s n x)

  (if (not (normal_state s))
      (list s)

  (if (zerop (fix n))
      (list (mark_state_indeterminate s))

  (if (rule e (prodn (tag 'range 'r)
     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GPF_element_list (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'value_list 'v)))
      (GPF_element_list (subtree e 'value_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'range_limits 'r)))
      (GPF_element_list (subtree e 'range_limits) c s (sub1 n) x)
```

```
  (if (rule e (prodn (tag 'range_limits 'r)
     (list (tag 'expression 'lo) 'DOT_DOT
   (tag 'expression 'hi))))
      (GPF_Grange_elements (GPF (subtree_i e 'expression 1) c s (sub1 n) x)
   (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
   s)

  (if (rule e (prodn (tag 'value_list 'v)
     (tag 'expression 'e)))
      (rcons nil (GPF (subtree e 'expression) c s (sub1 n) x))

  (if (rule e (prodn (tag 'value_list 'v)
     (list (tag 'value_list 'v2) 'COMMA
   (tag 'expression 'e))))
      (rcons (GPF_element_list (subtree e 'value_list) c s (sub1 n) x)
     (GPF (subtree e 'expression) c s (sub1 n) x))

    (list (mark_state_indeterminate s))))))))))

  ( (lessp (count n)) ))


; Unlike most of the other functions here, this doesn't store the value in
; the state.

(defn GPF_element_type (e c s n x)

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule e (prodn (tag 'range 'r)
     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GPF_element_type (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'value_list 'v)))
      (GPF_element_type (subtree e 'value_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
     (tag 'range_limits 'r)))
      (GPF_element_type (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'range_limits 'r)
```

```
     (list (tag 'expression 'lo) 'DOT_DOT
   (tag 'expression 'hi))))
      (base_type (type (result~ (GPF (subtree_i e 'expression 1)
     c s (sub1 n) x))))

  (if (rule e (prodn (tag 'value_list 'v)
     (tag 'expression 'e)))
      (base_type (type (result~ (GPF (subtree e 'expression) c s (sub1 n) x))))

  (if (rule e (prodn (tag 'value_list 'v)
     (list (tag 'value_list 'v2) 'COMMA
   (tag 'expression 'e))))
      (GPF_element_type (subtree e 'value_list) c s (sub1 n) x)

    nil)))))))

  ( (lessp (count n)) ))



; ************************
;  Quantified expressions
; ************************

(defn GPF_all (id vs e c s n x)
  (if (nlistp vs)
      (GPF_true s)
  (if (GP_new_namep id s)
      (if (zerop n)
  (mark_state_indeterminate s)
        (GPF_Gand (GPF_all id (rcdr vs) e c s (sub1 n) x)
  (GPF e c (allocate_const id (rcar vs) 'local s) (sub1 n) x)
  s))
    (set_condition s 'routineerror)))
  ( (lessp (count n)) ))


(defn GPF_some (id vs e c s n x)
  (if (nlistp vs)
      (GPF_false s)
  (if (GP_new_namep id s)
      (if (zerop n)
  (mark_state_indeterminate s)
(GPF_Gor (GPF_some id (rcdr vs) e c s (sub1 n) x)
```

```
 (GPF e c (allocate_const id (rcar vs) 'local s) (sub1 n) x)
 s))
    (set_condition s 'routineerror)))
  ( (lessp (count n)) ))


; *********************
;  Value Modifications
; *********************

(defn GPF_each (id vs sBV e c s n x)
  ; e is the <component modification>
  ; sBV is the state whose result~ component is the value being modified
  (if (not (normal_state s))
      s
  (if (nlistp vs)
      sBV
  (if (GP_new_namep id s)
      (if (zerop n)
  (mark_state_indeterminate s)
(GPF_each id (cdr vs)
  (GPF_modifiers sBV e c (allocate_const id (car vs) 'local s)
 (sub1 n) x)
  e c s (sub1 n) x))
    (set_condition s 'routineerror))))
  ( (lessp (count n)) ))


(defn GPF_adp (e c s n x)

  (if (not (normal_state s))
      (list s)

  (if (zerop (fix n))
      (list (mark_state_indeterminate s))

  (if (rule e (prodn (tag 'arg_list 'as)
     (list 'OPEN_PAREN (tag 'value_list 'vs)
   'CLOSE_PAREN)))
      (GPF_adp (subtree e 'value_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'value_list 'vs)
     (tag 'expression 'e)))
      (rcons nil (GPF (subtree e 'expression) c s (sub1 n) x))
```

```
  (if (rule e (prodn (tag 'value_list 'vs)
     (list (tag 'value_list 'vs2)
   'COMMA (tag 'expression 'e))))
      (rcons (GPF_adp (subtree e 'value_list) c s (sub1 n) x)
     (GPF (subtree e 'expression) c s (sub1 n) x))

    (list (mark_state_indeterminate s)))))))

  ( (lessp (count n)) ))

(defn GPF_selectors (e c s n x)

  (if (not (normal_state s))
      (list s)

  (if (zerop (fix n))
      (list (mark_state_indeterminate s))

  (if (rule e (prodn (tag 'selector_list 's)
     (tag 'component_selectors 's2)))
      (GPF_selectors (subtree e 'component_selectors) c s (sub1 n) x)

  (if (rule e (prodn (tag 'selector_list 's)
     (list (tag 'selector_list 's2)
   (tag 'component_selectors 's3))))
      (append (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x)
      (GPF_selectors (subtree e 'component_selectors) c s (sub1 n) x))

  (if (rule e (prodn (tag 'component_selectors 's)
     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (list (allocate 'result~
      (marked 'field_name (gname (subtree e 'IDENTIFIER)))
      s))

  (if (rule e (prodn (tag 'component_selectors 's)
     (tag 'arg_list 'd)))
      (GPF_adp (subtree e 'arg_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'arg_list 'as)
     (list 'OPEN_PAREN (tag 'value_list 'vs)
   'CLOSE_PAREN)))
      (GPF_adp (subtree e 'value_list) c s (sub1 n) x)
```

```
    (list (mark_state_indeterminate s)))))))))

  ( (lessp (count n)) ))

(defn GPF_modifiers (sBV e c s n x)
  ; e is the <value modifiers>
  ; sBV is the state whose result~ component is being modified

  (if (not (normal_state s))
      s

  (if (not (normal_state sBV))
      sBV

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule e (prodn (tag 'value_modifiers 'm)
     (tag 'component_selectors 's)))
      (GPF_modifiers sBV (subtree e 'component_selectors) c s (sub1 n) x)

  (if (rule e (prodn (tag 'component_selectors 's)
     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (GPF_record_get sBV
      (allocate 'result~
(marked 'field_name
(gname (subtree e 'IDENTIFIER)))
s)
      s)

  (if (rule e (prodn (tag 'component_selectors 's)
     (tag 'arg_list 'd)))
      (GPF_select_op sBV (GPF_adp (subtree e 'arg_list) c s (sub1 n) x) s)

  (if (rule e (prodn (tag 'value_modifiers 'm)
     (tag 'range 'r)))
      (GPF_modifiers sBV (subtree e 'range) c s (sub1 n) x)

  (if (rule e (prodn (tag 'range 'r)
     (list 'OPEN_PAREN (tag 'range_limits 'r2)
   'CLOSE_PAREN)))
      (GPF_modifiers sBV (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'range_limits 'r)
```

```
    (list (tag 'expression 'lo) 'DOT_DOT
  (tag 'expression 'hi))))
     (GPF_subsequence_get sBV
  (GPF (subtree_i e 'expression 1) c s (sub1 n) x)
  (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
  s)

 (if (rule e (prodn (tag 'value_modifiers 'm)
    (tag 'value_alterations 'a)))
     (GPF_modifiers sBV (subtree e 'value_alterations) c s (sub1 n) x)

 (if (rule e (prodn (tag 'value_alterations 'a)
    (list 'WITH 'OPEN_PAREN
  (tag 'component_alterations_list 'al)
  'CLOSE_PAREN)))
     (GPF_modifiers sBV (subtree e 'component_alterations_list)
    c s (sub1 n) x)

 (if (rule e (prodn (tag 'component_alterations_list 'al)
    (tag 'component_alterations 'a)))
     (GPF_modifiers sBV (subtree e 'component_alterations) c s (sub1 n) x)

 (if (rule e (prodn (tag 'component_alterations_list 'al)
    (list (tag 'component_alterations_list 'al2)
  'SEMI_COLON (tag 'component_alterations 'a))))
     (GPF_modifiers (GPF_modifiers sBV
   (subtree e 'component_alterations_list)
   c s (sub1 n) x)
    (subtree e 'component_alterations) c s (sub1 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 ((sVS (GPF_bound_values (subtree e 'opt_each_clause) c s x)))
   (if (normal_state sVS)
(GPF_each (bound_id (subtree e 'opt_each_clause))
 (result~ sVS) sBV (subtree e 'component_assignment)
 c s (sub1 n) x)
     sVS))
(GPF_modifiers sBV (subtree e 'component_assignment) c s (sub1 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 ((sVS (GPF_bound_values (subtree e 'opt_each_clause) c s x)))
    (if (normal_state sVS)
(GPF_each (bound_id (subtree e 'opt_each_clause))
  (result~ sVS) sBV (subtree e 'component_creation)
  c s (sub1 n) x)
      sVS))
(GPF_modifiers sBV (subtree e 'component_creation) c s (sub1 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 ((sVS (GPF_bound_values (subtree e 'opt_each_clause) c s x)))
    (if (normal_state sVS)
(GPF_each (bound_id (subtree e 'opt_each_clause))
  (result~ sVS) sBV (subtree e 'component_deletion)
  c s (sub1 n) x)
      sVS))
(GPF_modifiers sBV (subtree e 'component_deletion) c s (sub1 n) x))

  (if (rule e (prodn (tag 'component_assignment 'a)
     (list (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (GPF_put_op sBV
  (GPF_selectors (subtree e 'selector_list)
 c s (sub1 n) x)
  (GPF (subtree e 'expression) c s (sub1 n) x)
  s)

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'BEFORE (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x))
    (sU (GPF (subtree e 'expression) c s (sub1 n) x)))
(GPF_put_op sBV (rcdr sS)
    (GPF_Gseq_insert_before (GPF_select_op sBV (rcdr sS) s)
    (rcar sS) sU s)
    s))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'BEHIND (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
```

```
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x))
    (sU (GPF (subtree e 'expression) c s (sub1 n) x)))
(GPF_put_op sBV (rcdr sS)
    (GPF_Gseq_insert_behind (GPF_select_op sBV (rcdr sS) s)
    (rcar sS) sU s)
    s))

  (if (rule e (prodn (tag 'component_creation 'c)
     (list 'INTO (tag 'selector_list 's)
   'COLON_EQUAL (tag 'expression 'e))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x))
    (sU (GPF (subtree e 'expression) c s (sub1 n) x)))
(GPF_put_op sBV (rcdr sS)
    (GPF_Gmap_insert (GPF_select_op sBV (rcdr sS) s)
     (rcar sS) sU s)
    s))

  (if (rule e (prodn (tag 'component_deletion 'd)
     (list 'SEQOMIT (tag 'selector_list 's))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x)))
(GPF_put_op sBV (rcdr sS)
    (GPF_Gseqomit (GPF_select_op sBV (rcdr sS) s)
  (rcar sS) s)
    s))

  (if (rule e (prodn (tag 'component_deletion 'd)
     (list 'MAPOMIT (tag 'selector_list 's))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x)))
(GPF_put_op sBV (rcdr sS)
    (GPF_Gmapomit (GPF_select_op sBV (rcdr sS) s)
  (rcar sS) s)
    s))

    (mark_state_indeterminate s)))))))))))))))))))))))

  ( (lessp (count n)) ))



; ***********************************
;  Name references and function calls
; ***********************************

(defn GPF_apply_fun (fn adp acp sn s n x)
```

```
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (let ((h (car (ref fn sn x)))  ; scope fn is declared in
  (u (cdr (ref fn sn x)))) ; the function declaration
      (if (equal (kind u) 'function)
  (let ((fs (formal_dargs u)))                    ; formals
  (let ((ds (if (equal (length fs) 0) nil adp))   ; actuals
(ss (if (equal (length fs) 0) adp nil)))  ; selectors
    (GPF_select_op (GP_procedure_call fn ds acp sn s (sub1 n) x)
   ss s)))
      (if (equal (kind u) 'constant)
  (if (equal acp nil)
      (GPF_select_op
         (GPF_retype_result~ (GPF (constant_body u) h s (sub1 n) x)
     (type_desc (result_type u) h nil x))
 adp s)
    (set_condition s 'routineerror))
(set_condition s 'routineerror)))))
  ( (lessp (count n)) ))


(defn GPF_apply (fn adp acp sn s n x)
  ; adp is a list of states
  (if (state_componentp fn s)
      (if (equal acp nil)
  (GPF_apply_var fn s adp)
(set_condition s 'routineerror))
  (if (equal fn 'false)
      (if (equal acp nil)
  (GPF_select_op (GPF_false s) adp s)
(set_condition s 'routineerror))
  (if (equal fn 'true)
      (if (equal acp nil)
  (GPF_select_op (GPF_true s) adp s)
(set_condition s 'routineerror))
  (if (type_name_expp fn adp sn x)
      (if (equal acp nil)
  (GPF_type_name_arg fn sn s x)
(set_condition s 'routineerror))
  (if (equal fn 'domain)
      (if (equal acp nil)
  (GPF_std_domain adp s)
(set_condition s 'routineerror))
  (if (equal fn 'first)
```

```
    (if (equal acp nil)
  (GPF_std_first adp s)
(set_condition s 'routineerror))
  (if (equal fn 'initial)
    (if (equal acp nil)
  (GPF_std_initial adp s)
(set_condition s 'routineerror))
  (if (equal fn 'last)
    (if (equal acp nil)
  (GPF_std_last adp s)
(set_condition s 'routineerror))
  (if (equal fn 'lower)
    (if (equal acp nil)
  (GPF_std_lower adp s)
(set_condition s 'routineerror))
  (if (equal fn 'max)
    (if (equal acp nil)
  (GPF_std_max adp s)
(set_condition s 'routineerror))
  (if (equal fn 'min)
    (if (equal acp nil)
  (GPF_std_min adp s)
(set_condition s 'routineerror))
  (if (equal fn 'nonfirst)
    (if (equal acp nil)
  (GPF_std_nonfirst adp s)
(set_condition s 'routineerror))
  (if (equal fn 'nonlast)
    (if (equal acp nil)
  (GPF_std_nonlast adp s)
(set_condition s 'routineerror))
  (if (equal fn 'null)
    (if (equal acp nil)
  (GPF_std_null adp s)
(set_condition s 'routineerror))
  (if (equal fn 'ord)
    (if (equal acp nil)
  (GPF_std_ord adp s)
(set_condition s 'routineerror))
  (if (equal fn 'pred)
    (if (equal acp nil)
  (GPF_std_pred adp s)
(set_condition s 'routineerror))
  (if (equal fn 'range)
```

```
      (if (equal acp nil)
  (GPF_std_range adp s)
(set_condition s 'routineerror))
  (if (equal fn 'scale)
      (if (equal acp nil)
  (GPF_std_scale adp s)
(set_condition s 'routineerror))
  (if (equal fn 'size)
      (if (equal acp nil)
  (GPF_std_size adp s)
(set_condition s 'routineerror))
  (if (equal fn 'succ)
      (if (equal acp nil)
  (GPF_std_succ adp s)
(set_condition s 'routineerror))
  (if (equal fn 'upper)
      (if (equal acp nil)
  (GPF_std_upper adp s)
(set_condition s 'routineerror))
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (GPF_apply_fun fn adp acp sn s (sub1 n) x)))))))))))))))))))))))
  ( (lessp (count n)) ))


(defn GPF_list (es c s n x)
  (if (nlistp es)
      nil
  (if (zerop (fix n))
      (list (mark_state_indeterminate s))
    (cons (GPF (car es) c s (sub1 n) x)
  (GPF_list (cdr es) c s (sub1 n) x))))
  ( (lessp (count n)) ))

(defn GPF (e c s n x)
  ; The meta-function GPF(e,c,s,n,x) gives the state that results when the
  ; expression e is interpreted for at most n steps, in the context of scope
  ; c, initial state s, and Gypsy sentence x.
  ;
  ; The domain and range of GPF(e,c,s,n,x) are as follows:
  ;
  ; e is the parse tree of an expression which describes a mechanism.
  ; c is the (litatom) name of the Gypsy scope in which e appears.
  ; s is  the marked, initial state on which the expression mechanism
```

```
  ;   e begins to operate.
  ; n is the maximum number of steps that the expression mechanism is
  ;   allowed to operate.
  ; x is  the  parse  tree  of the program description sentence which
  ;   defines the library which  contains  the  declarations  of  the
  ;   Gypsy units which are referred to by e.
  ; gPF(e,c,s,n,x) is the marked, final state which results from operating the
  ;                mechanism on the initial state s for at most n steps.  If
  ;                the mechanism has not completed all of its operations in n
  ;                steps, then the final state is marked as indeterminate.

  (if (not (normal_state s))
      s

  (if (zerop (fix n))
      (mark_state_indeterminate s)


  ; ************************
  ;
  ;    <expression> ::= ...
  ;
  ; ************************

  (if (rule e (prodn (tag 'expression 'e)
     (tag 'modified_primary_value 'm)))
      (GPF (subtree e 'modified_primary_value) c s (sub1 n) x)

  (if (rule e (prodn (tag 'expression 'e)
     (list 'ALL (tag 'bound_expression 'b))))
      (let ((sVS (GPF_bound_values e c s x)))
(if (normal_state sVS)
    (GPF_all (bound_id (subtree e 'bound_expression))
     (result~ sVS) (cdr_quantified_exp e)
     c s (sub1 n) x)
  sVS))

  (if (rule e (prodn (tag 'expression 'e)
     (list 'SOME (tag 'bound_expression 'b))))
      (let ((sVS (GPF_bound_values e c s x)))
(if (normal_state sVS)
    (GPF_some (bound_id (subtree e 'bound_expression))
      (result~ sVS) (cdr_quantified_exp e)
      c s (sub1 n) x)
```

```
  sVS))

  (if (rule e (prodn (tag 'expression 'e)
     (list (tag 'unary_operator 'op) (tag 'expression 'e2))))
      (GPF_apply_unary_op (subtree e 'unary_operator)
  (GPF (subtree e 'expression) c s (sub1 n) x)
  s)

  (if (rule e (prodn (tag 'expression 'e)
     (list (tag 'expression 'e1) (tag 'binary_operator 'op)
   (tag 'expression 'e2))))
      (GPF_apply_binary_op (subtree e 'binary_operator)
   (GPF (subtree_i e 'expression 1) c s (sub1 n) x)
   (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
   s)


  ; ********************************
  ;
  ;    <modified primary value> ::=
  ;
  ; ********************************

  (if (rule e (prodn (tag 'modified_primary_value 'm)
     (tag 'primary_value 'p)))
      (GPF (subtree e 'primary_value) c s (sub1 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)
  (GPF_apply (object_name (subtree e 'modified_primary_value))
     (GPF_adp (arg_list (subtree e 'value_modifiers))
      c s (sub1 n) x)
     nil c s (sub1 n) x)
(GPF_modifiers (GPF (subtree e 'modified_primary_value)
    c s (sub1 n) x)
       (subtree e 'value_modifiers) c s (sub1 n) x))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
     (list (tag 'modified_primary_value 'm2)
   (tag 'actual_condition_parameters 'cp))))
      (if (fn_call_formp e)
  (GPF_apply (object_name (subtree e 'modified_primary_value))
```

```
    (GPF_adp (arg_list (subtree e 'modified_primary_value))
     c s (sub1 n) x)
    (actual_cargs e)
    c s (sub1 n) x)
(set_condition s 'routineerror))


  ; **********************
  ;
  ;    <primary value> ::=
  ;
  ; **********************

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'literal_value 'l)))
      (GPF (subtree e 'literal_value) c s (sub1 n) x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'set_or_sequence_value 's)))
      (GPF (subtree e 'set_or_sequence_value) c s (sub1 n) x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'ENTRY_VALUE 'e)))
      (GPF (subtree e 'ENTRY_VALUE) c s (sub1 n) x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'IDENTIFIER 'on)))
      (GPF (subtree e 'IDENTIFIER) c s (sub1 n) x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (tag 'if_expression 'i)))
      (GPF (subtree e 'if_expression) c s (sub1 n) x)

  (if (rule e (prodn (tag 'primary_value 'p)
     (list 'OPEN_PAREN (tag 'expression 'e) 'CLOSE_PAREN)))
      (GPF (subtree e 'expression) c s (sub1 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> ::=
  ;
  ; ***********************

  (if (rule e (prodn (tag 'constant_body 'b)
     (tag 'expression 'e)))
      (GPF (subtree e 'expression) c s (sub1 n) x)

  (if (rule e (prodn (tag 'constant_body 'b)
     'PENDING))
      (mark_state_indeterminate s)


  ; ***********************
  ;
  ;    <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 (GPF_type_check (boolean_desc)
(GPF (subtree_i e 'expression 1)
     c s (sub1 n) x))))
(if (not (normal_state bv))
    bv
  (if (Gtruep (result~ bv))
      (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
    (GPF (if_else_exp (subtree e 'if_expression_else_part))
 c s (sub1 n) x))))


  ; ***********************
  ;
```

```
;    <literal value> ::=
;
; ***********************

(if (rule e (prodn (tag 'literal_value 'l)
   (tag 'CHARACTER_VALUE 'ch)))
    (GPF (subtree e 'CHARACTER_VALUE) c s (sub1 n) x)

(if (rule e (prodn (tag 'literal_value 'l)
   (tag 'number 'n)))
    (GPF (subtree e 'number) c s (sub1 n) x)

(if (rule e (prodn (tag 'literal_value 'l)
   (tag 'STRING_VALUE 's)))
    (GPF (subtree e 'STRING_VALUE) c s (sub1 n) x)


; ****************
;
;    <number> ::=
;
; ****************

(if (rule e (prodn (tag 'number 'n)
   (tag 'DIGIT_LIST 's)))
    (GPF_minteger e s)

(if (rule e (prodn (tag 'number 'n)
   (list (tag 'base 'b) (tag 'DIGIT_LIST 's))))
    (GPF_minteger e s)


; ****************************************
;
;    <pre-computable label expression> ::=
;
; ****************************************

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (tag 'number 'n)))
    (GPF (subtree e 'number) c s (sub1 n) x)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
   (list 'MINUS (tag 'number 'n))))
```

```
    (GPF_apply_unary_op (mk_unary_operator 'MINUS)
 (GPF (subtree e 'number) c s (sub1 n) x)
 s)

 (if (rule e (prodn (tag 'pre_computable_label_expression 'p)
    (tag 'CHARACTER_VALUE 'ch)))
     (GPF (subtree e 'CHARACTER_VALUE) c s (sub1 n) x)

 (if (rule e (prodn (tag 'pre_computable_label_expression 'p)
    (tag 'IDENTIFIER 'i)))
     (GPF (subtree e 'IDENTIFIER) c s (sub1 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)))
     (GPF_Gset_or_seq (subtree e 'set_or_seq_mark)
      (GPF_element_list (subtree e 'element_list)
c s (sub1 n) x)
      (GPF_element_type (subtree e 'element_list)
c s (sub1 n) x)
      s)

 (if (rule e (prodn (tag 'set_or_sequence_value 's)
    (tag 'range 'r)))
     (GPF_Gset_or_seq nil
      (GPF_element_list (subtree e 'range) c s (sub1 n) x)
      (GPF_element_type (subtree e 'range) c s (sub1 n) x)
      s)


 ; ********************
 ;
 ;    PARSE TREE LEAVES
 ;
 ; ********************

 (if (character_valuep e)
```

```
      (GPF_Gchar e s)

  (if (digit_listp e)
      (GPF_minteger e s)

  (if (entry_valuep e)
      (GPF_apply_var (entry_name e) s nil)

  (if (identifierp e)
      (GPF_apply (gname e) nil nil c s (sub1 n) x)

  (if (string_valuep e)
      (GPF_Gstring_seq e s)

    (mark_state_indeterminate s)))))))))))))))))))))))))))))))))))

  ( (lessp (count n)) ))


; ****************************************************************************
;
;                      THE PROCEDURAL INTERPRETER
;
; ****************************************************************************

; ===========================================================
;  Computation of Actual Data Parameters for Procedure Call
; ===========================================================

(defn GP_parg (e c s n x)
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (let ((e2 (mk_name_expression e)))
      (if (rule e2 (prodn (tag 'name_expression 'e)
  (tag 'IDENTIFIER 'i)))
  (let ((vn (gname (subtree e2 'IDENTIFIER))))
    (let ((r (GPF_apply_var vn s nil)))
      (if (normal_state r)
  (allocate 'result~ (name_exp vn nil) s)
r)))

      (if (rule e2 (prodn (tag 'name_expression 'e)
  (list (tag 'IDENTIFIER 'i)
```

```
(tag 'selector_list 'ss))))
  (let ((vn (gname (subtree e2 'IDENTIFIER)))
(ss (GPF_selectors (subtree e2 'selector_list)
   c s (sub1 n) x)))
    (let ((r (GPF_apply_var vn s ss)))
      (if (normal_state r)
  (allocate 'result~ (name_exp vn (result~_list ss)) s)
r)))

(GPF e c s (sub1 n) x)))))

  ( (lessp (count n)) ))

(defn GP_parg_list (as c s n x)
  (if (nlistp as)
      nil
  (if (zerop (fix n))
      (list (mark_state_indeterminate s))
    (cons (GP_parg (car as) c s (sub1 n) x)
  (GP_parg_list (cdr as) c s (sub1 n) x))))
  ( (lessp (count n)) ))



; ===================
;  The Procedure Call
; ===================

(defn GP_procedure_call (pn Sadp acp c s n x)
  ; pn is the called procedure name
  ; Sadp is the list of states resulting from evaluation of actual data
  ;      parameters
  ; acp is the list of actual condition paramters
  ; entry (normal_state s)
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (let ((h (car (ref pn c x)))
  (u (cdr (ref pn c x))))
      (if (or (equal (kind u) 'procedure)
      (equal (kind u) 'function))
  (let ((s1 (GP_call_state u h Sadp acp s x)))
    (if (normal_state s1)
(GP_map_call_effects
  (GP_procedure_body (procedure_body u) h s1 (sub1 n) x)
```

```
  u (result~_list Sadp) acp c s n x)
      (map_cond_effects (cond~ s1) (formal_cargs u) acp
(marked (mark s1) (map s)))))
(set_condition s 'routineerror)))) ; indeterminate?
  ( (lessp (count n)) ))

(defn GP_procedure_body (m c s n x)

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule m (prodn (tag 'procedure_body 'b)
     'PENDING))
      (mark_state_indeterminate s)

  (if (rule m (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)))
      (let ((r (GP_deallocate_locals
(GP (subtree m 'opt_internal_statements)
    c
    (GP_set_keep
     (keep_spec m)
     (GP_locals (subtree m 'opt_internal_environment)
c
(GP_set_entry (prec m) c s n x)
(sub1 n) x)
     c n x)
    (sub1 n) x))))
(if (indeterminate r)
    r
(if (or (equal (cond~ r) 'normal)
(conditionp (cond~ r) r))
    (GP_set_exit (exit_spec m) c r n x)
(if (equal (cond~ r) 'leave)
    ; leave statement was not in a loop
    (set_condition r 'routineerror)
  ; condition signalled was not a forward condition; we are not
  ; allowed to signal routineerror
  (mark_state_indeterminate r)))))
```

```
  (mark_state_indeterminate s))))

  ( (lessp (count n)) ))

(defn GP_locals (m c s n x)

  (if (not (normal_state s))
      s

  (if (zerop n)
      (mark_state_indeterminate s)

  (if (rule m (prodn (tag 'opt_internal_environment 'iv)
     'empty))
      s

  (if (rule m (prodn (tag 'opt_internal_environment 'iv)
     (tag 'internal_environment 'iv2)))
      (GP_locals (subtree m 'internal_environment) c s (sub1 n) x)

  (if (rule m (prodn (tag 'internal_environment 'iv)
     (tag 'internal_data_or_condition_objects 'iv2)))
      (GP_locals (subtree m 'internal_data_or_condition_objects)
 c s (sub1 n) x)

  (if (rule m (prodn (tag 'internal_environment 'iv)
     (list (tag 'internal_environment 'iv2)
   (tag 'internal_data_or_condition_objects 'iv3))))
      (GP_locals (subtree m 'internal_data_or_condition_objects) c
 (GP_locals (subtree m 'internal_environment) c s (sub1 n) x)
 (sub1 n) x)

  (if (rule m (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)))
      (let ((ie (internal_initial_value_exp m)))
      (let ((iv (if (equal ie nil)
    ie
  (GPF ie c s (sub1 n) x))))
(GP_bind_locals (access m)
```

```
(id_list (subtree m 'identifier_list))
(type_desc (subtree m 'type_specification) c nil x)
iv s)))

  (if (rule m (prodn (tag 'internal_data_or_condition_objects 'iv)
     (list 'COND (tag 'identifier_list 'is) 'SEMI_COLON)))

      (GP_local_conds (id_list (subtree m 'identifier_list)) s)

    (mark_state_indeterminate s)))))))))

  ( (lessp (count n)) ))



; ****************
;  Case Statement
; ****************

(defn GP_case_body (k m c s n x)

  (if (not (normal_state s))
      s

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule m (prodn (tag 'case_composition_body 'b)
     'empty))
      s

  (if (rule m (prodn (tag 'case_composition_body 'b)
     (list 'ELSE 'COLON (tag 'opt_internal_statements 'ss))))
      (GP (subtree m 'opt_internal_statements) c s (sub1 n) x)

  (if (rule m (prodn (tag 'case_composition_body 'b)
     (list 'IS (tag 'case_labels 'cs) 'COLON
   (tag 'opt_internal_statements 'ss)
   (tag 'case_composition_body 'b2))))
      (let ((s1 (GPF_Gin k
 (GPF_Gset (GPF_list (case_labels m) c s (sub1 n) x)
   (base_type (type (result~ k)))
   s)
 s)))
```

```
(if (normal_state s1)
    (if (Gtruep (result~ s1))
(GP (subtree m 'opt_internal_statements) c s (sub1 n) x)
      (GP_case_body k (subtree m 'case_composition_body)
    c s (sub1 n) x))
  s1))

    (mark_state_indeterminate s))))))

  ( (lessp (count n)) ))



; ****************************************************************
;
;                      HANDLING CONDITIONS
;
;  <opt_condition_handlers> ::= <empty> | WHEN <opt handler list>
;  <opt_handler_list> ::= empty | <handler_list>
;  <handler_list> ::= <handler> { <handler> }
;
; ****************************************************************

(defn GP_cond (m c s n x)
  (if (or (indeterminate s) (condition_normal s))
      s
  (if (or (zerop n)
  (not (condition_labels_ok (handler_labels m) s)))
      (mark_state_indeterminate s)
    (let ((h (handler m (cond~ s))))
      (if (equal h nil)
  s
(GP h c (set_condition s 'normal) (sub1 n) x)))))
  ( (lessp (count n)) ))



;; **** The strange commented characters in the following function allow GNU
;;      Emacs to count parentheses correctly.


(defn GP (m c s n x)
  ; The meta-function GP (m,c,s,n,x) gives the state that results
  ; when the program fragment m is interpreted for at most n steps, in the
```

```
; context of scope c, initial state s, and Gypsy sentence x.
;
; The domain and range of GP (m,c,s,n,x) are as follows:
;
; m is the parse tree which describes a computer program mechanism.
; c is the (litatom) name of the Gypsy scope in which m appears.
; s is the marked, initial state on which the mechanism m begins to operate.
; n is the maximum number of steps  the  mechanism  is  allowed  to perform.
; x is  the  parse  tree  of the program description sentence which
;   defines the library which  contains  the  declarations  of  the
;   Gypsy units which are referred to by m.
; GP(m,c,s,n,x) is  the  marked,  final  state which results from operating
;               the mechanism on the initial state s for at most n steps.
;               If the mechanism  has  not  completed all of its operation
;               in n steps, then the final state is marked as indeterminate.

(if (not (normal_state s))
    s

(if (zerop (fix n))
    (mark_state_indeterminate s)


; ***************************************************
;
;                  STATEMENT LISTS
;
;  <statement list> ::= <statement> {; <statement> }
;
; ***************************************************

(if (rule m (prodn (tag 'statement_list 'ss)
   (tag 'statement 's)))
    (GP (subtree m 'statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'statement_list 'ss)
   (list (tag 'statement_list 'ss2) 'SEMI_COLON
 (tag 'statement 's))))
    (GP (subtree m 'statement)
c
(GP (subtree m 'statement_list) c s (sub1 n) x)
(sub1 n)
x)
```

```
; **************************************************************************
;
;  <statement> ::=   <procedural statement> | <procedure composition rule>
;                  | <assert specification>
;
; **************************************************************************

(if (rule m (prodn (tag 'statement 's)
   (tag 'procedural_statement 's2)))
    (GP (subtree m 'procedural_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'statement 's)
   (tag 'procedure_composition_rule 's2)))
    (GP (subtree m 'procedure_composition_rule) c s (sub1 n) x)

(if (rule m (prodn (tag 'statement 's)
   (tag 'assert_specification 's2)))
    (GP (subtree m 'assert_specification) c s (sub1 n) x)


; ************************************************************
;
;                   PROCEDURAL STATEMENT
;
;  <procedural statement> ::=   <assignment statement>
;                             | <leave statement>
;                             | <move statement>
;                             | <new statement>
;                             | <procedure statement>
;                             | <remove statement>
;                             | <signal statement>
;
; ************************************************************

(if (rule m (prodn (tag 'procedural_statement 's)
   (tag 'assignment_statement 's2)))
    (GP (subtree m 'assignment_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
   (tag 'leave_statement 's2)))
    (GP (subtree m 'leave_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
   (tag 'move_statement 's2)))
```

```
    (GP (subtree m 'move_statement) c s (sub1 n) x)

 (if (rule m (prodn (tag 'procedural_statement 's)
    (tag 'new_statement 's2)))
     (GP (subtree m 'new_statement) c s (sub1 n) x)

 (if (rule m (prodn (tag 'procedural_statement 's)
    (tag 'procedure_statement 's2)))
     (GP (subtree m 'procedure_statement) c s (sub1 n) x)

 (if (rule m (prodn (tag 'procedural_statement 's)
    (tag 'remove_statement 's2)))
     (GP (subtree m 'remove_statement) c s (sub1 n) x)

 (if (rule m (prodn (tag 'procedural_statement 's)
    (tag 'signal_statement 's2)))
     (GP (subtree m 'signal_statement) c s (sub1 n) x)


 ; **********************
 ;
 ;  ASSIGNMENT STATEMENT
 ;
 ; **********************

 (if (rule m (prodn (tag 'assignment_statement 's)
    (list (tag 'name_expression 'n) 'COLON_EQUAL
  (tag 'expression 'e))))
     (GP_assign (GP_parg (subtree m 'name_expression) c s (sub1 n) x)
(GPF (subtree m 'expression) c s (sub1 n) x)
s c n x)


 ; *****************
 ;
 ;  LEAVE STATEMENT
 ;
 ; *****************

 (if (rule m (prodn (tag 'leave_statement 's)
    'LEAVE))
     (set_condition s 'leave)
```

```
; ***************************************************************************
;
;                              MOVE STATEMENT
;
;  <move_statement> ::= MOVE <removable component> <component destination>
;  <component destination> ::=   <new dynamic variable component>
;                              | TO <sequence element name expression>
;
; ***************************************************************************
                                                                       ; |<<
(if (rule m (prodn (tag 'move_statement 's)
   (list 'MOVE (tag 'removable_component 'c)
 (tag 'component_destination 'd))))
    (let ((e (remove_exp_arg m)))
(GP_move (if (equal e nil) nil (GPF e c s (sub1 n) x))
 (GP_parg (remove_name_arg m) c s (sub1 n) x)
 (subtree m 'component_destination)
 (GP_parg (new_name_arg m) c s (sub1 n) x)
 c s n x))


; *************************************************************************
;
;                            NEW STATEMENT
;
;  <new_statement> ::= NEW <expression> <new dynamic variable component>
;
; *************************************************************************

(if (rule m (prodn (tag 'new_statement 's)
   (list 'NEW (tag 'expression 'e)
 (tag 'new_dynamic_variable_component 'dc))))
    (GP_new (subtree m 'new_dynamic_variable_component)
    (GPF (subtree m 'expression) c s (sub1 n) x)
    (GP_parg (new_name_arg m) c s (sub1 n) x)
    c s n x)


; ****************
;
;  PROCECURE CALL
;
; ****************
```

```
  (if (rule m (prodn (tag 'procedure_statement 's)
     (list (tag 'IDENTIFIER 'pn)
   (tag 'arg_list 'dp)
   (tag 'opt_actual_condition_parameters 'cp))))
      (GP_procedure_call (gname (subtree m 'identifier))
 (GP_parg_list (actual_dargs m) c s (sub1 n) x)
 (actual_cargs m)
 c s (sub1 n) x)


  ; *********************************************************
  ;
  ;                    REMOVE STATEMENT
  ;
  ;  <remove_statement> ::= REMOVE <removable component>
  ;  <removable component> ::=
  ;           ELEMENT <expression> FROM SET <name expression>
  ;         | <name expression>
  ;
  ; *********************************************************
                                                             ; |
  (if (rule m (prodn (tag 'remove_statement 's)
     (list 'REMOVE (tag 'removable_component 'c))))
      (let ((e (remove_exp_arg m)))
(GP_remove (if (equal e nil) nil (GPF e c s (sub1 n) x))
   (GP_parg (remove_name_arg m) c s (sub1 n) x)
   c s n x))


  ; ******************
  ;
  ;  SIGNAL STATEMENT
  ;
  ; ******************

  (if (rule m (prodn (tag 'signal_statement 's)
     (list 'SIGNAL (tag 'IDENTIFIER 'c))))
      (if (conditionp (gname (subtree m 'IDENTIFIER)) s)
  (set_condition s (gname (subtree m 'IDENTIFIER)))
(mark_state_indeterminate s))


  ; **********************************************************************
  ;
```

```
;                       PROCEDURE COMPOSITION
;
;  <procedure composition> ::   <if_composition> | <case_composition>
;                             | <loop_composition> | <begin_composition>
;
; *********************************************************************
                                                                    ; |
(if (rule m (prodn (tag 'procedure_composition_rule 's)
   (tag 'if_composition 's2)))
    (GP (subtree m 'if_composition) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedure_composition_rule 's)
   (tag 'case_composition 's2)))
    (GP (subtree m 'case_composition) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedure_composition_rule 's)
   (tag 'loop_composition 's2)))
    (GP (subtree m 'loop_composition) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedure_composition_rule 's)
   (tag 'begin_composition 's2)))
    (GP (subtree m 'begin_composition) c s (sub1 n) x)


; *****************************************************************
;
;                         IF COMPOSITION
;
;  <if composition> ::=
;         IF <boolean expression> THEN <opt internal statements>
;           <if composition else part>
;           <opt condition handlers>
;         END
;
;  <if composition else part> ::=
;           <empty>
;         | ELSE <opt internal statements>
;         | ELIF <boolean expression> THEN <opt internal statements>
;                <if composition else part>
;
; *****************************************************************

(if (rule m (prodn (tag 'if_composition 's)
                   (list 'IF (tag 'expression 'b) 'THEN
```

```
   (tag 'opt_internal_statements 'ss)
   (tag 'if_composition_else_part 'ep)
   (tag 'opt_condition_handlers 'cs) 'END)))
      (let ((bv (GPF_type_check (boolean_desc)
(GPF (subtree m 'expression)
     c s (sub1 n) x)))
    (ep (if_statement_else_part m)))
(GP_cond (subtree m 'opt_condition_handlers) c
 (if (normal_state bv)
     (if (Gtruep (result~ bv))
 (GP (subtree m 'opt_internal_statements)
     c s (sub1 n) x)
       (if (equal ep nil)
   s
 (GP ep c s (sub1 n) x)))
   bv)
 (sub1 n) x))


  ; ***************************************************************************
  ;
  ;                            CASE COMPOSITION
  ;
  ;  <case composition> ::= CASE <label expression>
  ;                          { IS <case labels> : [ <internal statements> ] }
  ;                          [ ELSE             : [ <internal statements> ] ]
  ;                          [ <condition handlers> ]
  ;                         END
  ;
  ; ***************************************************************************

  (if (rule m (prodn (tag 'case_composition 's)
     (list 'CASE (tag 'expression 'e)
   (tag 'case_composition_body 'b)
   (tag 'opt_condition_handlers 'c) 'END)))
      (let ((k (GPF (subtree m 'expression) c s (sub1 n) x)))
      (let ((r (GP_case_label_check k
    (GPF_list (case_labels m) c s (sub1 n) x)
    s)))
(GP_cond (subtree m 'opt_condition_handlers) c
 (if (normal_state r)
     (GP_case_body k (subtree m 'case_composition_body)
   c s (sub1 n) x)
   r)
```

```
 (sub1 n) x)))


  ; ******************************************************
  ;
  ;                  LOOP COMPOSITION
  ;
  ;  <loop composition> ::= LOOP [ <internal statements> ]
  ;                              [ <condition handlers> ]
  ;                         END
  ;
  ; ******************************************************

  (if (rule m (prodn (tag 'loop_composition 's)
     (list 'LOOP (tag 'opt_internal_statements 'ss)
   (tag 'opt_condition_handlers 'c) 'END)))
      (let ((p1 (GP (subtree m 'opt_internal_statements) c s (sub1 n) x)))
(GP_cond (subtree m 'opt_condition_handlers) c
 (if (condition_non_normal p1)
     (reset_leave_to_normal p1)
   (GP m c p1 (sub1 n) x))
 (sub1 n) x))


  ; ********************************************************
  ;
  ;                    BEGIN COMPOSITION
  ;
  ;  <begin composition> ::= BEGIN [ <internal statements> ]
  ;                                [ <condition handlers> ]
  ;                          END
  ;
  ; ********************************************************

  (if (rule m (prodn (tag 'begin_composition 's)
     (list 'BEGIN (tag 'opt_internal_statements 'ss)
   (tag 'opt_condition_handlers 'c) 'END)))
      (GP_cond (subtree m 'opt_condition_handlers) c
       (GP (subtree m 'opt_internal_statements) c s (sub1 n) x)
       (sub1 n) x)


  ; *************************************************************
  ;
```

```
;                      INTERNAL STATEMENTS
;
;  <opt_internal_statements> ::= <empty> | <statement list> [;]
;                                        | PENDING [;]
;
; ************************************************************

(if (rule m (prodn (tag 'opt_internal_statements 'ss)
   'empty))
    s

(if (rule m (prodn (tag 'opt_internal_statements 'ss)
   (list (tag 'statement_list 'ss2) 'opt_semi_colon)))
    (GP (subtree m 'statement_list) c s (sub1 n) x)

(if (rule m (prodn (tag 'opt_internal_statements 'ss)
   (list 'PENDING 'opt_semi_colon)))
    (mark_state_indeterminate s)


; *********************
;
;  ASSERT SPECIFICATION
;
; *********************

; Whenever an assertion is encountered in executable code, it is evaluated
; and the result AND'd to the assertion-accumulator component of the map.

(if (rule m (prodn (tag 'assert_specification 's)
   (list 'ASSERT (tag 'specification_expression 'e))))
    (GP (subtree m 'specification_expression) c s (sub1 n) x)

(if (rule m (prodn (tag 'specification_expression 'e)
   (tag 'non_validated_specification_expression 'e2)))
    (GP_update_assert (expression_from_spec m) c s n x)

(if (or (rule m (prodn (tag 'specification_expression 'e)
(tag 'validated_specification_expression 'e2)))
 (rule m (prodn (tag 'specification_expression 'e)
(list 'OPEN_PAREN
      (tag 'validated_specification_expression 'e2)
      'CLOSE_PAREN))))
    (GP (subtree m 'validated_specification_expression)
```

```
  c s (sub1 n) x)

  (if (rule m (prodn (tag 'validated_specification_expression 'e)
     (list (tag 'non_validated_specification_expression 'e2)
   'OTHERWISE (tag 'IDENTIFIER 'i))))
      (let ((r (GPF_type_check (boolean_desc)
       (GPF (expression_from_spec m)
    c s (sub1 n) x))))
(if (normal_state r)
    (if (Gtruep (result~ r))
(GP_record_assert (result~ r) s)
      (GP (mk_signal_stmt (subtree m 'IDENTIFIER))
  c (GP_record_assert (result~ r) s) (sub1 n) x))
  r))

    (mark_state_indeterminate s))))))))))))))))))))))))))))))))))))

  ( (lessp (count n)) ))

))
|#
```

DEFINITION:
mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-gpf_

= **case on** *mutual-flg*:

```
case = gp
then if ¬ normal_state (s)  then s
       elseif fix (n) ≃ 0  then mark_state_indeterminate (s)
       elseif rule (m,
                     prodn (tag ('statement_list, 'ss),
                            tag ('statement, 's)))
       then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
       elseif rule (m,
                     prodn (tag ('statement_list, 'ss),
                            list (tag ('statement_list, 'ss2),
                                  'semi_colon,
                                  tag ('statement, 's))))
       then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
        prodn (tag ('statement, 's),
              tag ('procedural_statement, 's2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

elseif rule (m,
prodn (tag (`'statement`, `'s`),
tag (`'procedure_composition_rule`, `'s2`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

elseif rule (m,
prodn (tag (`'statement`, `'s`),
tag (`'assert_specification`, `'s2`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

```
elseif rule (m,
        prodn (tag ('procedural_statement, 's),
               tag ('assignment_statement, 's2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
        prodn (tag ('procedural_statement, 's),
               tag ('leave_statement, 's2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

elseif rule (m,
prodn (tag (`'procedural_statement`, `'s`),
tag (`'move_statement`, `'s2`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

elseif rule (m,
prodn (tag (`'procedural_statement`, `'s`),
tag (`'new_statement`, `'s2`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

```
elseif rule (m,
            prodn (tag ('procedural_statement, 's),
                   tag ('procedure_statement, 's2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
            prodn (tag ('procedural_statement, 's),
                   tag ('remove_statement, 's2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
             prodn (tag ('procedural_statement, 's),
                    tag ('signal_statement, 's2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
             prodn (tag ('assignment_statement, 's),
                    list (tag ('name_expression, 'n),
                         'colon_equal,
                         tag ('expression, 'e))))
then gp_assign (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp

s,
c,
n,
x)
elseif rule (m, prodn (tag ('leave_statement, 's), 'leave))
then set_condition (s, 'leave)
elseif rule (m,

```
prodn (tag ('move_statement, 's),
       list ('move,
             tag ('removable_component, 'c),
             tag ('component_destination, 'd))))
then gp_move (if remove_exp_arg (m) = nil then nil
              else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_ca

              mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp

              subtree (m, 'component_destination),
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp

```
                     c,
                     s,
                     n,
                     x)
elseif rule (m,
            prodn (tag ('new_statement, 's),
                   list ('new,
                         tag ('expression, 'e),
                         tag ('new_dynamic_variable_component,
                              'dc))))
then gp_new (subtree (m, 'new_dynamic_variable_component),
             mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_p
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_p

```
            c,
            s,
            n,
            x)
elseif rule (m,
         prodn (tag ('procedure_statement, 's),
               list (tag ('identifier, 'pn),
                    tag ('arg_list, 'dp),
                    tag ('opt_actual_condition_parameters,
                         'cp))))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
            prodn (tag ('remove_statement, 's),
                   list ('remove,
                        tag ('removable_component, 'c))))
then gp_remove (if remove_exp_arg (m) = nil then nil
                else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-g

c,
s,
n,
x)
elseif rule (m,
prodn (tag ('`signal_statement`, '`s`),
list ('`signal`, tag ('`identifier`, '`c`))))
then **if** conditionp (gname (subtree (m, '`identifier`)), s)
then set_condition (s, gname (subtree (m, '`identifier`)))
else mark_state_indeterminate (s) **endif**
elseif rule (m,
prodn (tag ('`procedure_composition_rule`, '`s`),
tag ('`if_composition`, '`s2`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

elseif rule (m,
prodn (tag (`'procedure_composition_rule`, `'s`),
tag (`'case_composition`, `'s2`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

elseif rule (m,
prodn (tag (`'procedure_composition_rule`, `'s`),
tag (`'loop_composition`, `'s2`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

```
elseif rule (m,
        prodn (tag ('procedure_composition_rule, 's),
            tag ('begin_composition, 's2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
        prodn (tag ('if_composition, 's),
```

```
list ('if,
     tag ('expression, 'b),
     'then,
     tag ('opt_internal_statements, 'ss),
     tag ('if_composition_else_part,
         'ep),
     tag ('opt_condition_handlers, 'cs),
     'end)))
```

then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

```
elseif rule (m,
        prodn (tag ('case_composition, 's),
            list ('case,
                tag ('expression, 'e),
                tag ('case_composition_body, 'b),
                tag ('opt_condition_handlers, 'c),
                'end)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
            prodn (tag ('loop_composition, 's),
                   list ('loop,
                         tag ('opt_internal_statements, 'ss),
                         tag ('opt_condition_handlers, 'c),
                         'end)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
            prodn (tag ('begin_composition, 's),
                   list ('begin,
                        tag ('opt_internal_statements, 'ss),
                        tag ('opt_condition_handlers, 'c),
                        'end)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
        prodn (tag ('opt_internal_statements, 'ss),
            'empty))  then s
elseif rule (m,
        prodn (tag ('opt_internal_statements, 'ss),
            list (tag ('statement_list, 'ss2),
                'opt_semi_colon)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
            prodn (tag ('opt_internal_statements, 'ss),
                   '(pending opt_semi_colon)))
then mark_state_indeterminate (s)
elseif rule (m,
            prodn (tag ('assert_specification, 's),
                   list ('assert,
                        tag ('specification_expression, 'e))))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
            prodn (tag ('specification_expression, 'e),
                   tag ('non_validated_specification_expression,
                        'e2)))
```

```
then gp_update_assert (expression_from_spec (m), c, s, n, x)
elseif rule (m,
             prodn (tag ('specification_expression, 'e),
                    tag ('validated_specification_expression,
                         'e2)))
       ∨  rule (m,
                 prodn (tag ('specification_expression,
                             'e),
                        list ('open_paren,
                              tag ('validated_specification_expression,
                                   'e2),
                              'close_paren)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
elseif rule (m,
             prodn (tag ('validated_specification_expression,
                         'e),
                    list (tag ('non_validated_specification_expression,
                               'e2),
                         'otherwise,
                         tag ('identifier, 'i))))
then if normal_state (gpf_type_check (BOOLEAN_DESC,
                                      mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body
```

then if gtruep (result˜ (gpf_type_check (BOOLEAN_DESC,
mutual-gp-gp_cond-gp_case_body-gp_locals-gp_proced

then gp_record_assert (result˜ (gpf_type_check (BOOLEAN_DESC,
mutual-gp-gp_cond-gp_case_body-gp_loca

$s)$

else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-g

else gpf_type_check (BOOLEAN_DESC,
mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proced

else mark_state_indeterminate (s) **endif**
case $=$ *gp_cond*
then if indeterminate $(s) \lor$ condition_normal (s) **then** s
elseif $(n \simeq 0)$
$\lor$ $(\neg$ condition_labels_ok (handler_labels (m), $s))$
then mark_state_indeterminate (s)
elseif handler $(m$, cond˜ $(s)) =$ **nil** **then** s
else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

```
case = gp_case_body
 then if ¬ normal_state (s)  then s
      elseif fix (n) ≃ 0  then mark_state_indeterminate (s)
      elseif rule (m,
                prodn (tag ('case_composition_body, 'b),
                      'empty))  then s
      elseif rule (m,
                prodn (tag ('case_composition_body, 'b),
                      list ('else,
                          'colon,
                          tag ('opt_internal_statements,
                              'ss))))
      then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (m,
            prodn (tag ('case_composition_body, 'b),
                   list ('is,
                        tag ('case_labels, 'cs),
                        'colon,
                        tag ('opt_internal_statements,
                             'ss),
                        tag ('case_composition_body, 'b2))))
then if normal_state (gpf_gin (k,
                                gpf_gset (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_b
```

```
                                          base_type (type (result~ (k))),
                                          s),
                                s))
      then if gtruep (result~ (gpf_gin (k,
                                        gpf_gset (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_proc
```

base_type (type (result˜ (k))),
s),
s)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call

else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-

```
            else gpf_gin (k,
                          gpf_gset (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_pro
```

```
                                         base_type (type (result~ (k))),
                                         s),
                          s) endif
        else mark_state_indeterminate (s) endif
case = gp_locals
 then if ¬ normal_state (s)  then s
        elseif n ≃ 0  then mark_state_indeterminate (s)
        elseif rule (m,
                     prodn (tag ('opt_internal_environment, 'iv),
                            'empty))  then s
        elseif rule (m,
                     prodn (tag ('opt_internal_environment, 'iv),
                            tag ('internal_environment, 'iv2)))
        then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

elseif rule (m,
prodn (tag ('internal_environment, 'iv),
tag ('internal_data_or_condition_objects,
'iv2)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

elseif rule (m,
prodn (tag ('internal_environment, 'iv),
list (tag ('internal_environment, 'iv2),
tag ('internal_data_or_condition_objects,
'iv3))))

```
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

elseif rule (m,
            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 gp_bind_locals (access (m),
                               id_list (subtree (m, 'identifier_list)),
                               type_desc (subtree (m,
                                                   'type_specification),
                                          c,
                                          nil,
                                          x),
                               if internal_initial_value_exp (m) = nil
                               then internal_initial_value_exp (m)
                               else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proce
```

```
                               s)
          elseif rule (m,
                       prodn (tag ('internal_data_or_condition_objects,
                                   'iv),
                              list ('cond,
                                    tag ('identifier_list, 'is),
                                    'semi_colon)))
          then gp_local_conds (id_list (subtree (m, 'identifier_list)),
                               s)
          else mark_state_indeterminate (s) endif
case = gp_procedure_body
```

```
then if fix (n) ≃ 0  then mark_state_indeterminate (s)
     elseif rule (m, prodn (tag ('procedure_body, 'b), 'pending))
     then mark_state_indeterminate (s)
     elseif rule (m,
                  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 if indeterminate (gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedu
```

then gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp

elseif (cond˜ (gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure

$= \quad$ `'normal`)

$\vee \quad$ conditionp (cond˜ (gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_local gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_pr

then gp_set_exit (exit_spec (m),
c,
gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_proced

n,

x)

elseif cond˜ (gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_l

$=$ `'leave`

then set_condition (gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_proc

'routineerror)
else mark_state_indeterminate (gp_deallocate_locals (mutual-gp-gp_cond-gp_case_body-gp_loc

else mark_state_indeterminate (s) **endif**

case $=$ *gp_procedure_call*

then if fix (n) $\simeq$ `0` **then** mark_state_indeterminate (s)

elseif (kind (cdr (ref ($pn,\ c,\ x$))) $=$ `'procedure`)

$\vee$ (kind (cdr (ref ($pn,\ c,\ x$))) $=$ `'function`)

then if normal_state (gp_call_state (cdr (ref ($pn,\ c,\ x$)),

car (ref ($pn,\ c,\ x$)),

$sadp$,

acp,

s,

x))

then gp_map_call_effects (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_

```
                              cdr (ref (pn, c, x)),
                              result~_list (sadp),
                              acp,
                              c,
                              s,
                              n,
                              x)
          else map_cond_effects (cond~ (gp_call_state (cdr (ref (pn,
                                                                c,
                                                                x)),
                                                       car (ref (pn,
                                                                c,
                                                                x)),
                                                       sadp,
                                                       acp,
                                                       s,
                                                       x)),
                                formal_cargs (cdr (ref (pn, c, x))),
                                acp,
                                marked (mark (gp_call_state (cdr (ref (pn,
                                                                      c,
                                                                      x)),
                                                             car (ref (pn,
                                                                      c,
                                                                      x)),
                                                             sadp,
                                                             acp,
                                                             s,
                                                             x)),
                                       map (s))) endif
     else set_condition (s, 'routineerror) endif
case = gp_parg_list
 then if as ≃ nil then nil
     elseif fix (n) ≃ 0 then list (mark_state_indeterminate (s))
     else cons (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par

```
case = gp_parg
 then if fix (n) ≃ 0  then mark_state_indeterminate (s)
        elseif rule (mk_name_expression (e),
                     prodn (tag ('name_expression, 'e),
                            tag ('identifier, 'i)))
        then if normal_state (gpf_apply_var (gname (subtree (mk_name_expression (e),
                                                              'identifier)),
                                             s,
                                             nil))
              then allocate ('result~,
                             name_exp (gname (subtree (mk_name_expression (e),
                                                       'identifier)),
                                       nil),
```

```
                s)
        else gpf_apply_var (gname (subtree (mk_name_expression (e),
                                            'identifier)),
                            s,
                            nil) endif
elseif rule (mk_name_expression (e),
             prodn (tag ('name_expression, 'e),
                    list (tag ('identifier, 'i),
                          tag ('selector_list, 'ss))))
then if normal_state (gpf_apply_var (gname (subtree (mk_name_expression (e),
                                                     'identifier)),
                                     s,
                                     mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_bod
```

```
        then allocate ('result~,
                       name_exp (gname (subtree (mk_name_expression (e),
                                                 'identifier)),
                                 result~_list (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedu
```

s)
else gpf_apply_var (gname (subtree (mk_name_expression (e),
`'identifier`)),
s,
mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proced

else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-

```
case = gpf
 then if ¬ normal_state (s)  then s
      elseif fix (n) ≃ 0  then mark_state_indeterminate (s)
      elseif rule (e,
                   prodn (tag ('expression, 'e),
                          tag ('modified_primary_value, 'm)))
      then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
      elseif rule (e,
                   prodn (tag ('expression, 'e),
                          list ('all, tag ('bound_expression, 'b))))
      then if normal_state (gpf_bound_values (e, c, s, x))
           then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa
```

```
        else gpf_bound_values (e, c, s, x) endif
elseif rule (e,
             prodn (tag ('expression, 'e),
                    list ('some, tag ('bound_expression, 'b))))
then if normal_state (gpf_bound_values (e, c, s, x))
        then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa
```

```
         else gpf_bound_values (e, c, s, x) endif
elseif rule (e,
                 prodn (tag ('expression, 'e),
                            list (tag ('unary_operator, 'op),
                                  tag ('expression, 'e2))))
then gpf_apply_unary_op (subtree (e, 'unary_operator),
                                          mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proce
```

```
                                          s)
elseif rule (e,
                 prodn (tag ('expression, 'e),
                            list (tag ('expression, 'e1),
                                  tag ('binary_operator, 'op),
                                  tag ('expression, 'e2))))
then gpf_apply_binary_op (subtree (e, 'binary_operator),
                                           mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proce
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proc

s)
elseif rule (e,
prodn (tag (`'modified_primary_value`, `'m`),
tag (`'primary_value`, `'p`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

```
elseif rule (e,
             prodn (tag ('modified_primary_value, 'm),
                    list (tag ('modified_primary_value, 'm2),
                          tag ('value_modifiers, 'vm))))
then if fn_call_formp (e)
      then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa
```

else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa

```
elseif rule (e,
           prodn (tag ('modified_primary_value, 'm),
                  list (tag ('modified_primary_value, 'm2),
                        tag ('actual_condition_parameters,
                             'cp))))
then if fn_call_formp (e)
     then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa
```

```
        else set_condition (s, 'routineerror) endif
elseif rule (e,
            prodn (tag ('primary_value, 'p),
                   tag ('literal_value, 'l)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('primary_value, 'p),
                   tag ('set_or_sequence_value, 's)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('primary_value, 'p),
                   tag ('entry_value, 'e)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('primary_value, 'p),
                   tag ('identifier, 'on)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('primary_value, 'p),
                   tag ('if_expression, 'i)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('primary_value, 'p),
                   list ('open_paren,
                        tag ('expression, 'e),
                        'close_paren)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

elseif rule (e,
    prodn (tag (`'constant_body`, `'b`),
        tag (`'expression`, `'e`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

elseif rule (e, prodn (tag (`'constant_body`, `'b`), `'pending`))
then mark_state_indeterminate (s)

```
elseif rule (e,
            prodn (tag ('if_expression, 'i),
                   list ('if,
                         tag ('expression, 'b),
                         'then,
                         tag ('expression, 'p),
                         tag ('if_expression_else_part, 'e))))
then if ¬ normal_state (gpf_type_check (BOOLEAN_DESC,
                                        mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_b
```

```
      then gpf_type_check (BOOLEAN_DESC,
                           mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proc
```

elseif gtruep (result˜ (gpf_type_check (BOOLEAN_DESC,
mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedu

then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa

else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par

elseif rule (e,
prodn (tag (`'literal_value`, `'l`),
tag (`'character_value`, `'ch`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

elseif rule (e,
    prodn (tag (`'literal_value`, `'l`),
        tag (`'number`, `'n`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

elseif rule (e,
    prodn (tag (`'literal_value`, `'l`),
        tag (`'string_value`, `'s`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

```
elseif rule (e,
        prodn (tag ('number, 'n), tag ('digit_list, 's)))
then gpf_minteger (e, s)
elseif rule (e,
        prodn (tag ('number, 'n),
              list (tag ('base, 'b), tag ('digit_list, 's))))
then gpf_minteger (e, s)
elseif rule (e,
        prodn (tag ('pre_computable_label_expression,
                   'p),
              tag ('number, 'n)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
        prodn (tag ('pre_computable_label_expression,
                   'p),
              list ('minus, tag ('number, 'n))))
then gpf_apply_unary_op (mk_unary_operator ('minus),
                        mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proce
```

```
                              s)
elseif rule (e,
           prodn (tag ('pre_computable_label_expression,
                      'p),
                 tag ('character_value, 'ch)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
          prodn (tag ('pre_computable_label_expression,
                      'p),
                 tag ('identifier, 'i)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
          prodn (tag ('set_or_sequence_value, 's),
                 list ('open_paren,
                     tag ('set_or_seq_mark, 'm),
                     tag ('element_list, 'e),
                     'close_paren)))
then gpf_gset_or_seq (subtree (e, 'set_or_seq_mark),
                 mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure

```
                  s)
elseif rule (e,
        prodn (tag ('set_or_sequence_value, 's),
              tag ('range, 'r)))
then gpf_gset_or_seq (nil,
                  mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure

s)
elseif character_valuep (e) **then** gpf_gchar (e, s)
elseif digit_listp (e) **then** gpf_minteger (e, s)
elseif entry_valuep (e) **then** gpf_apply_var (entry_name (e), s, **nil**)
elseif identifierp (e)
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

elseif string_valuep$\,(e)$ **then** gpf_gstring_seq$\,(e,\ s)$
else mark_state_indeterminate$\,(s)$ **endif**
case $=$ *gpf_list*
then if $es \simeq$ **nil then nil**
elseif fix$\,(n) \simeq 0$ **then** list (mark_state_indeterminate$\,(s)$)
else cons (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par

```
case = gpf_apply
 then if state_componentp (fn, s)
       then if acp = nil then gpf_apply_var (fn, s, adp)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'false
       then if acp = nil then gpf_select_op (gpf_false (s), adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'true
       then if acp = nil then gpf_select_op (gpf_true (s), adp, s)
             else set_condition (s, 'routineerror) endif
       elseif type_name_expp (fn, adp, sn, x)
       then if acp = nil then gpf_type_name_arg (fn, sn, s, x)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'domain
       then if acp = nil then gpf_std_domain (adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'first
       then if acp = nil then gpf_std_first (adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'initial
       then if acp = nil then gpf_std_initial (adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'last
       then if acp = nil then gpf_std_last (adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'lower
       then if acp = nil then gpf_std_lower (adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'max
       then if acp = nil then gpf_std_max (adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'min
       then if acp = nil then gpf_std_min (adp, s)
             else set_condition (s, 'routineerror) endif
       elseif fn = 'nonfirst
       then if acp = nil then gpf_std_nonfirst (adp, s)
             else set_condition (s, 'routineerror) endif
```

```
elseif fn = 'nonlast
then if acp = nil then gpf_std_nonlast (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'null
then if acp = nil then gpf_std_null (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'ord
then if acp = nil then gpf_std_ord (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'pred
then if acp = nil then gpf_std_pred (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'range
then if acp = nil then gpf_std_range (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'scale
then if acp = nil then gpf_std_scale (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'size
then if acp = nil then gpf_std_size (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'succ
then if acp = nil then gpf_std_succ (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fn = 'upper
then if acp = nil then gpf_std_upper (adp, s)
     else set_condition (s, 'routineerror) endif
elseif fix (n) ≃ 0 then mark_state_indeterminate (s)
else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-
```

```
case = gpf_apply_fun
 then if fix (n) ≃ 0  then mark_state_indeterminate (s)
      elseif kind (cdr (ref (fn, sn, x))) = 'function
      then gpf_select_op (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_ca
```

```
                              if length (formal_dargs (cdr (ref (fn, sn, x))))
                                 =  0  then adp
                              else nil endif,
                              s)
      elseif kind (cdr (ref (fn, sn, x))) = 'constant
      then if acp = nil
           then gpf_select_op (gpf_retype_result˜ (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_proced
```

```
                                        type_desc (result_type (cdr (ref (fn,
                                                                          sn,
                                                                          x))),
                                                   car (ref (fn,
                                                             sn,
                                                             x)),
                                                   nil,
                                                   x)),
                        adp,
                        s)
          else set_condition (s, 'routineerror) endif
      else set_condition (s, 'routineerror) endif
case = gpf_modifiers
 then if ¬ normal_state (s)  then s
      elseif ¬ normal_state (sbv)  then sbv
      elseif fix (n) ≃ 0  then mark_state_indeterminate (s)
      elseif rule (e,
                   prodn (tag ('value_modifiers, 'm),
                          tag ('component_selectors, 's)))
      then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('component_selectors, 's),
                   list ('dot, tag ('identifier, 'fn))))
then gpf_record_get (sbv,
                     allocate ('result~,
                               marked ('field_name,
                                       gname (subtree (e,
                                                       'identifier))),
                               s),
                     s)
elseif rule (e,
            prodn (tag ('component_selectors, 's),
                   tag ('arg_list, 'd)))
then gpf_select_op (sbv,
                    mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_ca
```

```
                   s)
elseif rule (e,
          prodn (tag ('value_modifiers, 'm),
                 tag ('range, 'r)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
          prodn (tag ('range, 'r),
                 list ('open_paren,
                     tag ('range_limits, 'r2),
                     'close_paren)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
        prodn (tag ('range_limits, 'r),
            list (tag ('expression, 'lo),
                'dot_dot,
                tag ('expression, 'hi))))
then gpf_subsequence_get (sbv,
                mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proc
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_proc

```
                 s)
elseif rule (e,
        prodn (tag ('value_modifiers, 'm),
             tag ('value_alterations, 'a)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list
```

```
elseif rule (e,
        prodn (tag ('value_alterations, 'a),
             list ('with,
                'open_paren,
                tag ('component_alterations_list,
                   'al),
                'close_paren)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list
```

```
elseif rule (e,
            prodn (tag ('component_alterations_list, 'al),
                   tag ('component_alterations, 'a)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('component_alterations_list, 'al),
                   list (tag ('component_alterations_list,
                              'al2),
                        'semi_colon,
```

```
                tag ('component_alterations, 'a))))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
          prodn (tag ('component_alterations, 'as),
                list (tag ('opt_each_clause, 'e),
                      tag ('component_assignment, 'a))))
```

```
then if each_clausep (subtree (e, 'opt_each_clause))
        then if normal_state (gpf_bound_values (subtree (e,
                                                    'opt_each_clause),
                                               c,
                                               s,
                                               x))
                then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call
```

```
                else gpf_bound_values (subtree (e,
                                                'opt_each_clause),
                                       c,
                                       s,
                                       x) endif
        else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa
```

```
elseif rule (e,
            prodn (tag ('component_alterations, 'as),
                   list (tag ('opt_each_clause, 'e),
                         tag ('component_creation, 'c))))
then if each_clausep (subtree (e, 'opt_each_clause))
      then if normal_state (gpf_bound_values (subtree (e,
                                                       'opt_each_clause),
                                              c,
                                              s,
                                              x))
            then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call
```

```
          else gpf_bound_values (subtree (e,
                                          'opt_each_clause),
                                 c,
                                 s,
                                 x) endif
      else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par
```

```
elseif rule (e,
             prodn (tag ('component_alterations, 'as),
                    list (tag ('opt_each_clause, 'e),
                          tag ('component_deletion, 'd))))
then if each_clausep (subtree (e, 'opt_each_clause))
      then if normal_state (gpf_bound_values (subtree (e,
                                                        'opt_each_clause),
                                              c,
                                              s,
                                              x))
            then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call
```

```
     else gpf_bound_values (subtree (e,
                                     'opt_each_clause),
                            c,
                            s,
                            x) endif
else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par
```

elseif rule (e,
prodn (tag (`'component_assignment`, `'a`),
list (tag (`'selector_list`, `'s`),
`'colon_equal`,
tag (`'expression`, `'e`))))
then gpf_put_op (sbv,
mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call

```
                    s)
elseif rule (e,
          prodn (tag ('component_creation, 'c),
                list ('before,
                    tag ('selector_list, 's),
                    'colon_equal,
                    tag ('expression, 'e))))
then gpf_put_op (sbv,
                rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedur
```

```
                gpf_gseq_insert_before (gpf_select_op (sbv,
                                                       rcdr (mutual-gp-gp_cond-gp_case_body-gp_lo
```

s),
rcar (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_proce

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_

```
                                       s),
                s)
elseif rule (e,
          prodn (tag ('component_creation, 'c),
                list ('behind,
                     tag ('selector_list, 's),
                     'colon_equal,
                     tag ('expression, 'e))))
then gpf_put_op (sbv,
                rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedur
```

```
                gpf_gseq_insert_behind (gpf_select_op (sbv,
                                                      rcdr (mutual-gp-gp_cond-gp_case_body-gp_l
```

s),
rcar (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_proce

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure.

```
                                        s),
                   s)
elseif rule (e,
            prodn (tag ('component_creation, 'c),
                   list ('into,
                         tag ('selector_list, 's),
                         'colon_equal,
                         tag ('expression, 'e))))
then gpf_put_op (sbv,
                 rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedur

                 gpf_gmap_insert (gpf_select_op (sbv,
                                                 rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-g
```

s),
rcar (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_b

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-

```
                              s),
                   s)
elseif rule (e,
            prodn (tag ('component_deletion, 'd),
                   list ('seqomit, tag ('selector_list, 's))))
then gpf_put_op (sbv,
                 rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedur
```

```
                 gpf_gseqomit (gpf_select_op (sbv,
                                              rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_
```

```
                    s),
          rcar (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_bod

                s),
        s)
elseif rule (e,
         prodn (tag ('component_deletion, 'd),
                list ('mapomit, tag ('selector_list, 's))))
then gpf_put_op (sbv,
        rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedur
```

gpf_gmapomit (gpf_select_op (sbv,

rcdr (mutual-gp-gp_cond-gp_case_body-gp_locals-gp

s),

rcar (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_bo

```
                                        s),
                          s)
       else mark_state_indeterminate (s) endif
case = gpf_selectors
 then if ¬ normal_state (s)  then list (s)
       elseif fix (n) ≃ 0  then list (mark_state_indeterminate (s))
       elseif rule (e,
                    prodn (tag ('selector_list, 's),
                           tag ('component_selectors, 's2)))
       then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list
```

```
       elseif rule (e,
```

```
        prodn (tag ('selector_list, 's),
               list (tag ('selector_list, 's2),
                     tag ('component_selectors, 's3))))
then append (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp

             mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp

elseif rule (e,
```

```
            prodn (tag ('component_selectors, 's),
                   list ('dot, tag ('identifier, 'fn))))
then list (allocate ('result~,
                     marked ('field_name,
                             gname (subtree (e, 'identifier))),
                     s))
elseif rule (e,
            prodn (tag ('component_selectors, 's),
                   tag ('arg_list, 'd)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
elseif rule (e,
            prodn (tag ('arg_list, 'as),
                   list ('open_paren,
                         tag ('value_list, 'vs),
                         'close_paren)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
        else list (mark_state_indeterminate (s)) endif
case = gpf_adp
 then if ¬ normal_state (s)  then list (s)
        elseif fix (n) ≃ 0  then list (mark_state_indeterminate (s))
        elseif rule (e,
                    prodn (tag ('arg_list, 'as),
                           list ('open_paren,
                                tag ('value_list, 'vs),
                                'close_paren)))
        then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
        elseif rule (e,
                    prodn (tag ('value_list, 'vs),
                           tag ('expression, 'e)))
        then rcons (nil,
                     mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_p
```

```
elseif rule (e,
            prodn (tag ('value_list, 'vs),
                   list (tag ('value_list, 'vs2),
                        'comma,
                        tag ('expression, 'e))))
then rcons (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_p
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_p

else list (mark_state_indeterminate (s)) **endif**
case $=$ *gpf_each*
then if $\neg$ normal_state (s) **then** s
elseif $vs \simeq$ **nil** **then** sbv
elseif gp_new_namep (id, s)
then if $n \simeq 0$ **then** mark_state_indeterminate (s)
else mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_pa

```
        else set_condition (s, 'routineerror) endif
case = gpf_some
 then if vs ≃ nil  then gpf_false (s)
        elseif gp_new_namep (id, s)
        then if n ≃ 0  then mark_state_indeterminate (s)
               else gpf_gor (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_ca
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_ca

```
                              s) endif
         else set_condition (s, 'routineerror) endif
case = gpf_all
 then if vs ≃ nil then gpf_true (s)
         elseif gp_new_namep (id, s)
         then if n ≃ 0 then mark_state_indeterminate (s)
                else gpf_gand (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_

```
                          s) endif
      else set_condition (s, 'routineerror) endif
case = gpf_element_type
 then if fix (n) ≃ 0  then mark_state_indeterminate (s)
      elseif rule (e,
                   prodn (tag ('range, 'r),
                          list ('open_paren,
                                tag ('range_limits, 'r2),
                                'close_paren)))
      then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

elseif rule (e,
prodn (tag (`'element_list`, `'e`),
tag (`'value_list`, `'v`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

elseif rule (e,
prodn (tag (`'element_list`, `'e`),
tag (`'range_limits`, `'r`)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis

```
elseif rule (e,
            prodn (tag ('range_limits, 'r),
                   list (tag ('expression, 'lo),
                        'dot_dot,
                        tag ('expression, 'hi))))
then  base_type (type (result~ (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_p
```

```
elseif rule (e,
            prodn (tag ('value_list, 'v),
                   tag ('expression, 'e)))
then  base_type (type (result~ (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_p
```

```
elseif rule (e,
            prodn (tag ('value_list, 'v),
                   list (tag ('value_list, 'v2),
                         'comma,
                         tag ('expression, 'e))))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_lis
```

```
        else nil endif
otherwise if ¬ normal_state (s)  then list (s)
          elseif fix (n) ≃ 0  then list (mark_state_indeterminate (s))
          elseif rule (e,
                       prodn (tag ('range, 'r),
                              list ('open_paren,
                                    tag ('range_limits, 'r2),
                                    'close_paren)))
          then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par
```

```
          elseif rule (e,
                       prodn (tag ('element_list, 'e),
                              tag ('value_list, 'v)))
          then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par
```

```
elseif rule (e,
            prodn (tag ('element_list, 'e),
                   tag ('range_limits, 'r)))
then mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_par
```

```
elseif rule (e,
            prodn (tag ('range_limits, 'r),
                   list (tag ('expression, 'lo),
                        'dot_dot,
                        tag ('expression, 'hi))))
then gpf_grange_elements (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_

s)
elseif rule (e,
prodn (tag ('`value_list`, '`v`),
tag ('`expression`, '`e`)))
then rcons (**nil**,
mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-

```
elseif rule (e,
            prodn (tag ('value_list, 'v),
                   list (tag ('value_list, 'v2),
                        'comma,
                        tag ('expression, 'e))))
then  rcons (mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-
```

mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-

else list (mark_state_indeterminate (s)) **endif** **endcase**

DEFINITION:
gp (m, c, s, n, x)
= mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gp_cond (m, c, s, n, x)
= mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gp_case_body $(k,\ m,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gp_locals $(m,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gp_procedure_body $(m,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gp_procedure_call $(pn,\ sadp,\ acp,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

Definition:

gp_parg_list $(as,\ c,\ s,\ n,\ x)$

$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

Definition:

gp_parg $(e,\ c,\ s,\ n,\ x)$

$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf $(e,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_list $(es,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

Definition:
gpf_apply (fn, adp, acp, sn, s, n, x)
= mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

Definition:
gpf_apply_fun (fn, adp, acp, sn, s, n, x)
= mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_modifiers (sbv, e, c, s, n, x)
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_selectors (e, c, s, n, x)
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_adp $(e,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_each $(id,\ vs,\ sbv,\ e,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_some $(id,\ vs,\ e,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_all $(id,\ vs,\ e,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_element_type $(e,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

DEFINITION:
gpf_element_list $(e,\ c,\ s,\ n,\ x)$
$=$ mutual-gp-gp_cond-gp_case_body-gp_locals-gp_procedure_body-gp_procedure_call-gp_parg_list-gp_parg-gpf-

```
; *********************
;  The Meta Function P
; *********************
```

DEFINITION:

meta_p $(m,\ c,\ s,\ n,\ x)$

$=$ **let** *ptm* **be** pt $(m,$ `'statement`$)$,
    ptx **be** pt $(x,$ `'program_description`$)$
  in
  if $ptm =$ **nil** **then** marked (`'statement_syntax_error`, **nil**)
  elseif $ptx =$ **nil**
  then marked (`'program_description_syntax_error`, **nil**)
  else gp $(ptm,\ c,\ s,\ n,\ ptx)$ **endif** **endlet**

Index