© Copyright 1984

Ada Project
Courant Institute
New York University
251 Mercer Street,
New York, NY 10012

A D A S E M†

STATIC SEMANTICS FOR

A d a®

This is a listing of the NYU Ada/Ed Compiler, Version 1.4, validated June 28th, 1984. It may not be reproduced in machine readable form without the written consent of the NYU Ada Project.

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

---

† This work was supported by U.S. Army contract number DAAB07-82-K-J196 (CORADCOM - Fort Monmouth) and by the Ada Joint Program Office.

Table of Contents

A D A S E M 1

DOCUMENTATION OF THE MODULE 2
General structure of the semantic actions. 2
Symbol table organization. 5
Contents of the symbol table maps 6
Object entities 7
Type entities 8
Subprograms and operators. 10
Packages and tasks. 11
Generic entities. 12

UTILITY MACROS FOR SEMANTIC MODULES. 12
Global macros for the use of all modules: 12
Macros for manipulation of the AST. 13
Macros for accessing type information. 13
Macros for scope manipulation. 15
Miscellaneous type predicates. 15

GLOBAL DECLARATIONS and DEFINITIONS 16
Constants 16
Variables 17

Representation specifications 18

proc INIT_SEM : primary initialization procedure 19

ADASEM : Grammar rules for semantic processing 26
Chapter 2. Lexical elements 27
Chapter 3. Declarations and types 28
Chapter 4. Names and expressions 32
Chapter 5. Statements 35
Chapter 6. Subprograms 37
Chapter 7. Packages 38
Chapter 8. Visibility rules 39
Chapter 9. Tasks 40
Chapter 10. Program structure... 42
Chapter 11. Exceptions 43
Chapter 12. Generics 44
Chapter 13. Representation specs... 45

UTILITIES 47
symbtab_enter 47
new_arith_ops 48
ADA_SEMANTICS 48

end_sem 49
warning 49
errmsg 49

2. Lexical elements 50
process_pragma 50

3. Declarations and types. 54
obj_decl 54
const_decl 55
const_redecl 57
set_type_mark 58
check_init 59
is_deferred_constant 60
number_decl 61
type_decl 61
find_type_name 62
build_type 64
ais_type 65
subtype_decl 66
make_subtype 67
derived_type 69
is_derived_type 72
derive_subprograms 72
derive1_subprogram 73
is_derived_subprogram 74
new_enum_type 74
new_integer_type 75
new_floating_type 75
new_fixed_type 77
real_bound 78
constrain_scalar 78
apply_range 81
array_typedef 81
new_array_type 82
new_unconstrained_array 83
new_constrained_array 83
anonymous_array 84
constrain_array 84
make_index 85
constrain_index 87
record_decl 88
is_record 89
process_discr 89
discr_decl 90
discr_redecl 91
conformance_error 92
comp_decl 92

constrain_record 93
check_discriminant 95
variant_decl 96
incomplete_decl 97
promote_subtype 98
subtype_expr 98
Miscellanous type predicates. 99
is_type 99
is_character_type 99
is_discrete_type 100
is_numeric 100
is_scalar_type 100
is_incomplete_type 101
is_unconstrained 101
can_constrain 101
base type 101
new_type_list 102
named_type 103
anonymous_type 103

4. Names and Expressions 104
check_type 104
constraint_kind 106
apply_constraint 107
expr_type 107
find_check_type 107
- RESOLVE1 - 108
-RESOLVE2- 118
-RESOLVE2- expression 118
-RESOLVE2- allocator 123
-RESOLVE2- attribute 124
-RESOLVE2- conversion 127
-RESOLVE2- subtype 129
-RESOLVE2- constant fold and note type of expression 131
result_types 131
disambiguate 133
compatible_op 134
remove_conversions 135
valid_op_types 136
valid_arith_types 138
intersect_types 142
valid_arg_list 142
complete_arg_list 143
order_arg_list 144
complete_op_expr 145
specialize 150
literal_expression 150
bind_arg 154

valid_array_expn 156
complete_array_expn 157
valid_selected_expn 158
complete_selected_expn 159
is_component 160
complete_aggregate 161
complete_a_aggregate 161
complete_component 163
complete_string_literal 164
complete_r_aggregate 165
valid_task_name 170
complete_task_name 170
is_task_type 171
valid_attribute 171
res2_check 177
compatible_types 178
type_error 179
pass1_error 181
full_type_name 181

5. Statements. 182
assign_statement 182
is_variable 185
case_statement 186
is_static_subtype 191
new_block 192
block 192
end_function 193
has_return 194
loop_statement 194
iter_var 194
exit_statement 195
return_statement 196
label_decl 197
lab_init 197
lab_end 197

6. Subprograms 198
subprog_decl 198
subprog_spec 199
subprog_body 200
build_signature 202
get_formals 202
process_formals 203
reprocess_formals 204
normalize 205
conform 206
statement_name 207

proc_or_entry 208
desig_to_name 209
chain_overloads 210
new_over_spec 213
same_signature 213
same_type 214

7. Packages 214
package_head 214
module_body_id 215
module_body 217
private_decl 220
fully_declared 220
check_fully_declared2 221
is_private 221
is_limited_type 221
private_kind 222
is_fully_private 223
check_priv_decl 223
private_ancestor 224
end private 226
end_specs 227
check_incomplete_decls 228
needs_body 229
get_specs 229

8. Visibility rules 230
find_old 230
name_error 231
check_old 232
collect_inported_names 234
array_or_call 235
parameterless_callable 236
index_or_slice 237
find_qual 237
find_agg_types 240
find_access_types 240
find_new 241
new_agg_or_access 242
original_name 243
renames 243
renamed 247
check_modes 248
op_matches_spec 249
renamed_attribute 249
renamed_entry 250
entry_as_procedure 251
check_nat_type 251

newscope 253
popscope 253
newmod 253
use_clause 254

9. Tasks. 255
task_spec 255
accept_statement 257
entry_decl 259
entry_family_decl 259
entry_call 260
check_entry_call 261
find_entry_name 262
select_statement 264
terminate_statement 265
abort_statement 265

10. Program structure and Compilation issues. 266
init_compunit 267
new_compunit 267
compunit 268
unit_symbtab 269
predef_scopes 270
save_subprog_info 270
save_spec_info 271
save_body_info 271
establish_context 271
elaborate_pragma 274
stub_head 275
save_stub 275
separate_context 276
check_separate 277
retrieve_env 278

11. Exceptions 278
except_decl 278
exception_part 279
exception_handler 280
raise_statement 280

12. Generic Program units. 281
generic_subprog_spec 281
generic_subprog_body 283
generic_pack_spec 285
generic_obj_decl 286
generic_type_decl 288
is_generic_type 289
generic_priv_decl 289

check_generic_usage 290
generic_subp_decl 290
bind_names 291
subprog_instance 291
package_instance 293
instantiate_generics 298
instantiate_object 302
is_discriminant_dependent 303
instantiate_type 306
valid_scalar_instance 307
check_actual_conatraint 308
valid_priv_instance 309
valid_access_instance 310
valid_array_instance 311
is_valid_disc_instance 312
get_array_info 313
generic_subprogram_instance 313
replace 315
replace_prefix 316

13. Representation Clauses 316
check_repr_tyep 317
length_clause 317
enum_rep_clause 318
rec_rep_clause 320

Appendix A. Predefined language attributes. 321
check_range_attribute 322

Constant folding 322
const_val 322
eval_static 323
check_overflow 350
check_null_aggregate 351
make_ghost_out_of 352

Print utilities 352
print_xref 352
sort 355
make_lines 355
format_numbers 356
format_subprog 357

DROP SEMANTICS MACROS. 358

ADASEM

SEMANTIC ACTIONS FOR

A D A

Ada Project
Courant Institute
New York University
251 Mercer Street
New York, NY 10012

module ada - semantics;

DOCUMENTATION OF THE MODULE

General structure of the semantic actions.

```
24 --
25 --  The semantic actions described in this module annotate, validate
26 --  and modify the parse tree, enforcing all static semantic checks
27 --  required by the language, and convert the parse tree into bindable
28 --  code.
29 --
30 --  These semantic actions are invoked as various tree fragments are
31 --  produced. In this module these actions are organized according to
32 --  the corresponding chapter of the RM:
33 --
34 --  c2) Lexical elements. The only item relevant to semantic processing
35 --      here is the processing of pragmas.
36 --
37 --  c3) Declaration processing for objects, constants, types, subtypes,
38 --      discriminants. A number of useful type predicates are also in-
39 --      troduced in this chapter.
40 --
41 --  c4) Expression processing: type checking and overload resolution
42 --      are intimately linked, and performed together in this chapter.
43 --
44 --  c5) Statement processing: all statement forms are processed here.
45 --      The case statement handler of this chapter is also used by the
46 --      record declaration routines to process variant parts.
47 --
48 --  c6) Subprogram processing: non-generic subprograms are handled here
49 --     together with the chaining of overloaded constructs and the
50 --      declarations of formal parameters.
51 --
52 --  c7) Package processing: the largest part of this chapter is devoted
53 --      to the enforcement of rules regarding private and limited types
54 --
55 --  c8) Visibility rules: name resolution for simple names, indexed and
56 --      selected components, slices, and attributes is perfomed here.
57 --      New entities are inserted in the symbol table, and scope
58 --      manipulation (stacking and unstacking, definition of new symbol
59 --      tables, rules for unique name formations, etc.) are performed.
60 --      Overloaded constructs are recognized and marked in the tree,
61 --      for subsequent type checking and resolution. Renaming declara-
62 --      tions are processed. (also used for some generic matching).
63 --
```

```
64 --   c9) Task processing: task declarations, resolution of entry names,
65 --       select statements, etc.
66 --
67 --   c10)Compilation issues: symbol tables for separate units and for
68 --       stubs are merged, library files are updated. Enforcement of
69 --       order-of-compilation rules takes place in the binder module.
70 --
71 --   c11)Exception declarations, exception handlers.
72 --
73 --   c12)Generic processing: declarations of generic units, instantia-
74 --       tion, and construction of the bodies and symbol tables for
75 --       instantiated objects.
76 --
77 --   c13)Representation clauses are checked for semantic correctness,
78 --       but otherwise ignored by this implementation.
79 --
80 --   c14)The predefined I/O packages are given in module PREDEF.ADA.
81 --       An implementation-defined pragma IO_INTERFACE establishes the
82 --       link to the implementation of I/O primitives. The Ada I/O is
83 --       otherwise not visible in the semantic module.
84 --
85 --   Finally, a constant folding procedure recognizes and evaluates
86 --   static expressions,and performs some modest constant folding.
87 --
88 --   Input to the semantic actions is a parse tree, in the form of a
89 --   labelled tree, whose nodes are atoms. Two maps are defined on the
90 --   nodes:
91 --     AST(n) is the tuple of descendents of node n, or a terminal
92 --     string.
93 --     SPANS(n) gives the coordinates in the source program, of the
94 --     nonterminal denoted by node n.
95 --
96 --   The AST for a non-terminal is a tuple whose first component is an
97 --   operator that establishes the nature of that node, eg. 'obj_decl',
98 --   'formal', 'package_spec', etc. The routine ADASEM, which is the
99 --   driver for the semantic actions, performs a (mostly) top-down
100 --  traversal of the AST, interpreting each node. Details on the struc-
101 --  ture of each node can be found in ADASEM, and in the REDUCE routine
102 --  in the parser module.
103 --
104 --  Ouput of the semantic actions is another tree, designated AIS code,
105 --  together with the symbol table information needed to support sepa-
106 --  rate compilation. The AIS code is a tree represented as a SETL tuple
107 --  whose structure is otherwise similar to that of the AST, except that
108 --  name resolution has replaced all source identifiers by unique names,
109 --  some constant folding has taken place, and statement numbers have
110 --  been inserted for tracing purposes.
111 --
112 -- All semantic processing revolves around a single common data-structure
```

```
113 -- namely the symbol table. We proceed to describe its organization.
114
```

Symbol table organization.

```
116
117 --   The symbol table holds the attributes of all objects appearing
118 -- in a program. Different classes of objects are described by different
119 -- attributes, and it is convenient to describe each of them by a sepa-
120 -- rate map. The domains of these maps overlap, but do not always coin-
121 -- cide. Six such maps are used : NATURE, TYPE_OF, SIGNATURE, OVERLOADS,
122 -- SCOPE_OF, and ALIAS. The term 'symbol table' refers collectively to
123 -- these six, and the global macros SYMBTAB, SYMBTABF denote tuples of
124 -- their values.
125 --
126 --   Each object in the program is given a unique name, which denotes
127 -- its symbol table slot. The set of these unique names is the domain of
128 -- the five symbol table maps. The semantic actions that deal
129 -- with name resolution have as their purpose to determine the unique
130 -- name, or symbol table slot, which corresponds to a given identifier
131 -- at a given point in the program. Most subsequent semantic actions deal
132 -- with that unique name, and not with the original identifiers. Confor-
133 -- mance checks, and named associations in various parameter lists, make
134 -- it useful to refer to the original names, or source identifiers, of
135 -- unique names.
136 --
137 --   In order to relate program identifiers to unique names, a global
138 -- map called DECLARED, is used throughout. Every lexical scope corres-
139 -- ponds to an entry in DECLARED. In turn, this entry is a map from
140 -- source identifiers to their unique names in that scope.
141 --
142 --   DECLARED : scope-name → (map : identifier → unique-name )
143 --
144 -- The scope manipulation routines and the name resolution routines
145 -- revolve around DECLARED.
146 --
147 --   Internally, these unique names are character strings. In order to
148 -- retain a modicum of readability for the intermediate code, these
149 -- strings are constructed using the original identifiers to which they
150 -- correspond, and adding various prefixes and suffixes in order to
151 -- insure
152 -- their uniqueness. These affixes are chosen to reflect the lexical
153 -- placement of the original identifiers, and as a result these construc-
154 -- ted unique names carry some information encoded within themselves, and
155 -- do not act simply as pointers. This (probably unwise) hybrid purpose
156 -- should disappear when a more conventional, lower-level organization
157 -- for the symbol table is implemented.
158 --
159 --   Compiler-generated names (for constants, scopes and anonymous types)
```

```
160 -- also have symbol-table entries. Their unique names are internally
161 -- generated following similar rules. Their natures are described below.
162 --

        Contents of the symbol table maps

164 -- In what follows, -name- denotes a unique name.
165 -- a) NATURE(name) is a string which indicates the kind of entry which
166 --    name been made for -name-. It is thus a discriminant for the rest
167 --    of the symbol table fields. The following are the possible values
168 --    of the nature of program entities, and the chapters in which
169 --    they are introduced:
170 --
171 -- Predefined entitities (the package STANDARD) include the following:
172 --    'op'          'attribute'
173 --
174 -- Declarations (Chapter 3) deal with the following:
175 --    'obj'     'constant'
176 --    'type'    'subtype'  'array'     'record'   'enum'    'literal'
177 --    'access'  'aggregate'
178 --
179 -- Blocks, loops (Chapter 5) and exception handlers are:
180 --    'block'
181 -- statements may be labelled by an entity of nature
182 --    'label'
183 --
184 -- Subprograms and their formal parameters (chapter 6) make use of:
185 --    'procedure_spec'     'function_spec'     'procedure'  'function'
186 --    'in'      'inout'     'out'
187 --
188 -- Packages (Chapter 7) introduce:
189 --    'package_spec'   'package'   'private_part'
190 --
191 -- Tasks and their entries (Chapter 9) deal with:
192 --    'task_type'     'task_type_spec'    'task_obj'     'task_obj_spec'
193 --    'entry'         'entry_family'      'entry_former'
194
195 -- Generic units (Chapter 12) involve:
196 --    'generic_procedure'  'generic_function'   'generic_package'
197 --    and the corresponding specs.
198 --
199 --    The content and meaning of the other symbol table maps is
200 --    determined by the nature of the object. Before giving a detailed,
201 --    nature-by nature description, let us sketch their use :
202 --
203 -- b) TYPE_OF(name) specifies the ADA type of an object;the value of this
204 --    field is therefore the unique name of an entity which designates a
205 --    type.
206 --
```

```
207 -- c) SIGNATURE(name) is used primarily to hold information about the
208 --    formal parameters of a subprogram. It is used in miscellaneous
209 --    fashions for array types, for subtypes, for packages, and for
210 --    generic objects. It is always a tuple.
211 --
212 -- d) OVERLOADS(name) is a SETL set, whose elements are the unique names
213 --    of the objects whose program identifier is the same as that of name
214 --    and which are defined in the same scope. This map is defined for
215 --    overloadable objects in the language : procedures, functions,
216 --    literals, operators.
217 --    For packages, the OVERLOADS field store the private declarations of
218 --    the package. These declarations are swapped in and out of the symbol
219 --    table for the body, when the body is encountered/exited.
220 --
221 -- e) SCOPE_OF(name) is the unique name of the scope in which name is
222 --    declared. It is declared for all symbol table entries. If -id- is
223 --    the source identifier of -name-, then the following holds:
224 --
225 --       DECLARED(SCOPE_OF(name))(id) = name
226 --
227 -- f) ALIAS(name) is used for derived subprograms and for renamed subpro-
228 --    grams. It is a unique name. Such subprograms do not have a body; a
229 --    call to them is transformed into a call to their parent subprogram.
230 --
231 --    For array types, record types, access types,  generic units, etc.
232 --    additional information must be kept. This is done by miscellaneous
233 --    use of the SIGNATURE and OVERLOADS maps. These maps are accessed by
234 --    means of macros described below.
235 --
```

Object entities

```
237 -- Let us now flesh out this sketch.
238 --
239 -- 'obj' :  Denotes a symbol table slot for a program variable.
240 --          The corresponding entry in TYPE_OF is always a unique name,
241 --          namely that of a user-defined or internally generated type.
242 --          SIGNATURE and OVERLOADS are not used for an obj.
243 --
244 -- 'constant' : Denotes a symbol table slot for a user-defined constant
245 --          and for a number declaration.
246 --          The TYPE_OF entry is the specified type for it.
247 --          The SIGNATURE entry holds the SETL representation of the value
248 --          of this constant or the AIS code fragment that evaluates it,
249 --           the case the constant is not static.
250 --
251 -- 'literal'   : an entry of this nature denotes an enumeration literal.
252 --            The SIGNATURE of the literal is the same as that of a parame-
253 --           terless function, i.e. [ ].
```

```
254 --
255 --
256 -- 'attribute' : An entry of this nature denotes a predefined attribute
257 --          of the language.
258 --          Many attributes are generic : they apply to classes of types.
259 --          The TYPE_OF entry for an attribute is either a predefined type,
260 --          (INTEGER or STRING) or the marker 'overloaded'. This indicates
261 --          that the type returned by the attribute depends on the type of
262 --          its arguments.
263 --          The signature and overloads entries are unused.
264 --
```

Type entities

```
266 --
267 --
268 -- 'type' :    This denotes an entry for a new, or derived type
269 --          from one of the numeric types 'INTEGER', 'FLOAT',
270 --          or '$FIXED', or from another numeric type, or
271 --          the error type 'any'.
272 --          The TYPE_OF entry gives the unique name of the type from
273 --          which the current one is derived.
274 --          The SIGNATURE entry gives the constraint (see 'subtype'
275 --          below).
276 --          Private and incomplete type declarations produce type
277 --          entries whose TYPE_OF field is 'private','limited private'
278 --          or 'incomplete' respectively.
279 --          The map 'misc_type_attributes' collects various useful predi-
280 --          cates on types, in particular composite and generic types.
281 --          This map overlays the OVERLOADS map.
282 --
283 -- 'subtype' : This denotes an entry for a declared subtype, or for a
284 --          range descriptor which has been promoted to a subtype.
285 --          The TYPE_OF entry is the unique name of the type of which
286 --          the current one is a subtype, i.e. its base type.
287 --          The SIGNATURE entry holds the constraint which defines the
288 --          subtype. For scalar types, the constraint has one of the
289 --          following formats :
290 --
291 --              ['range' low, high]          for discrete subtypes.
292 --
293 --              ['digits', low, high, digits]    for floating subtypes.
294 --
295 --              ['delta', low, high, delta]      for fixed subtypes.
296 --
297 --          In each case, -low- and -high- are the bounds of the subytpe.
298 --          digits is a SETL integer, delta is a rational (a pair of SETL
299 --          integers. See module adaarith for details).
300 --          Both of these have their standard Ada meaning .
```

```
301 --
302 --          for record subtypes, the constraint as the format :
303 --                  ['discr', list of discriminant values]
304 --
305 --          for array subtypes, see below.
306 --
307 -- 'array' : This denotes an entry for an unconstrained array type.
308 --          The TYPE-OF an array entry is itself.
309 --          The SIGNATURE of an array entry is the pair :
310 --                  [ index-types, component-type]
311 --
312 --          where index_types is a tuple of type names.
313 --
314 --          an array subtype denotes and entry for a constrained array.
315 --          Its nature is 'subtype', its type and signature are those of
316 --          its (unconstrained) base type.
317 --
318 -- 'record' : Denotes a record type.
319 --          The TYPE_OF a record entry is the type name itself.
320 --
321 --          The SIGNATURE for a record is a triple formatted as
322 --          follows:
323 --
324 --          [ [invariant_part, variant_part], discriminant_list ,
325 --                                                  declared_map ]
326 --
327 --          the discriminant list is [ ] for a record without discriminant
328 --
329 --          The declared_map is the map from selector names
330 --          to their unique names.
331 --
332 -- 'enum'   : Denotes an enumeration type.
333 --          The TYPE_OF the enumeration is the parent type, if derived,
334 --          or the typename itself for BOOLEAN, CHARACTER, and non-derived
335 --          enumerations.
336 --          The SIGNATURE of the enumeration type is the range for
337 --          the enumeration type in the format:
338 --
339 --                  ['range', low, high]
340 --
341 --          where low and high are the integer values for the first
342 --          and last enumeration literals of the type.
343 --            The OVERLOADS entry is a map, from the literals to their
344 --          POS in the type. By default the range of this map is dense
345 --          and 0-origined, but it can be modified by a representation
346 --          clause. This map is accessed by the macro literal_map.
347 --
348 -- 'access' : denotes an access type.
349 --          The type_of map of an access type is itself. The macro
```

```
350 --          designated_type(access_type) retrieves the unique name of
351 --          the type of designated objects of the access type.
352 --          this macro is also available on  subtypes of access types.
353 --
354 --
355 -- 'aggregate' : Every array and record declaration introduces an aggre-
356 --                   gate into the environment, i.e. one possible meaning
357 --          for any aggregate expression appearing in the code. In order
358 --          to disambiguate such an expression, it is convenient to view
359 --          aggregates as overloaded constructs. Each aggregate entry
360 --          is uniquely characterized by its type (held in TYPE_OF).
361 --
```

Subprograms and operators.

```
363 -- 'procedure' : Denotes a symbol table slot for a procedure subprogram.
364 --          Its TYPE_OF field holds the marker 'none'.
365 --          The SIGNATURE describes the formal parameters of the procedure.
366 --          It is a tuple, each of whose components has the format :
367 --
368 --                   [name, mode, type, default]
369 --
370 --          -name- is the unique name of the formal parameter.
371 --          -mode- is one of the strings : 'in', 'out','inout' .
372 --
373 --          -type- is the type of the formal parameter. It is always a
374 --          unique name.
375 --
376 --          -default- is the type-checked and resolved expression which is
377 --          to be  used as an actual  argument, when none  is  provided in
378 --          the call.
379 --
380 --           OVERLOADS for a procedure is the set of procedures with the
381 --          same name (source identifier, that is) which appear in the
382 --          current lexical scope, or in enclosing scopes, and which are
383 --          not hidden by the current entry. This set is built when the
384 --          procedure specification is processed (See procedure : chain-
385 --          overloads in the semantic actions) and is central to the
386 --          overload resolution mechanism.
387 --          OVERLOADS(id) always includes -id- itself.
388 --        If the procedure is the only visible instance of the name,
389 --          (i.e. it overloads no other identifier) then the OVERLOADS
390 --          entry is the singleton set containing its own name.
391 --
392 --
393 -- 'function'  : identical to 'procedure'. The TYPE_OF a function is the
394 --          type which it returns.
395 --
396 -- 'procedure_spec', 'function_spec' : ditto. An entry is given this
```

```
397 --              nature after the corresponding subprogram specification
398 --              is processed. When the subprogram body is encountered
399 --          the specification is examined again, and if it coincides with
400 --          an existing symbol table slot, the _spec tag is removed.
401 --
402 -- 'op' :  An entry with this nature denotes a predefined operator in the
403 --          language.
404 --          The TYPE_OF entry for an operator is usually a generic marker
405 --          which indicates the family of types to which this operator can
406 --          apply. This marker obviates the need to introduce a separate
407 --          operator for each type derived from the predefined types. These
408 --          generic type marks are also given a symbol table slot. The
409 --          following are used :
410 --
411 --          1) 'universal_integer' : for all arithmetic operators which
412 --          apply to integers.
413 --          2) 'universal_real'    : ditto for floating and fixed types.
414 --          3) 'array_type'    : used for the (idiosyncratic) concatenation
415 --              operator.
416 --          4) 'order_type'    : for comparison operators.
417 --           5) 'boolean type' : for boolean operators. This simplifies the
418 --              handling of arrays of booleans and derived booleans.
419 --          6) 'string_type'   : for string literals, which overload any
420 --              array of characters.
421 --          7) 'character_type' : the component type of 'string_type'
422 --           8) 'composite_type': for aggregates
423 --
424 --          The signature of an operator follows the same format as the
425 --          one for a subprogram. The formal parameter names LEFT and
426 --          RIGHT are not stored explicitly, but recognized by the proc.
427 --          order_arg_list in those rare cases when an operator is used in
428 --          functional notation with named associations.
429 --
430 --          The OVERLOADS entry of an operator has the same meaning as for
431 --          subprograms. Given that operators are entered into the symbol
432 --          table during initialization, at the outermost scope, most ope-
433 --          rators in fact do not overload anything. For arithmetic ope-
434 --          rators however, we give different unique names to the integer
435 --          and floating point version of them.
436
```

Packages and tasks.

```
438 -- 'package', 'package_spec' : these entries have no defined type nor
439 --          signature. Access to entities declared in a package is via
440 --          its declared map. Access to entities in the specification is
441 --          via the mapping VISIBLE. The domain of visible(p) is a sub-
442 --          set of that of declared(p).
443 --          The private declarations of the package are stored in a
```

```
444 --        separate structure, private_decls(p). These private decla-
445 --        rations are installed in the global symbol table when the
446 --        body of the package is compiled, and removed afterwards, so
447 --        that only the visible attributes remain accessible outside
448 --        of the package.
449 --
450 -- 'private_part': for technical reasons, it is convenient to regard the
451 --        private part of a package specification as a scope in its own
452 --        right. This simplifies the segregation of private declarations
453 --        from visible ones, and the detection of sundry semantic errrors
454 --
455 -- 'task_obj', 'task_type': Denotes a task. Access to its entries
456 --           is obtained by means of the -declared- map for either.
457 --
458 -- 'entry_former' : denotes the type of an entry object. Calls to entries
459 --           (rendez-vous) have the same syntax as procedure calls, and
460 --           named and default parameters are also present. The SIGNATURE
461 --           of the entry type is the list of formals for the entry, and
462 --           has the same format as that of a procedure.
463 --           As for procedures, the TYPE_OF an entry is the marker 'none'.
464 --
465 -- 'entry'  denotes an entry object. Its type_of field is an entry
466 --         former. Its signature is the formals list, and is identical
467 --           to that of a subprogram. Ditto for OVERLOADS.
468 --
469 -- 'entry_family' : denotes an entry family object. This is a non-over-
470 --         loadable entity, constructed as an array of entries. Its type
471 --         is an anonymous array type whose component is an entry-former
472
```

Generic entities.

```
474 --
475 -- Generic objects have the same organization as their non-generic
476 -- counterparts. The SIGNATURE of a generic entity has, in addition to
477 -- its non-generic purpose, the further role of holding the list of
478 -- generic parameters and the body of the generic object.
```

UTILITY MACROS FOR SEMANTIC MODULES.

Global macros for the use of all modules:

```
481
482 macro find;    assert exists          endm;
483
484 macro top(x); x(#x)                   endm;
485
```

Macros for manipulation of the AST.

```
487
488 macro node_kind(node); (AST(node)(1))   endm ;
489
490 macro is_empty(node);  (node=[ ])       endm ;
491
492
493 macro SYMBTAB(name);
494     [NATURE(name),TYPE_OF(name),SIGNATURE(name),OVERLOADS(name)]
495 endm;
496
497 -- For library and stub manipulation, the scope_of map must also
498 -- be saved and restored. The following macro is usedfor that purpose.
499
500 macro SYMBTABF(name) ;
501     [NATURE(name), TYPE_OF(name), SIGNATURE(name), OVERLOADS(name),
502      SCOPE_OF(name), ALIAS(name) ]
503 endm ;
504
505 -- The following is a lexical macro which converts a double-quoted
506 -- character which appears as a designator, into a one-character
507 -- string. This distinction is imposed by the need to differentiate
508 -- the literal  a  from the character  "a"  when they appear in an
509 -- enumeration literal, but nowhere else.
510
511 macro char_to_name(desig) ;
512 (if # desig = 3 and desig(1) = '"' and desig(3) = '"' then
513       desig(2) else desig end)
514 endm ;
515
```

Macros for accessing type information.

```
517
518 macro root_type(type_mark) ; ALIAS(type_mark)          endm ; --
519
520 macro index_types(array_type) ;
521         signature(array_type)(1)
522 endm ;
523
524 macro component_type(array_type) ;
525         signature(array_type)(2)
526 endm ;
527
528 macro no_dimensions(array_type) ;
529         (#index_types(array_type))
530 endm ;
531
532 macro index_type(array_type) ;
533         (index_types(array_type)(1))
534 endm ;
535
536 macro literal_map(enumeration) ;
537         overloads(enumeration)
538 endm ;
539
540 macro record_declarations(record); signature(record)          endm ;
541
542 macro all_components(record) ;    signature(record)(1)     endm ;
543
544 macro discriminant_list(record) ; signature(record)(2)          endm ;
545
546 macro invariant_part(record) ;    all_components(record)(1) endm ;
547
548 macro variant_part(record) ;    all_components(record)(2) endm ;
549
550 macro declared_components(record); signature(record)(3)     endm ;
551
552 macro discr_decl_tree(record);    signature(record)(4)     endm ;
553
554 macro has_discriminants(record) ;
555     (discriminant_list(record) ≠ [ ])                    endm ;
556
557 macro designated_type(access_type) ;
558                         (signature(access_type))         endm ;
559
560 macro is_range_attribute(e) ;
561     (is_tuple(e) and e(1) = "" and e(2) = 'RANGE')          endm ;
562
```

```
563 macro label_status(lab) ; signature(lab)                endm ;
564
565 macro default_expr(nam) ; signature(nam)                endm ;
566
567 macro formal_decl_tree(sig) ; sig(1)(5)                       endm ;
```

Macros for scope manipulation.

```
569
570 --
571 macro init_scope_info ;
572    ['STANDARD#0', '', ['STANDARD#0', 'UNMENTIONABLE#0'], [ ],
573     ['ASCII'], ''
574    ]
575 endm;
576
577
578 macro is_comp_unit ; (#scope_st = 0)      endm ;
579
580 macro scope_info ;
581    [scope_name, prefix, open_scopes, used_mods, vis_mods, suffix]
582 endm ;
583
```

Miscellaneous type predicates.

```
585 macro is_access(name) ;
586    (nature(root_type(name)) = 'access')              endm ;
587
588 macro is_identifier(name) ; is_string(name)                  endm ;
589
590 macro is_literal(name) ; nature(name) = 'literal'       endm ;
591
592 macro is_overloaded(name) ; (is_set(name))                   endm ;
593
594 macro is_constant(name) ;
595    (is_identifier(name) and nature(name) = 'constant')
596 endm ;
597
598 macro is_proc(proc_name) ;
599    nature(proc_name) ∈
600     {'procedure', 'function','generic_procedure', 'generic_function'}
601 endm;
602
603
604 macro is_anonymous(typ) ; (original_name(typ) = '') endm ;
605
606 --
607 macro is_first_named_subtype(typ);  (is_anonymous(base_type(typ))) endm;
```

```
608
609 macro is_array(typ) ; (nature(root_type(typ)) ='array')  endm ;
610
611
612 --
613 macro is_formal(id) ; (nature(id) ∈ ['in', 'out', 'inout']) endm ;
614
615 macro is_generic(scope ; nat) ;
616  ((nat:=nature(scope)) ≠ om and match(nat,'generic_') ≠ om)
617 endm ;
618
619 macro misc_type_attributes(type_mark) ;
620                                  overloads(type_mark)    endm;
621
622 macro private_dependents(type_mark) ;                                  --
623                  misc_type_attributes(type_mark)                 endm;
624
625 macro private_decls(package);        overloads(package)        endm;
626
627 --
628 macro use_declarations(package) ;
629         signature(declared(package)('$used'))            endm;
630
631 macro is_anonymous_task(name) ;
632         (is_task_type(name) and 'task_type:' ∈ original_name(name))
633 endm ;
634
635 macro task_of(entry_type); type_of(entry_type)          endm;
636
637 macro is_integer_type(x) ;
638 (x ∈ {'INTEGER', 'SHORT_INTEGER', 'LONG_INTEGER', 'universal_integer'})
639 endm ;
640
641 -- The following macro is used by cross-reference making statements.
642 -- spans(node) is either [l_span] or [l_span, r_span] ,
643 -- l_span := [line_no, col1, col2].
644
645 macro TO_XREF(name) ;
646 if spans ≠ om and spans(current_node) ≠ om then
647    XREF(name) := ( XREF(name) ? {} ) +
648                  { spans(current_node)(1)(1) } ;
649 end if
650 endm;
651
```

GLOBAL DECLARATIONS and DEFINITIONS

Constants

```
654 const
655    OP_DESIGNATORS = {'and', 'or', 'xor', 'not', 'mod', 'rem', 'abs',
656                      '=',  '≠', '<=',  '<',  '>=',  '>',
657                      '+',  '-', '&',   '*',  '/',   '**'} ;
658
659 const comparison_ops = {'<','<=','>','>=','=','≠'} ;
660
661 const univ_types = {'universal_integer', 'universal_real'} ;
662 const priv_types = {'private', 'limited private'} ;
663 const incp_types = {'private','limited private','limited','incomplete'};
664 const base_attributes = {'BASE', 'POS', 'PRED', 'SUCC', 'VAL', 'VALUE'};
665 --
666 const qualifiers = {'qual_range',  'qual_length',  'qual_discr',
667                     'qual_arange', 'qual_alength', 'qual_adiscr' };
668
```

Variables

```
670 var
671        BASE_DECLARED, -- declared map for standard environment
672        DECLARED, -- maps scope names to map of names declared
673                  -- in scope to their unique names
674        PROD_SPANS,       -- pair consisting of left and right token
675                  -- locations of AST node
676
677        ALIAS,              -- unique name → parent subprogram or root type
678        NATURE,             -- unique name → nature of entity
679        OVERLOADS,          -- unique name → set of unique names in current
680                  -- scope, that have the  same source identifier.
681        SCOPE_OF,  -- unique name → scope of entity
682        SIGNATURE,-- unique name → tuple of attributes
683        TYPE_OF,   -- unique name → type of entity
684       XREF,        -- unique name → line numbers that use entity
685
686        NEWTYPES, -- list of anonymous types whose declaration is
687                  -- emitted in front of current declaration.
688        GOTO_STMTS,       -- set of GOTO's encountered in unit
689        LAB_SCOPE_STK,    -- stack of label scopes for prog units
690        LAB_SCOPE_MAP,    -- ancestor map for label scopes
691        LAB_SEEN,   -- set of labels in procedure .
```

```
692         STMLABS,     -- set of statement labels
693
694         ALL_VIS,     -- Modules whose visibility is required
695         CONTEXT,     -- Tuple of with & use clauses
696         ENCLOSING_SCOPE,-- Tuple representation of subunit ancestor name.
697         GENERIC_CLAUSE, -- Contents of current generic declarations.
698         INSTANTIATION_CODE,
699                      -- To collect checks on actual parameters.
700         OPEN_SCOPES,     -- nest of currently open scopes, from inner
701                      -- to outer.  Outermost is STANDARD.
702         PROC_STACK,  -- Stack of statically nested subprograms
703         PREFIX,          -- prefix for forming unique name
704         SUFFIX,          -- suffix for forming unique name
705         SCOPE_NAME,      -- unique name for each scope
706         SCOPE_ST,    -- stack of lexical scopes
707         SEPARATE,    -- separate unit switch
708         UNIT_NAME,       -- compilation unit information
709         UNIT_TYPE,   -- type of compilation unit
710         USED_MODS,       -- packages appearing in a use clause
711       VERSIONS_USED,     -- Compilation dates of units used by unit.
712         VIS_MODS,    -- list of package names visible in this unit
713         VISIBLE;     -- declared map for visible parts of packages
714
715
716 var all_imported_names, -- for error reporting when name is hidden.
717
718     comp_units,          -- Count of compilation units in compilation
719     noop_error,          -- Used during type checking
720
721     out_context,         -- Signals valid appearance of out parameter.
722     full_others,       -- To enforce the rules of 4.3.2(6) et al.
723     fold_context ;       -- To inhibit constant folding on conformance
724                          -- checks.
725
```

727

proc INIT_SEM : primary initialization procedure

```
729 -- This is the primary initialization procedure.  It is called
730 -- when the compiler is initialized.
731 --
732 proc INIT_SEM;
733
734   const
735     standard = 'STANDARD#0',
736
737     boolean_ops    = {'not','and', 'or', 'xor'},
738     short_circ_ops = { 'andthen', 'orelse'},
739     equal_ops      = {'=', '≠', 'in', 'notin'},
740     order_ops      = {'<', '<=', '>=', '>'},
741     arith_ops      = {'+u','-u','abs','+', '-', '*', '/'},
742     expon_ops      = {'**'},
743     mod_ops        = {'mod', 'rem'},
744     concat_ops     = {'&'},
745
746
747     misc_attributes = {'ADDRESS', 'IMAGE'},
748
749     u_integer_attributes = {
750       'AFT',          'COUNT',       'DIGITS',      'EMAX',
751        'FIRST_BIT',    'FORE',        'LAST_BIT',    'LENGTH',
752       'MACHINE_EMAX', 'MACHINE_EMIN', 'MACHINE_MANTISSA',
753       'MACHINE_RADIX', 'MANTISSA',    'POS',         'POSITION',
754       'SAFE_EMAX',    'SIZE',        'STORAGE_SIZE', 'WIDTH'
755                 },
756
757     overloaded_attributes = {
758         'BASE',         'FIRST',       'LAST',        'PRED',
759         'RANGE',        'SUCC',        'VAL',         'VALUE'
760                 },
761 --
762     float_attributes = {
763        'DELTA',        'EPSILON',   'LARGE',   'SMALL', 'SAFE_LARGE',
764       'SAFE_SMALL'
765                 },
766
767     boolean_attributes = {
768         'CONSTRAINED', 'MACHINE_OVERFLOWS', 'MACHINE_ROUNDS',
769         'CALLABLE',    'TERMINATED'
770                 },
771
```

```
772   predefined_types =
773         {'BOOLEAN', 'CHARACTER', 'DURATION', 'FLOAT', 'INTEGER',
774          '$FIXED', 'NATURAL',  'POSITIVE', 'STRING',
775         'universal_integer',   'universal_real',   'discrete_type',
776         'universal_fixed',     'string_type',      'array_type',
777         'any', 'none'} ,
778
779
780   exceptions = { 'CONSTRAINT_ERROR', 'NUMERIC_ERROR', 'PROGRAM_ERROR',
781                  'STORAGE_ERROR', 'TASKING_ERROR' };      --
782
783
784
785   attributes := misc_attributes + u_integer_attributes +
786       overloaded_attributes + boolean_attributes + float_attributes;
787
788
789   NATURE := {};
790   TYPE_OF := {};
791   SIGNATURE := {};
792   OVERLOADS := {};
793   SCOPE_OF  := {};
794   ALIAS := {} ;                                    --
795   XREF := {};
796   DECLARED := {};
797   SCOPE_ST:= [ ];
798   NEWTYPES:= [ ] ;
799   PROC_STACK := [ ] ;                                  --
800   NON_LOCAL_NAMES := {} ;                              --
801   STMLABS := {};
802   LAB_SCOPE_STK := [ ];
803   LAB_SCOPE_MAP := [ ];
804   LAB_SEEN          := [ ];
805   GOTO_STMTS    := [ ];
806   CURRENT_NODE := [ ];
807   UNIT_NAME  := ['','];
808   UNIT_TYPE  := '';
809   ALL_VIS    := {};
810   VERSIONS_USED := {};
811
812   GENERIC    := false;
813   RESTRICTED := false;
814   SEPARATE   := false;
815   VISIBLE    := {};
816
817   scope_info := init_scope_info ;
818   noop_error := false ;
819   out_context := false ;                           --
820   full_others := false;                 -- common case for assignment.
```

```
821     fold_context := true ;              -- Everywhere but for conformance
822
823     init_symbtab := {
824         ['INTEGER', ['type', 'INTEGER',
825          ['range', ['ivalue', ADA_MIN_INTEGER],
826                    ['ivalue', ADA_MAX_INTEGER] ]]],
827         ['LONG_INTEGER', ['type', 'INTEGER',
828          ['range', ['ivalue', ADA_MIN_INTEGER],
829                    ['ivalue', ADA_MAX_INTEGER] ]]],
830         ['SHORT_INTEGER', ['type', 'INTEGER',
831          ['range', ['ivalue', -2 ** 15],
832                    ['ivalue', 2 ** 15 -1] ]]],
833         ['FLOAT',   ['type', 'FLOAT',
834            ['digits', ['ivalue', ADA_MIN_REAL],
835                       ['ivalue', ADA_MAX_REAL],
836                       ['ivalue', ADA_REAL_DIGITS] ]]],
837         ['LONG_FLOAT',   ['type', 'FLOAT',
838            ['digits', ['ivalue', ADA_MIN_REAL],
839                       ['ivalue', ADA_MAX_REAL],
840                       ['ivalue', ADA_REAL_DIGITS] ]]],
841         ['$FIXED',  ['type', '$FIXED',
842            ['delta', ['ivalue',[ -1, 0]],
843                      ['ivalue',[  1, 0]],
844                      ['ivalue',[  0, 1]] ] ]],
845         ['universal_fixed',  ['type', 'universal_fixed',
846            ['delta', ['ivalue',[ -1, 0]],
847                      ['ivalue',[  1, 0]],
848                      ['ivalue',[  0, 1]] ] ]],
849         ['NATURAL',   ['subtype', 'INTEGER',
850            ['range', ['ivalue', 0], ['ivalue', ADA_MAX_INTEGER] ]]],
851         ['POSITIVE',  ['subtype', 'INTEGER',
852            ['range', ['ivalue', 1], ['ivalue', ADA_MAX_INTEGER] ]]],
853         ['STRING' , ['array', 'STRING', [['POSITIVE'], 'CHARACTER']]],
854         ['universal_integer', ['type', 'INTEGER',
855          ['range', ['ivalue', ADA_MIN_INTEGER],
856                    ['ivalue', ADA_MAX_INTEGER] ]]],
857         ['universal_real',   ['type', 'FLOAT',
858            ['digits', ['ivalue', ADA_MIN_REAL],
859                       ['ivalue', ADA_MAX_REAL],
860                       ['ivalue', ADA_REAL_DIGITS] ]]],
861        ['string_type', ['array', 'string_type',[['POSITIVE'],
862                                                 'character_type']]],
863       ['array_type',  ['array', 'any', [['any'], 'CHARACTER']]],
864         ['discrete_type', ['type', 'discrete_type']],
865         ['IMAGE',     ['attribute', 'STRING' ]],
866         ['ADDRESS',     ['attribute', 'INTEGER']],          --
867 --
868         ['DURATION',          ['type', '$FIXED',
869                               ['delta', ['ivalue', [-86400, 1]],
```

```
870                                     ['ivalue', [ 86400, 1]],
871                                     ['ivalue', [1, 100]] ]]],
872 --
873         ['I:1',    ['constant','universal_integer', ['ivalue', 1] ]],
874         ['UNMENTIONABLE#0', ['package']],
875         [standard, ['package']],
876         ['ASCII',  ['package']],
877 --
878         ['any',    ['type', 'any' ]],
879        ['any_id', ['obj',  'any' ]],
880         ['none',   ['type', 'none']],
881         ['NULL',   ['null', 'any' ]]
882            };
883
884    init_symbtab +:=
885         { [name,['attribute','universal_integer']] :
886                 name ∈ u_integer_attributes} +
887         { [name,['attribute','overloaded']] :
888                 name ∈ overloaded_attributes } +
889         { [name,['attribute','universal_real']] :
890                 name ∈ float_attributes } +
891         { [name,['attribute','BOOLEAN']] :
892                 name ∈ boolean_attributes };
893
894    (for v = init_symbtab(n)) SYMBTAB(n) := v; end for;
895
896 -- So that the usual naming conventions can be followed for the package
897 -- STANDARD, STANDARD is declared within yet another scope, UNMENTIONABLE.
898 --
899    DECLARED := {[standard, {}], ['ASCII', {}],
900                 ['UNMENTIONABLE#0', {['STANDARD', standard]}]
901                };
902
903
904 -- Operators (most) entered into SYMBTAB and DECLARED.
905
906    symbtab_enter(boolean_ops,     'boolean_type');
907    symbtab_enter(short_circ_ops, 'boolean_type');          --
908    symbtab_enter(equal_ops,       'BOOLEAN')    ;
909    symbtab_enter(order_ops,       'order_type') ;
910    symbtab_enter(arith_ops,      'numeric')    ;
911    symbtab_enter(mod_ops,         'numeric')    ;
912    symbtab_enter({'**'},         'numeric')    ;
913    symbtab_enter({'&'},          'array_type') ;
914    symbtab_enter({'any_op'},      'any')        ;
915
916    -- The ABS operator is not a reserved word, and the interpreter
917    -- nevertheless expects its name in lower case. Thus :
918    declared(standard)('ABS') := declared(standard)('abs') ;
```

```
919
920     int_ops  := { opn + 'i'  : opn ∈ arith_ops + mod_ops } ;
921     float_ops := { opn + 'fl' : opn ∈ arith_ops} ;
922     fixed_ops := { opn + 'fx' : opn ∈ arith_ops} ;
923
924     new_arith_ops(int_ops, 'universal_integer') ;
925     new_arith_ops(float_ops, 'universal_real') ;
926     new_arith_ops(fixed_ops, '$FIXED') ;
927
928     -- The fixed multiplication and division operators return the
929     -- singular type universal_fixed, which must be explicitly con-
930     -- verted before use.
931
932
933     type_of('*fx') := type_of('/fx') := 'universal_fixed' ;
934
935
936     -- There are two types of mixed mode arithmetic operators: the ones
937     -- for fixed-integer operations, used at run-time, and the ones for
938     -- universal_real - universal_integer operations, used only in literal
939     -- espressions.
940
941     new_arith_ops({'*fxi', '*ifx', '/fxi'}, '$FIXED') ;
942     new_arith_ops({'*fli', '*ifl', '/fli'}, 'universal_real') ;
943     new_arith_ops({'**i'}, 'universal_integer') ;
944     new_arith_ops({'**fl'}, 'universal_real') ;
945
946 -- Exceptions entered into SYMBTAB.
947
948     (forall excep ∈ exceptions)
949         SYMBTAB(excep) := ['exception', 'exception', [ ]];
950     end forall;
951
952 -- Types and exceptions entered into DECLARED.
953
954 --
955     other_standard := {'ASCII', 'NULL'};
956     loop forall item ∈ (predefined_types + other_standard
957         + exceptions)
958     do
959         DECLARED(standard)(item) := item;
960     end loop;
961
962 -- Printable characters are entered into SYMBTAB, as overloaded
963 -- literals whose source name is the character between single quotes.
964
965     (forall i ∈ [32..126])
966         chain_overloads("'" + char i + "'",
967                                     ['literal', 'CHARACTER',[ ]]);
```

```
968     end forall ;
969
970     ascii_vals := { ["'" + char i + "'", i] : i ∈ [0 .. 127] } ;
971
972     SYMBTAB('CHARACTER') := ['enum', 'CHARACTER',
973             ['range', ['ivalue', 0], ['ivalue', 127] ] ,ascii_vals] ;
974
975     -- Enter the scope of ASCII to initialize that package with the named
976     -- characters.
977
978     scope_st with:= scope_info;
979     scope_info :=
980      ['ASCII','',['ASCII','STANDARD#0','UNMENTIONABLE#0'],[ ],[ ],''] ;
981
982     char_names := [
983         'NUL', 'SOH', 'STX', 'ETX', 'EOT', 'ENQ', 'ACK', 'BEL',
984         'BS', 'HT', 'LF', 'VT', 'FF', 'CR', 'SO', 'SI',
985         'DLE', 'DC1', 'DC2', 'DC3', 'DC4', 'NAK', 'SYN', 'ETB',
986         'CAN', 'EM', 'SUB', 'ESC', 'FS', 'GS', 'RS', 'US',
987        'EXCLAM', 'QUOTATION', 'SHARP', 'DOLLAR', 'PERCENT','AMPERSAND',
988        'COLON', 'SEMICOLON', 'QUERY', 'AT_SIGN','L_BRACKET',
989        'BACK_SLASH', 'R_BRACKET', 'CIRCUMFLEX', 'UNDERLINE', 'GRAVE',
990        'LC_A', 'LC_B', 'LC_C', 'LC_D', 'LC_E', 'LC_F', 'LC_G', 'LC_H',
991         'LC_I', 'LC_J', 'LC_K', 'LC_L', 'LC_M', 'LC_N', 'LC_O', 'LC_P',
992         'LC_Q', 'LC_R', 'LC_S', 'LC_T', 'LC_U', 'LC_V', 'LC_W', 'LC_X',
993         'LC_Y', 'LC_Z', 'L_BRACE','BAR', 'R_BRACE',     'TILDE','DEL'
994                 ];
995
996     char_values := [0..31] + [33..38] + [58,59,63,64] + [91..127] ;
997
998     (forall c = char_names(i))
999         DECLARED('ASCII')(c) := c ;
1000        SYMBTAB(c) := ['constant', 'CHARACTER',
1001                                   ['ivalue', char_values(i)]] ;
1002    end forall ;
1003
1004
1005    popscope(); -- Leave ASCII, back to STANDARD
1006
1007 -- For the enumeration type BOOLEAN
1008
1009    booleans := ['FALSE', 'TRUE'] ;
1010    (forall i ∈ [1..2])
1011       chain_overloads(booleans(i), ['literal','BOOLEAN',[ ] ]) ;
1012    end forall ;
1013
1014    SYMBTAB('BOOLEAN') := ['enum', 'BOOLEAN',
1015                ['range', ['ivalue', 0], ['ivalue', 1]] ,
1016                          { ['FALSE', 0], ['TRUE', 1]} ] ;
```

```
1017
1018
1019 --  The only predefined aggregate is the one for string literals.
1020     new_agg_or_access('aggregate', 'STRING');
1021
1022
1023 -- Initialize the scope map for all identifiers in standard packages.
1024
1025    (forall [scop,decl] ∈ declared)
1026       (forall [-,id] ∈ decl) SCOPE_OF(id) := scop ; end ;
1027    end forall;
1028    (for t ∈ predefined_types)
1029        root_type(t) := t ;
1030    end for ;
1031 -- except for...
1032 root_type('NATURAL')    := root_type('POSITIVE') := 'INTEGER' ;
1033 root_type('DURATION')   := '$FIXED' ;
1034 --- root_type('array_type') := 'any' ;
1035    BASE_DECLARED := declared;
1036    VISIBLE('ASCII') := DECLARED('ASCII');
1037    comp_units := 0 ;
1038    NEWTYPES with:= [ ] ;
1039
1040 end proc INIT_SEM;
```

ADASEM : Grammar rules for semantic processing

```
1042 proc adasem(node) ;
1043
1044 -- This is the driver routine for all semantic processing. It is called
1045 -- by the parser whenever the syntax tree for a compilation unit has
1046 -- been built. The input to this routine is an AST node, on which two
1047 -- maps are defined : AST, and SPANS. These maps are global to the front
1048 -- end.
1049
1050 macro sem(n)        ; adasem(tree(n))              endm ;
1051 macro sem_list(n)   ; [adasem(i) : i ∈ tree(n)]    endm ;
1052 macro line_1(lines) ; spans(lines)(1)(1)           endm ; --
1053
1054
1055
1056
1057 tree := AST(node) ;
1058
1059 if    is_string(node) or is_empty(node) then return node ;
1060 elseif is_string(tree) or is_empty(tree) then return tree ;
1061 elseif tree = om then
1062     WARNING('Probable System error : undefined node') ;
1063     return [ ] ;
1064 end if ;
1065
1066
1067 -- The current node is placed in a global variable, from which the error
1068 -- routines can extract its span.
1069
1070 save_node := current_node ;
1071 current_node := node ;
1072
1073
1074 case tree(1) of
1075
```

Chapter 2. Lexical elements

```
1077 -- pragma → ['pragma' identifier argument_list]
1078
1079     ('pragma'):
1080         result := [process_pragma(node)] ;
1081
1082 -- argument_association → ['arg' identifier expression]
```

```
1083
1084    ('arg'):
1085        result := [sem(2), sem(3)] ;
1086
```

Chapter 3. Declarations and types

```
1088 -- object_declaration → ['obj_decl' identifier_list subtype_indic
1089 --                                                  opt_expression]
1090
1091    ('obj_decl'):
1092        result := obj_decl(node) ;
1093
1094 -- const_declaration → ['const_decl' identifier_list subtype_indic
1095 --                                                  opt_expression]
1096
1097    ('const_decl'):
1098        result := const_decl(node) ;
1099
1100
1101 -- num_declaration → ['num_decl' identifier_list expression]
1102
1103    ('num_decl'):
1104        result := number_decl(sem_list(2), sem(3)) ;
1105
1106
1107 -- type_decl → ['type_decl' identifier discriminant_list
1108 --                                                  type_definition]
1109
1110    ('type_decl'):
1111        discriminants := sem_list(3) ;
1112        result := [ais_type(type_decl(sem(2), discriminants, tree(4)))];
1113
1114
1115 -- Subtype_decl → ['subtype_decl' identifier subtype_indic]
1116
1117    ('subtype_decl'):
1118        subtype := make_subtype(sem(3)) ;
1119        result := subtype_decl(sem(2), subtype) ;
1120
1121
1122 -- subtype_indication → ['subtype_indic', name opt_constraint]
1123
1124    ('subtype_indic'):
1125        result := [sem(2), sem(3)] ;
1126
1127
1128 -- derived_type_definition → ['derived_type' subtype_indication]
1129
```

```
1130    ('derived_type'):
1131        result := ['new', make_subtype(sem(2))] ;
1132
1133
1134 -- discrete_range → ['range' expression expression]
1135
1136    ('range'):
1137          lo := sem(2) ;
1138          hi := sem(3) ;
1139          check_range_attribute(lo) ;
1140          check_range_attribute(hi) ;
1141        result := ['range', lo, hi] ;
1142
1143
1144 -- discrete_range → ['range_expression' expression]
1145
1146    ('range_expression'):
1147        result := sem(2) ;
1148
1149
1150 -- constraint → ['constraint' general_aggregate]
1151
1152    ('constraint'):
1153          result := sem_list(2) ;
1154
1155
1156 -- enumeration_type → ['enum' enumeration_literal_list]
1157
1158    ('enum'):
1159          result := ['enum', sem_list(2)] ;
1160
1161
1162    ('int_type'):
1163        result := ['int_type',   sem(2)] ;
1164
1165
1166    ('float_type'):
1167        result := ['float_type', sem(2)] ;
1168
1169    ('fixed_type'):
1170        result := ['fixed_type', sem(2)] ;
1171
1172
1173    ('digits'):
1174        result := ['digits', sem(2), sem(3)] ;
1175
1176
1177    ('delta'):
1178        result := ['delta', sem(2), sem(3)] ;
```

```
1179
1180
1181 -- array_type_definition → ['array_type' index_list subtype_indication]
1182
1183    ('array_type'):
1184       index_list := [make_index(adasem(i)) : i ∈ tree(2)] ;
1185       result := array_typedef(index_list,
1186                                  promote_subtype(make_subtype(sem(3)))) ;
1187
1188
1189 -- subtype_definition → ['box' identifier]
1190
1191    ('box'):
1192       result := ['box', sem(2)] ;
1193
1194
1195 --  discrete_range                → ['subtype' opt_name range_constraint]
1196 --  general_component_association → ['subtype' opt_name range-constraint]
1197
1198    ('subtype'):
1199       opt_name := if not is_empty(tree(2)) then find_old(sem(2))
1200                          else [ ] end ;
1201       result := ['subtype', opt_name, sem(3)] ;
1202
1203
1204 -- record_decl → ['record' component_list]
1205
1206    ('record'):
1207        [components, variant] := sem(2) ;
1208        result := ['record', components, variant] ;
1209
1210
1211 -- component_list → [ 'component_list' component_decl_list variant]
1212
1213    ('component_list'):
1214         comps   := [ ]+/sem_list(2) ;          --
1215         variant := sem(3) ;
1216         result  := [comps, variant] ;
1217
1218
1219 -- component_declaration → ['field' identifier_list subtype_indic
1220 --                                                    opt_expression]
1221
1222    ('field'):
1223       type_mark := promote_subtype(make_subtype(sem(3))) ;
1224       result := comp_decl(sem_list(2), type_mark, sem(4)) ;
1225
1226
1227 -- discr_specification → ['discr_spec' identifier_list name opt_expr]
```

```
1228
1229    ('discr_spec'):
1230       result := [sem_list(2), sem(3), sem(4)] ;                --
1231
1232
1233 -- variant_part → ['variant_decl' simple_name variant_list]
1234
1235    ('variant_decl'):
1236        result := variant_decl(sem(2), tree(3)) ;
1237
1238
1239 -- component_association → ['choice_list' choice_list expression]
1240
1241    ('choice_list'):
1242    result := ['choice_list', sem_list(2), sem(3)] ;
1243
1244    ('simple_choice'):
1245       result := ['simple_choice', sem(2), node] ;
1246
1247
1248    ('range_choice'):
1249        result := ['range_choice',  sem(2), node] ;
1250
1251
1252    ('others_choice'):
1253       result := ['others_choice', '',     node] ;
1254
1255
1256    ('access_type'):
1257       subtype_indic := sem(2) ;
1258       [id, constraint] := subtype_indic ;
1259        if constraint = [ ] and is_type(type_mark := check_old(id))
1260       and type_of(type_mark) = 'incomplete'
1261        then
1262          -- only permissible context for an incomplete type.
1263          desig_type := type_mark ;
1264       else
1265          desig_type := promote_subtype(make_subtype(subtype_indic)) ;
1266       end if ;
1267
1268       result := ['access', desig_type] ;
1269
1270
1271 -- incomplete_type_decl → ['incomplete_decl' identifier discriminant]
1272
1273    ('incomplete_decl'):
1274       discriminants := sem_list(3) ;
1275       type_name := incomplete_decl(sem(2), discriminants) ;
1276       result := [['type', type_name, ['delayed']]] ;           --
```

```
1277
1278
1279 -- declarations → ['declarations'  declaration_list]
1280
1281    ('declarations'):
1282       result := [ ] ;
1283
1284       (forall d ∈ tree(2))
1285            NEWTYPES with:= [ ] ;       -- To collect anonymous types
1286          decl := adasem(d) ;
1287          anon_types := new_type_list() ; -- generated by declaration.
1288            result +:=
1289              (if spans(d) ≠ om then [['stmt', line_1(d)]] else [ ] end)
1290              + anon_types + decl ;
1291       end forall ;
1292
```

Chapter 4. Names and expressions

```
1294 -- name  → ['character_literal'  character]
1295 -- Character literals also appear as enumeration literals, and as
1296 -- selectors.
1297
1298    ('character_literal'):
1299          result := "'" + tree(2) + "'";
1300
1301
1302 -- name  → ['simple_name'  identifier]
1303
1304    ('simple_name'):
1305       result := tree(2) ;
1306
1307
1308 -- name  → ['call?'  name  general_aggregate]
1309
1310    ('call?'):
1311 --
1312       nam := if node_kind(tree(2)) = 'string' then
1313            LOWER_CASE_OF(sem(2)) else sem(2) end ;
1314 --
1315       result := ['call?', nam, sem_list(3)] ;
1316
1317
1318 -- name → ['operator'  operator_symbol]
1319
1320    ('operator'):
1321          result := tree(2) ;
1322
1323    ('string'):
```

```
1324          result := tree(2) ;
1325
1326
1327
1328 -- name → ['.' name selector]
1329
1330    ('.'):
1331       result := ['.', sem(2), sem(3) ] ;
1332
1333    ('@'):
1334       result := ['@', sem(2)] ;
1335
1336    ('"'):
1337       result := ['"', sem(2), sem(3), 'I:1'] ;
1338
1339
1340 -- aggregate → ['aggregate' expression_list]
1341
1342    ('aggregate'):
1343       result := ['aggregate'] + sem_list(2) ;
1344
1345
1346 -- parenthesised_expression → ['()', expression]
1347
1348    ('()'):
1349          result := ['()', sem(2)] ;
1350
1351
1352 -- expression → [operator_designator expression opt_expression]
1353
1354    ('and', 'or', 'xor', 'mod', 'rem', 'andthen',   'orelse',
1355     '=',  '≠',  '<',   '<=',  '>',  '>=',
1356     '+',  '-',  '&',   '*',   '/',  '**' ):
1357
1358          op_name := find_old(tree(1)) ;
1359          arg1 := sem(2) ;
1360          arg2 := sem(3) ;
1361          check_range_attribute(arg1) ;                    --
1362          check_range_attribute(arg2) ;
1363          result := [op_name, arg1, arg2] ;
1364
1365    ('in',  'notin'):
1366
1367          op_name := tree(1) ;
1368          arg1 := sem(2) ;
1369          arg2 := sem(3) ;
1370
1371          if arg2(1) = 'range' then
1372             arg2 := ['subtype', [ ], arg2] ;
```

```
1373        end if ;
1374
1375          result := [op_name, arg1, arg2];
1376
1377    ('not', 'abs'):
1378
1379          arg1 := sem(2) ;
1380          check_range_attribute(arg1) ;                    --
1381          result := [find_old(tree(1)), arg1] ;
1382
1383    ('+u', '-u'):
1384
1385       -- Recover user-defined functions with the same designator.
1386       desig := tree(1)(1) ;
1387          arg1 := sem(2) ;
1388          check_range_attribute(arg1) ;                    --
1389       result := [find_old(desig), arg1] ;
1390
1391
1392    ('int_literal'):
1393          result := tree ;
1394
1395
1396    ('real_literal'):
1397          result := tree ;
1398
1399
1400    ('string_literal'):
1401          result := tree ;
1402
1403
1404 ('null'):
1405     result := ['null'] ;                                --
1406
1407 ('name'):
1408     result := ['name', sem(2)] ;
1409
1410 ('qualify'):
1411     result := ['qualify', find_old(sem(2)), sem(3)] ;
1412
1413
1414 -- allocator → ['new_init' name aggregate]
1415
1416     ('new_init'):
1417        result := ['new_init', sem(2), sem(3)] ;
1418
1419
1420 -- allocator → ['new' name constraint_list]
1421
```

```
1422    ('new'):
1423        constraint := sem_list(3) ;
1424        result := ['new', sem(2), constraint] ;
1425
```

Chapter 5. Statements

```
1427 -- sequence_of_statements → ['statements' statement_list, label_list]
1428
1429    ('statements'):
1430
1431        labs := {find_old(l) : l ∈ sem_list(3)} ;
1432        (forall l ∈ labs) label_status(l) := 'reachable' ; end ;
1433        stats := [ ] +/ sem_list(2) ;
1434        result := [['labdef', l]: l ∈ labs] + stats ;
1435        (forall l ∈ labs) label_status(l) := 'unreachable' ; end ;
1436
1437 -- statement → ['statement' label_list statement]
1438
1439    ('statement'):
1440 --
1441        labs := {find_old(l) : l ∈ sem_list(2)} ;
1442        if exists l ∈ labs | nature(l) ≠ 'label' then
1443            errmsg(original_name(l), ' is not a visible label', '8.3');
1444        end if ;
1445        result := [['label', l]: l ∈ labs] + [sem(3)] ;
1446        if result ≠ [ ] then                -- Add line number for tracing.
1447          result := [['stmt' , line_1(current_node)]] +  result ;
1448       end if ;
1449
1450
1451
1452 -- labels_declaration → ['labels' label_list]
1453
1454    ('labels'):
1455        label_decl(sem_list(2)) ;
1456        result := [ ] ;
1457
1458 -- null_statement → [null_s']
1459
1460    ('null_s'):
1461        result := ['null'] ;
1462
1463
1464 -- assignment → [':=' name expression ]
1465
1466    (':='):
1467        result := assign_statement(sem(2), sem(3)) ;
1468
```

```
1469
1470 -- if_statement → ['if' if_part_list opt_else]
1471
1472    ('if'):
1473       if_part := [ [adasem(cond), adasem(stats)] :
1474                                          [cond, stats] ∈ tree(2)] ;
1475       result := ['if', if_part, sem(3)] ;
1476
1477
1478 -- condition → ['condition' expression]
1479
1480 ('condition'):
1481    result := check_type('boolean_type', sem(2)) ;          --
1482
1483
1484 -- case_statement → ['case' expression alt_list]
1485
1486    ('case'):
1487          result := case_statement(sem(2), tree(3)) ;
1488
1489
1490 -- loop_statement → ['loop'  opt_loop_id iteration_rule statements]
1491
1492    ('loop'):
1493          loop_id := sem(2) ;
1494 --
1495 -- loop names are declared when procedure containing them is entered.
1496 --
1497          loop_name := find_old(loop_id) ;
1498          SYMBTAB(loop_name) := ['block' , 'loop', [false]] ;
1499          newscope(loop_name) ;
1500          iterator := if is_empty(tree(3)) then ['loop'] else sem(3) end ;
1501          result := loop_statement(iterator, sem(4)) ;
1502
1503
1504 -- iteration_rule → ['while'  condition]
1505
1506    ('while'):
1507       result := ['while', sem(2)] ;
1508
1509
1510 -- iteration rule → ['for'    identifier  discrete_range]
1511
1512    ('for'):
1513       result := ['for',    iter_var(sem(2), sem(3))] ;
1514
1515
1516 -- iteration_rule → ['forrev' identifier  discrete_range]
1517
```

```
1518    ('forrev'):
1519        result := ['forrev', iter_var(sem(2), sem(3))] ;
1520
1521
1522 -- block → ['block' identifier declarations statements exceptions]
1523
1524    ('block'):
1525        opt_id := sem(2) ;
1526        new_block(if is_empty(opt_id) then str newat else opt_id end) ;
1527
1528        result := ['block', block(sem(3), sem(4), sem(5)) ] ;
1529         popscope() ;
1530
1531 -- exit_statement → ['exit' opt_name opt_expression]
1532
1533    ('exit'):
1534        result := exit_statement(sem(2), sem(3)) ;
1535
1536
1537 -- return_statement → ['return' opt_expression]
1538
1539    ('return'):
1540        result := return_statement(sem(2)) ;
1541
1542
1543 ('goto'):
1544    result := goto_statement(sem(2)) ;                    --
1545
```

Chapter 6. Subprograms

```
1547 -- subprogram_declaration → ['subprogram_decl', subprogram_spec]
1548
1549 ('subprogram_decl'):
1550    result := subprog_decl(sem(2)) ;                      --
1551
1552 -- subprogram_specification → ['procedure' identifier formals_list]
1553 --                          → ['function'  identifier formals_list name]
1554
1555 ('procedure', 'function'):
1556
1557    id := sem(2) ;
1558    -- Save possible previous meaning of name, and introduce subprogram
1559    -- name, to prevent premature use in declarations of formals.
1560
1561    old_name := declared(scope_name)(id) ;
1562    if old_name ≠ om then old_sym := SYMBTAB(old_name) ;
1563        declared(scope_name)(id) := om ;
1564    end if ;
```

```
1565
1566    new_name := find_new(id) ;
1567    newscope(new_name) ;
1568    formals := get_formals(sem_list(3)) ;                    --
1569    popscope() ;
1570
1571    declared(scope_name)(id) := old_name ;  -- restore current meaning
1572    if old_name ≠ om then SYMBTAB(old_name) := old_sym ; end if ;
1573
1574    return_type := if tree(1) = 'procedure' then 'none'
1575                                            else find_old(sem(4)) end ;
1576
1577    result := [tree(1), id, formals, return_type] ;
1578
1579
1580
1581 -- subprogram_body → ['subprogram' subprogram_spec  declarations
1582 --                                                  statements opt_exceptions]
1583
1584 ('subprogram'):
1585    result := subprog_body(node) ;
1586
1587 -- parameter_specification → ['formal' id_list mode name opt_expression]
1588
1589 ('formal'):
1590    result := [sem_list(2), sem(3), sem(4), sem(5)] ;          --
1591
1592 -- mode → ['mode'  identifier]
1593
1594    ('mode'):
1595        result := tree(2) ;
1596
1597
1598 -- call_statement → ['call' name]
1599
1600    ('call'):
1601       result := statement_name(sem(2)) ;
```

Chapter 7. Packages

```
1603 -- package_specification → ['package_spec' identifier declarations
1604 --                                                  opt_private_part]
1605
1606 ('package_spec'):
1607    pack_head := package_head(sem(2)) ;
1608    visible_decls := sem(3) ;
1609    -- The declarations so far constitute the visible part of the package
1610    visible(scope_name) := declared(scope_name) ;
1611    -- Check that no incomplete types are left.
```

```
1612    (forall [-,name] ∈ visible(scope_name))
1613        if type_of(name) = 'incomplete' then
1614           errmsg('missing full declaration for ',
1615                                        original_name(name),'3.8.1') ;
1616        end if ;
1617    end forall ;
1618    -- Now process private part of package.
1619
1620    newscope(str newat) ;
1621    nature(scope_name) := 'private_part' ;
1622    priv_decls := end_private(sem(4)) ;
1623    specs := visible_decls + priv_decls;
1624    result := end_specs(pack_head, specs) ;
1625
1626 -- package_body → ['package_body' identifier declarations
1627 --                                    opt_statements  opt_handler]
1628
1629    ('package_body'):
1630        pack := module_body_id('package', sem(2)) ;
1631        body := block(sem(3), sem(4), sem(5)) ;
1632        result := module_body(pack, body) ;
1633
1634
1635 -- private_type_declaration → ['private_decl' identifier
1636 --                                    discriminant_list priv_kind]
1637
1638    ('private_decl'):
1639        discriminants := sem_list(3) ;
1640        type_name := private_decl(sem(2), discriminants, tree(4)) ;
1641       result := [['type', type_name, ['delayed']]] ;
1642
```

Chapter 8. Visibility rules

```
1644 -- use_clause → ['use' identifier_list]
1645
1646    ('use'):
1647       use_clause(sem_list(2)) ;
1648       result := [ ] ;
1649
1650
1651 -- renaming_declaration → ['rename_ex' identifier name]
1652
1653    ('rename_ex'):
1654       result := renames(sem(2), 'exception', sem(3)) ;
1655
1656
1657 -- renaming_declaration → ['rename_pack' identifier  name]
1658
```

```
1659    ('rename_pack'):
1660        result := renames(sem(2), 'package',   sem(3)) ;
1661
1662
1663 -- renaming_declaration → ['rename_obj' identifier type_mark name]
1664
1665    ('rename_obj'):
1666        result := renames(sem(2), sem(3), sem(4)) ;
1667
1668
1669 -- renaming declarations → ['rename_sub' subprogam_spec name]
1670
1671    ('rename_sub'):
1672        [kind, name, formals, ret_type] := sem(2) ;
1673        result := renames([name, formals, ret_type], kind, sem(3)) ;
```

Chapter 9. Tasks

```
1675 -- task_specification → [task_kind identifier opt_entry_declaration
1676 --                                                    opt_rep_clause]
1677 -- task_kind          → 'task_spec'
1678 --                    → 'task_type_spec'
1679
1680    ('task_spec', 'task_type_spec'):
1681        result := task_spec(node) ;
1682
1683
1684 -- task_body → ['task' identifier declarations statements
1685 --                                                    opt_exceptions]
1686
1687    ('task'):
1688        task := module_body_id('task', sem(2)) ;
1689        body := block(sem(3), sem(4), sem(5)) ;
1690        result := module_body(task, body) ;
1691
1692
1693 -- entry_declaration → ['entry' identifier formals_list]
1694
1695    ('entry'):
1696        formals := get_formals(sem_list(3)) ;                    --
1697        result := entry_decl(sem(2), formals) ;
1698
1699
1700 -- entry_declaration → ['entry_family' identifier discrete_range
1701 --                                                    formals_list]
1702
1703    ('entry_family'):
1704        formals := get_formals(sem_list(4)) ;                    --
1705        result := entry_family_decl(sem(2), sem(3), formals) ;
```

```
1706
1707
1708 -- accept_statement → ['accept' name opt_expression opt_formal_part
1709 --                                                    opt_statements]
1710
1711    ('accept'):
1712         id := sem(2) ;
1713         indx := sem(3) ;
1714         formals := get_formals(sem_list(4)) ;
1715        result := accept_statement(id, indx, formals, tree(5)) ;
1716
1717
1718 -- delay_statement → ['delay'  expression]
1719
1720    ('delay'):
1721    result := ['delay', check_type('DURATION', sem(2))] ;
1722
1723
1724 -- selective_wait → ['selective_wait' alternative_list else_part]
1725
1726    ('selective_wait'):
1727       result := select_statement(node) ;
1728
1729
1730
1731 -- select_alternative → ['guard' condition selective_wait_alternative]
1732
1733    ('guard'):
1734         result := ['guard', sem(2), sem(3)] ;
1735
1736 -- selective_wait_alternative → ['accept_alt' accept_statement opt_stats]
1737 --                            → ['delay_alt'  delay_statement  opt_stats]
1738
1739    ('accept_alt'):
1740       alternative := sem(2) ;
1741         opt_stats := sem(3) ;
1742         result := alternative with opt_stats ;
1743
1744    ('delay_alt'):
1745         delay := sem(2) ;
1746         opt_stats := sem(3) ;
1747        result := delay with opt_stats ;
1748
1749
1750 -- selective_wait_alternative → ['terminate_alt' ]
1751
1752    ('terminate_alt'):
1753         result := terminate_statement() ;                    --
1754
```

```
1755
1756 -- conditional_entry_call → ['conditional_entry_call' call_statement
1757 --                                                    statements else_stat]
1758
1759    ('conditional_entry_call'):
1760        call := check_entry_call(tree(2));                    --
1761        result := ['conditional_entry_call', call, sem(3), sem(4)] ;
1762
1763
1764 -- timed_entry_call → ['timed_entry_call',call_statement statements
1765 --                                                    delay_alternative]
1766
1767    ('timed_entry_call'):
1768        call := check_entry_call(tree(2)) ;
1769        [ -, delay_exp, opt_stats] := sem(4) ;                --
1770      result := ['timed_entry_call',call, sem(3),             --
1771                                    ['delay',delay_exp], opt_stats];
1772
1773
1774 -- abort_statement → ['abort' task_name_list]
1775
1776    ('abort'):
1777        result := abort_statement(tree(2)) ;
1778
```

Chapter 10. Program structure...

```
1780    ('compilation'):
1781    -- This node is used for pragmas that precede a compilation unit.
1782    -- TBSL
1783       result := [ ] ;
1784
1785
1786 -- unit_declaration → ['unit' context_clause unit_body]
1787
1788    ('unit'):
1789       context := [ [ ]+/sem_list(2), top(tree(2))] ;          --
1790       result := compunit(sem(3)) ;
1791
1792
1793 -- context_clause → ['with_use_list' [with_or_use...]]
1794    ('with_use_list'):
1795       result := [ ] ;
1796        (for node ∈ tree(2))
1797          if (kind := node_kind(node)) ∈ ['with', 'use'] then
1798              [-,n_list] := AST(node);
1799            result with:= [kind, [adasem(n) : n ∈ n_list]] ;
1800          else                              -- pragma ELABORATE
1801              result with:= process_pragma(node) ;
```

```
1802            end if ;
1803         end for ;
1804         -- TBSL: validate names in with clauses at this point.
1805
1806
1807 -- body_stub    → ['subprogam_stub' subprogram_specification]
1808 --              → ['package_stub'  name]
1809 --              → ['task_stub'     name]
1810
1811    ('subprogram_stub'):
1812       name := subprog_spec(sem(2)) ;                              --
1813        nature(name) +:= '_spec' ;
1814       if original_name(name) ∈ op_designators then
1815            errmsg('Name of separately compiled unit cannot be ',
1816                        'an operator designator', '10.1') ;
1817            result := [ ] ;
1818        else
1819            result := save_stub('subprog', name) ;
1820       end if ;
1821
1822
1823    ('package_stub'):
1824         name := stub_head('package', sem(2)) ;
1825         result := save_stub('body', name) ;
1826
1827    ('task_stub'):
1828         name := stub_head('task', sem(2)) ;
1829         result := save_stub('body', name) ;
1830
1831
1832 -- subunit  → ['separate' parent_name unit]
1833
1834    ('separate'):
1835       separate_context(sem(2)) ;
1836         result := sem(3) ;
1837
```

Chapter 11. Exceptions

```
1839
1840 -- Exception_declaration  → ['except_decl'  identifier_list]
1841
1842    ('except_decl'):
1843         result := except_decl(sem_list(2)) ;
1844
1845
1846 -- exceptions  → ['exception'  handler_list]
1847
1848    ('exception'):
```

```
1849      result := exception_part(node) ;
1850
1851
1852 -- exception_handler → ['handler' exception_choice_list statements]
1853
1854    ('handler'):
1855       result := exception_handler(node) ;
1856
1857    ('others'):
1858         result := 'others' ;
1859
1860
1861 -- raise_statement → ['raise opt_identifier]
1862
1863    ('raise'):
1864       result := raise_statement(sem(2)) ;
1865
```

Chapter 12. Generics

```
1867
1868 ('generic_procedure','generic_function'):
1869     result := generic_subprog_spec(node) ;
1870
1871
1872 ('generic_package'):
1873     result := generic_pack_spec(node) ;
1874
1875
1876 -- Generic part → ['generic_formals generic_decl_list]
1877
1878    ('generic_formals'):
1879         NEWTYPES with:= [ ] ;      -- Anonymous types may be created (???)
1880         generic_list := [ ]+/sem_list(2) ;
1881         result := [generic_list, new_type_list()] ;
1882
1883
1884 -- Generic_formal → ['generic_obj' id_list mode name opt_expression]
1885
1886    ('generic_obj'):
1887     result := generic_obj_decl(node) ;
1888
1889
1890 -- Generic formal → ['generic_type' identifier type_def]
1891
1892    ('generic_type'):
1893         result := generic_type_decl(node) ;
1894
1895
```

```
1896 -- Generic_formal → ['gen_priv_type' private_type_declaration]
1897
1898    ('gen_priv_type'):
1899        result := generic_priv_decl(node) ;
1900
1901
1902 -- Generic_formal → ['generic_subp', subprogram_spec opt_is]
1903
1904    ('generic_subp'):
1905        result := generic_subp_decl(node) ;
1906
1907
1908 -- Generic_type_definition → ['generic' identifier]
1909
1910    ('generic'):
1911        return tree ;
1912
1913
1914 -- Package_instance → ['package_instance' identifier name instance_list]
1915
1916    ('package_instance'):
1917        new_name     := package_head(sem(2)) ;
1918       generic_name := find_old(sem(3)) ;
1919        instance     := sem_list(4) ;
1920        result := package_instance(new_name, generic_name, instance) ;
1921
1922
1923 -- subprogram_instance
1924 --       → ['function_instance' designator name generic_actual_part]
1925 --       → ['procedure_instance' identifier name generic_actual_part]
1926
1927 ('function_instance', 'procedure_instance'):
1928
1929    kind := tree(1) ;
1930    new_id := sem(2) ;
1931    new_compunit('subprog', new_id) ;                              --
1932    generic_name := find_old(sem(3)) ;
1933    instance := sem_list(4) ;
1934    result := subprog_instance(kind, new_id, generic_name, instance) ;
1935
1936
1937
1938 -- generic_parameter_association → ['instance' opt_identifier expression]
1939
1940 ('instance'):
1941    result := [sem(2), sem(3)] ;
1942
```

Chapter 13. Representation specs...

```
1944
1945 -- length_clause → ['length_clause' attribute simple_expression ]
1946
1947  ('length_clause'):
1948      [-, attr_type, attr_prefix, -] := sem(2);
1949      result := length_clause (attr_type, attr_prefix, sem(3));
1950
1951
1952 -- enumeration_representation_clause → ['enum_rep_clause'
1953 --                                    simple_name aggregate ]
1954
1955  ('enum_rep_clause'):
1956      type_name := sem(2);
1957      result := enum_rep_clause (type_name, tree(3));
1958
1959
1960 -- record_representation_clause →
1961 --    ['rec_rep_clause' simple_name opt_align_clause comp_clause_list ]
1962
1963  ('rec_rep_clause'):
1964     type_name := sem(2);
1965     align_clause := sem(3);
1966     comp_clause_list := sem_list(4);
1967     result :=rec_rep_clause(type_name, align_clause, comp_clause_list);
1968
1969
1970 -- component_clause → ['compon_clause' name simple_expression range]
1971
1972  ('compon_clause'):
1973      result := [sem(2), sem(3), sem(4)];
1974
1975
1976 -- address_clause → ['address_clause' simple_name simple_expression]
1977
1978  ('address_clause'):
1979      result := [ ];
1980
1981
1982 else
1983    errmsg('System error: invalid node type ', tree(1), 'none') ;
1984
1985 end case ;
1987
1988
1989 current_node := save_node ;
1990 return result;
```

```
1991
1992
1993 drop sem,
1994      line_1, --
1995      sem_list;
1996
1997 end proc adasem ;
1998
```

UTILITIES

symbtab_enter

```
2001 proc symbtab_enter(oprs, result) ;
2002
2003 -- Enter into symbol table the names of all predefined operators.
2004 -- They have generic types which cover all types derived from the pre-
2005 -- defined ones, and overload themselves.
2006
2007    const standard = 'STANDARD#0';
2008
2009
2010    (forall op_name ∈ oprs)
2011
2012        declared(standard)(op_name) := op_name ;
2013        -- Each of the predefined operators overloads itself.
2014        SYMBTABF(op_name) := ['op', result, [ ], {op_name}, standard] ;
2015
2016    end forall;
2017
2018 end proc symbtab_enter;
```

new_arith_ops

```
2020 proc new_arith_ops(oprs, result) ;
2021
2022 -- Arithmetic operators are introduced for integer, floating point and
2023 -- fixed point results. These operators are never accessed directly by
2024 -- name resolution routines, but rather by the type resolution routine
2025 -- valid_arith_args. As a result, they need not appear in declared(St.)
2026 -- and need not have nay overloading specified.
2027
2028    (forall op_name ∈ oprs)
2029        SYMBTAB(op_name) := ['op', result, [ ]] ;
2030    end forall ;
2031
2032 end proc new_arith_ops ;
```

ADA_SEMANTICS

```
2034 proc ADA_SEMANTICS(comp_unit) ;
2035
2036    adasem(comp_unit) ;
```

```
2037
2038 end proc ADA_SEMANTICS ;
```

end_sem

```
2040 proc end_sem ;
2041
2042 --clean up and turn off the lights.
2043
2044
2045    NATURE := TYPE_OF := SIGNATURE := OVERLOADS := SCOPE_OF:= XREF:= om;
2046    DECLARED := BASE_DECLARED := VISIBLE := om;
2047    SCOPE_ST := NEWTYPES := LAB_SCOPE_STK := om;
2048    STMLABS := LAB_SCOPE_MAP := LAB_SEEN := om;
2049    GOTO_STMTS := LABHIDE := OPEN_SCOPES := om;
2050    USED_MODS := VIS_MODS := om;
2051
2052    return;
2053 end proc end_sem ;
```

warning

```
2055 proc warning(msg);
2056
2057    p_s := SPANS(current_node) ? [[1, 1, 1]] ;
2058    ERR_LIST([p_s,'*** Warning: '+msg]);
2059
2060 end proc Warning;
```

errmsg

```
2062 proc errmsg(msg,obj,lrm_sec);              -- Semantic errors
2063
2064    macro expanderr(msg,obj,lrm_sec);
2065        ((msg) +
2066            (if type(obj)≠'STRING' then str(obj) else (obj) end)
2067          + (if lrm_sec = 'none' then '' else
2068            (' (RM ' + lrm_sec + ')') end) )
2069    endm;
2070
2071
2072    if type msg = 'STRING' then
2073        p_s := SPANS(current_node) ? [[1, 1, 1]] ;
2074        ERR_LIST([p_s,'*** Semantic Error: ' +
2075                        expanderr(msg, obj, lrm_sec)]);
2076    elseif type msg ≠ 'TUPLE' then
2077        ERR_LIST(msg);                      -- << Should never happen
2078    elseif type msg(2) ≠ 'STRING' then
2079        ERR_LIST(msg);                      -- << Should never happen
```

```
2080    else
2081        ERR_LIST([msg(1),'*** '+expanderr(msg(2),obj,lrm_sec)]);
2082    end if;
2083
2084    ERRCOUNT +:= 1;
2085
2086 drop expanderr;
2087 end proc errmsg;
2088
```

2. Lexical elements

process_pragma

```
2091 proc process_pragma(node) ;
2092
2093 -- This arbitrarily extensible procedure processes pragma declarations.
2094 -- The name of the pragma determines the way in which the args are
2095 -- processed. If no meaning has been attached to a pragma name, the user
2096 -- is notified, and the pragma is ignored.
2097
2098 [-, id_node, arg_list] := AST(node) ;
2099 id := adasem(id_node) ;
2100
2101
2102 if is_empty(arg_list) then              -- pragma with no parameters
2103
2104    case id of
2105      ('EDUMP',
2106       'TDUMP',
2107       'TRACE_ON',
2108       'TRACE_OFF',
2109       'TRACE_TABLES_OFF'):
2110        warning('This pragma is provided for compiler debugging only');
2111        return ['pragma', id] ;
2112    else
2113        errmsg('Format error in pragma', id, 'Appendices B,F');
2114        return [ ];
2115    end case id ;
2116
2117 else
2118    -- Process list of arguments.
2119    arg_list := [adasem(a) : a ∈ arg_list] ;
2120    -- New syntax : named associations can be used for pragmas. For
2121    -- now, discard the names.
2122
2123    args := [a(2) : a ∈ arg_list] ;
2124
2125    case id of
2126
2127    ('IO_INTERFACE') :
2128
2129        -- Current interface to predefined procedures (e.g. text_io).
2130        -- The pragma makes up the body of a predefined procedure.
```

```
2131        -- This body is formatted into a single tuple :
2132        --
2133        --                  [ 'predef_', marker , name1, name2...]
2134        --
2135        -- where the marker is the  second argument  of the  pragma. This
2136        -- marker is  used as an  internal switch by the tio interpreter.
2137        -- The remaining components of  the tuple are the unique names of
2138        -- the formal parameters of the procedure.
2139
2140          [-,id] := args(1) ;
2141        proc_name := declared(scope_name)(id) ;
2142          nat := nature(proc_name) ;
2143          rmatch(nat,'_spec') ;
2144          nature(proc_name) := nat ;   -- pragma supplies the body
2145        if #signature(proc_name) > 0 then
2146            formal_decl_tree(signature(proc_name)) := om ;
2147        end if ;
2148        return [nat, proc_name, build_signature(proc_name),
2149                    [[ ],
2150                    [['predef_', args(2)(2)] + [
2151                     [formal : [formal,-] ∈ signature(proc_name)]
2152                       + [find_old(t(2)) : t ∈ args(3..)]  ]],
2153                    [ ]]
2154                ] ;
2155
2156    ('ELABORATE'):
2157        return ['elaborate', [a(2): a ∈ args]] ;
2158
2159    ('PRIORITY'):
2160      if #args = 1 then
2161 --
2162          if 'SYSTEM' ∈ vis_mods then
2163            p_type := declared('SYSTEM')('PRIORITY') ;
2164          else                    -- there should be a warning.
2165            return [ ] ;
2166          end if ;
2167        priority := check_type(p_type, args(1));
2168        if is_value(priority) then
2169            return ['set_priority', priority(2)] ;
2170        else
2171            errmsg('Priority must be static','','Appendix B');
2172            return [ ];
2173        end if;
2174      else
2175        errmsg('Invalid format for pragma priority',
2176                    '','Appendix B');
2177        return [ ];
2178      end if #args;
2179
```

```
2180      ('DEBUG'):
2181      -- An implementation defined debug utility : the expressions in
2182      -- the argument list are written on the standard output in a simple
2183      -- format without the use of text_io.
2184
2185        arg_list := [ ];
2186        (for arg ∈ args)
2187           [aexpr, typ] := find_check_type(arg);
2188           if typ ≠ om then
2189              arg_list with:= aexpr;
2190           end if;
2191        end for;
2192        return ['pragma', 'DEBUG', arg_list] ;
2193
2194      ('DUMP','TRACE_TABLES_ON'):
2195
2196           warning('This pragma provided for compiler debugging only');
2197           arg_list := [ ];
2198           (for arg ∈ args)
2199              if is_tuple arg and arg(1) = 'name'
2200                and is_identifier(arg(2)) then
2201                 arg_list with:= arg(2);
2202              else
2203                 errmsg('Invalid pragma format','','none');
2204                 return [ ];
2205              end if;
2206           end for;
2207           return ['pragma', id, arg_list] ;
2208
2209      ('INCLUDE_CODE'):
2210
2211 -- Reads an AIS file and includes its compilation units in the current
2212 -- library. An implementation defined separate compilation feature.
2213
2214           (forall arg ∈ args)
2215              if is_tuple(arg) and arg(1) = {'string_literal'} then
2216                    INCLUDE_AIS(arg(2));
2217              else
2218                    errmsg('Invalid file name ','','none');
2219              end if;
2220           end forall;
2221           return [ ];
2222
2223        ('CONTROLLED',
2224          'INCLUDE',
2225          'INLINE',
2226           'INTERFACE',
2227          'LIST',
2228          'MEMORY_SIZE',
```

```
2229            'OPTIMIZE',
2230            'PACK',
2231            'STORAGE_UNIT',
2232            'SUPRESS',
2233            'SYSTEM'):
2234              warning('This pragma will have no effect');
2235              return [ ];
2236
2237        else
2238            errmsg('format error in pragma', id, 'Appendices B,F');
2239            return [ ];
2240
2241        end case id ;
2242
2243 end if ;
2244
2245 end proc process_pragma;
```

3. Declarations and types.

obj_decl

```
2248 proc obj_decl(node) ;
2249
2250 -- Process variable declaration.
2251 -- Verify that the type is a constrained one, or that default values
2252 -- exist for the discriminants of the type.
2253
2254
2255
2256 [-, id_list, type_indic_node, init_node] := AST(node) ;
2257
2258
2259 nam_list := [find_new(adasem(id_node)): id_node ∈ id_list] ;
2260 opt_init := adasem(init_node) ;
2261 type_mark := set_type_mark(nam_list, type_indic_node) ;
2262
2263 check_fully_declared(type_mark) ;
2264
2265
2266 -- If an initialization is provided, verify that it has the specified
2267 -- type.
2268
2269     if opt_init = [ ] then
2270         init_val := om ;
2271     else
2272         [-, init_val] := check_init(type_mark, opt_init) ;
2273     end if ;
2274
2275     if is_unconstrained(type_mark)                    --
2276     then
2277      errmsg('Unconstrained '+nature(type_mark),' in object declaration',
2278                                          '3.6.1, 3.7.2');
2279     end if ;
2280
2281 (forall n ∈ nam_list) nature(n) := 'obj' ; end forall ;
2282
2283 return [['object', nam_list, type_mark, init_val]];
2284
2285 end proc obj_decl;
2286
```

const_decl

```
2288 proc const_decl(node) ;
2289
2290 -- Process constant declarations. This may be a new declaration, or the
2291 -- full declaration of a deferred constant in the private part of a
2292 -- package. In this later case, recover the names of the constants, and
2293 -- update their definitions.
2294
2295
2296
2297 [-, id_list, type_indic_node, init_node] := AST(node) ;
2298
2299
2300 id_list := [adasem(id_node) : id_node ∈ id_list] ;
2301 opt_init := adasem(init_node) ;
2302
2303
2304  if nature(scope_name) = 'private_part' then
2305      package_name := open_scopes(2) ;
2306      if exists id ∈ id_list | declared(package_name)(id) ≠ om then
2307          -- It must be a deferred constant.
2308        return const_redecl(id_list, type_indic_node, opt_init) ;
2309        else                                    -- Fully private constant.
2310          pass ;
2311        end if ;
2312  end if ;
2313
2314  nam_list := [find_new(id): id ∈ id_list] ;
2315 --
2316  type_mark := set_type_mark(nam_list, type_indic_node);
2317
2318  if opt_init = [ ] then                -- Deferred constant.
2319       init_val := [ ] ;
2320
2321      if type_of(base_type(type_mark)) ? ''
2322                                  priv_types
2323        then
2324       errmsg('Missing initialization',' in constant declaration',
2325                                      '3.2');
2326
2327      elseif scope_of(type_mark) ≠ scope_name then
2328        errmsg('Wrong scope ', 'for type of deferred constant',
2329                                      '7.4') ;
2330
2331      elseif nature(scope_name)
2332                                  {'package_spec','generic_package_spec'}
2333      then
```

```
2334            errmsg('Invalid context for deferred constant',
2335                                              '','3.2, 7.4') ;
2336          elseif is_generic_type(type_mark)
2337           or is_generic_type(base_type(type_mark)) then        --
2338            errmsg('constants of a generic type cannot be deferred','',
2339                                                        '12.1.2') ;
2340          elseif is_anonymous(type_mark) then                   --
2341            errmsg('a deferred constant must be defined with a type mark',
2342                                                        '','7.4.3') ;
2343        end if ;
2344      else
2345        [t_m, init_val] := check_init(type_mark, opt_init) ;
2346
2347        if t_m ≠ type_mark then
2348          -- t_m is an anonymous type created from the bounds of init_val
2349            (for n ∈ nam_list) type_of(n) := t_m ; end for ;
2350        end if ;
2351      end if ;
2352
2353      (for n ∈ nam_list)
2354          nature(n)    := 'constant';
2355          signature(n) := init_val;
2356      end for ;
2357
2358      -- If the constant is a static scalar constant, there is no need to
2359      -- emit code for it: it is constant-folded wherever used. If
2360      -- it is a deferred constant, code is not emitted either : it
2361      -- will be emitted when the private declaration is processed.
2362
2363      if (is_value(init_val) and is_scalar_type(type_mark)) or
2364                                                  init_val = [ ] then
2365          return [ ];
2366      else
2367          return [['constant', nam_list, type_mark, init_val]];
2368      end if;
2369
2370  end proc const_decl;
2371
```

const_redecl

```
2373  proc const_redecl(id_list, type_indic_node, opt_init) ;
2374
2375  -- Process the full declaration of deferred constants. At least one id
2376  -- in id_list is known to have been declared in the visible part
2377  -- of the current package, whose private part is being processed.
2378
2379
2380
```

```
2381 package_name := open_scopes(2) ;
2382 nam_list := [ ] ;
2383 -- The type indication must be a subtype indication .
2384
2385 if node_kind(type_indic_node) = 'subtype_indic' then
2386    type_mark := promote_subtype(make_subtype(adasem(type_indic_node)));
2387 else
2388    -- An anonymous array is syntactically possible, but incorrect.
2389    type_mark := anonymous_array(type_indic_node) ;
2390 end if ;
2391
2392 (forall id ∈ id_list)
2393    u_n := declared(package_name)(id) ;
2394    if u_n = om then
2395       errmsg(id, ' is not a deferred constant','3.2, 7.4') ;
2396       continue forall id;
2397
2398    elseif nature(u_n) ≠ 'constant' or signature(u_n) ≠ [ ]
2399        then   errmsg('Invalid redeclaration of ', id,'8.3') ;
2400       continue forall id;
2401
2402    elseif (ut := type_of(u_n)) ≠ type_mark
2403       -- They may still be the same subtype of some private type.
2404       and (type_of(ut) ≠ type_of(type_mark) or
2405                                  signature(ut) ≠ signature(type_mark))
2406    then
2407       errmsg('incorrect type in redeclaration of ', id,'7.4, 7.4.1') ;
2408
2409    elseif opt_init = [ ] then              -- No initialization ?
2410        errmsg('Missing initialization in redeclaration of ',
2411                                        id,'7.4');
2412
2413    else
2414       [-,  init_val] := check_init(type_mark, opt_init) ;
2415        signature(u_n) := init_val ;
2416       TO_XREF(u_n) ;
2417        nam_list with:= u_n ;
2418    end if ;
2419
2420 end forall ;
2421
2422 return [['constant', nam_list, type_mark, init_val]] ;
2423
2424 end proc const_redecl ;
```

set_type_mark

```
2426
2427 proc set_type_mark(nam_list, type_indic_node) ;
2428
2429 -- Set the symbol table entry for object or constant declarations.
2430 -- The type indication is a subtype indication or an array definition. In
2431 -- the later case, an anonymous array type must be created for each item
2432 -- in the name list. For the interpreter, any of these types will do.
2433
2434 if node_kind(type_indic_node) = 'subtype_indic' then
2435
2436    type_mark := promote_subtype(make_subtype(adasem(type_indic_node)));
2437    (forall n ∈ nam_list)
2438       type_of(n) := type_mark ;
2439    end forall ;
2440
2441 else
2442    (forall n ∈ nam_list)
2443       type_mark := anonymous_array(type_indic_node) ;
2444       type_of(n) := type_mark ;
2445    end forall ;
2446 end if ;
2447
2448 return type_mark ;
2449
2450 end proc set_type_mark ;
```

check_init

```
2452 proc check_init(type_mark, opt_init) ;
2453
2454 -- Validate the initialization of an object or constant declaration.
2455 -- Return the resolved expression, and the type (or a subtype of it
2456 -- in the case of constants of unconstrained type).
2457
2458 if is_limited_type(type_mark) then
2459            errmsg('Initialization not available for entities of ',
2460                        'limited type','7.4.4') ;
2461 end if;
2462
2463 init_val := check_type(type_mark, opt_init) ;
2464
2465 if has_deferred_constant(init_val)  then
2466    errmsg(' premature use of deferred constant before its full ',
2467                                    'declaration', '7.4.3') ;
2468 elseif (is_tuple(init_val) and
2469   init_val(1) ∈ {'qual_discr', 'qual_length'} ) then
```

```
2470    -- remove constraint check: the interpreter performs it in
2471    -- any case.
2472    init_val := init_val(3) ;
2473 end if ;
2474 --
2475 if nature(type_mark) = 'array' and is_value(init_val) then
2476    -- Constant definition: bounds are given by the initial expression.
2477    -- Create an anonymous subtype for the constant with those bounds.
2478
2479    [-, init_array] := init_val ;
2480
2481    new_indices := [ ] ;
2482    -- Unpack aggregate to obtain actual bounds on each dimension,
2483    -- and create constrained index for each.
2484
2485    (for index ∈ index_types(type_mark))
2486
2487       [-, components, lo, hi] := init_array ;
2488       new_index := anonymous_type() ;                -- Its new subtype.
2489       SYMBTAB(new_index) := ['subtype', index,
2490             ['range', ['ivalue', lo, index], ['ivalue', hi, index]]] ;
2491       root_type(new_index) := root_type(index) ;
2492       new_indices with:= new_index ;
2493       init_array := components(1) ;        -- A representative element.
2494
2495    end for ;
2496
2497    new_array := anonymous_type() ;
2498    SYMBTAB(new_array) := ['subtype', type_mark,
2499                           [new_indices, component_type(type_mark)]] ;
2500    root_type(new_array) := root_type(type_mark) ;
2501    misc_type_attributes(new_array) := misc_type_attributes(type_mark) ;
2502
2503    type_mark := new_array ;
2504 end if ;
2505
2506 return [type_mark, init_val] ;
2507 --
2508 end proc check_init ;
```

is_deferred_constant

```
2510 proc is_deferred_constant(c) ;
2511    return
2512          is_identifier(c)
2513       and nature(c) = 'constant'
2514       and is_empty(signature(c)) ;
2515 end proc ;
2516 proc has_deferred_constant(e) ;
```

```
2517 return
2518  is_identifier(e) and nature(e) = 'constant' and is_empty(signature(e))
2519  or is_tuple(e) and
2520    ((kind := e(1)) = '()' and has_deferred_constant(e(2))
2521    or
2522     kind ∈ qualifiers    and has_deferred_constant(e(3))
2523    or
2524     kind = 'convert'      and has_deferred_constant(e(4)) ) ;
2525 end proc has_deferred_constant ;
2526 --
```

number_decl

```
2528 proc number_decl(id_list, expn) ;
2529
2530
2531
2532 -- A number declaration differs from a constant declaration in that
2533 -- the type of the declared object is a universal numeric type, obtained
2534 -- from the value of the  literal expression supplied for it.
2535 -- No intermediate code is generated for these: they act as macros,
2536 -- and are always represented by their value.
2537
2538 nam_list := [find_new(id): id ∈ id_list] ;
2539
2540 num_val := check_type('universal_type', expn)    ;
2541 number_type := expr_type(num_val) ;
2542
2543 if not is_value(num_val) then
2544     errmsg('Expect literal expression in number declaration',
2545                                          '','3.2') ;
2546     number_type := 'any' ;
2547
2548 end if ;
2549 (forall n ∈ nam_list)
2550     SYMBTAB(n) := ['constant', number_type, num_val] ;
2551 end ;
2552
2553 return [ ] ;
2554
2555 end proc number_decl ;
2556
```

type_decl

```
2558 proc type_decl(id, opt_disc, type_def) ;
2559
2560
2561
```

```
2562 -- Process a type declaration. Create new name for type, or find unique
2563 -- name already given for incomplete declaration.
2564
2565 type_name := find_type_name(id) ;
2566 if type_name = 'any' then
2567     return 'any' ;                    -- Invalid redeclaration, etc.
2568 end if ;
2569
2570 if opt_disc ≠ [ ] then
2571     if AST(type_def)(1)  ['record' , 'derived_type'] then      --
2572         errmsg('Invalid use of discriminants', '', '3.7.1') ;
2573     elseif type_of(type_name) ∈ incp_types then
2574         -- full declaration of incomplete or private type. Verify that
2575         -- discriminant declarations are conmformi[redacted].
2576         discr_redecl(type_name, opt_disc) ;
2577     end if ;
2578 elseif type_of(type_name) ∈ incp_types and has_discriminants(type_name)
2579 then
2580     errmsg('missing discriminants in full type declaration', '', '3.8');
2581 end if;
2582
2583 kind := type_of(type_name) ;
2584
2585 build_type(type_name, opt_disc, type_def) ;
2586
2587 if kind ∈ priv_types then check_priv_decl(kind, type_name) ; end if ;
2588
2589 return type_name ;
2590
2591 end proc type_decl ;
2592
2593
```

find_type_name

```
2595 proc find_type_name(id) ;
2596
2597
2598
2599 -- Create a unique  name for a type  definition, or find  the unique name
2600 -- already created, in the case of the  full declaration of an incomplete
2601 -- or  private type. An  incomplete type  declaration  will  be  found in
2602 -- the same scope as the full one; a private type declaration is given in
2603 -- the visible  part of a package, which  is the enclosing scope  for the
2604 -- private part in which the full declaration is given.
2605
2606 -- Find incomplete declaration, if some was given.
2607
2608 old_name := declared(scope_name)(id) ;
```

```
2609
2610
2611 incomplete := if nature(scope_name) ≠ 'private_part' then
2612     old_name
2613 else old_name ?                          -- May be fully private, or
2614     declared(open_scopes(2))(id)          -- in enclosing scope(visible).
2615 end ;
2616 if incomplete = om then                   -- New type declaration.
2617     type_name := find_new(id) ;
2618
2619 else                                      -- Previous declaration exists.
2620     type_name := incomplete ;
2621     TO_XREF(incomplete) ;
2622     if type_of(incomplete)  incp_types then
2623         errmsg('Invalid redeclaration of ', id,'8.3') ;
2624         type_name := 'any' ;
2625     end if ;
2626
2627     if type_of(incomplete) = 'incomplete' then
2628         if scope_of(incomplete) ≠ scope_name then
2629             errmsg('Incomplete type definition must be completed ',
2630                 'in the same scope in which it first appears',
2631                                 '3.8') ;
2632         type_name := 'any';
2633         end if ;
2634         (forall [-,a_t] ∈ declared(scope_name) +
2635             (if nature(scope_name) = 'private_part'
2636                             then declared(open_scopes(2)) else {} end)
2637                         | (is_type(a_t) and is_access(a_t) and
2638                         base_type(designated_type(a_t)) = type_name))
2639             -- access type is now dereferenceable.
2640             misc_type_attributes(a_t) less:= 'incomplete' ;
2641         end forall ;
2642     else
2643         -- Full declaration for private type. Save visible declaration in
2644         -- private decls for this package, so that full declaration can
2645         -- be installed.
2646         package_name := open_scopes(2) ;
2647         if scope_of(incomplete) ≠ package_name then
2648             errmsg('Invalid context for redeclaration of ',
2649                                             'private type','7.4') ;
2650             type_name := 'any' ;
2651         else
2652 --
2653         private_decls(package_name)(type_name) := SYMBTABF(type_name) ;
2654          end if ;
2655
2656         if is_generic_type(incomplete) then
2657             errmsg('Generic private type ' + id,
```

```
2658                    ' cannot have declaration in private part','12.1') ;
2659              type_name := 'any' ;
2660          end if ;
2661      end if ;
2662 end if ;
2663
2664 return type_name ;
2665
2666 end proc find_type_name ;
2667
2668
```

build_type

```
2670 proc build_type(type_name, opt_disc, type_def) ;
2671
2672
2673
2674 -- For scalar types, both generic and non-generic, this procedure  simply
2675 -- constructs the symbol table entry on the basis of the type definition.
2676 -- Enumeration  types, array  types and  derived  types  require  further
2677 -- processing. They generate additional symbol table entries for literals
2678 -- and anonymous types.
2679
2680 kind := AST(type_def)(1) ;
2681
2682 case kind of
2683
2684 ('enum'):          new_enum_type(type_name, type_def);
2685
2686 ('int_type'):      new_integer_type(type_name, type_def) ;
2687
2688 ('float_type'):    new_floating_type(type_name, type_def) ;
2689
2690 ('fixed_type'):    new_fixed_type(type_name, type_def) ;
2691
2692 ('array_type'):    new_array_type(type_name, type_def);
2693
2694 ('record'):        record_decl(type_name, opt_disc, type_def) ;
2695
2696 ('derived_type'):  derived_type(type_name, type_def) ;
2697
2698 ('access_type'):   [-, desig_type] := adasem(type_def) ;
2699                    SYMBTAB(type_name) := ['access', type_name, desig_type];
2700                    new_agg_or_access('access', type_name) ;
2701 --
2702  end case ;
2703 -- The new type inherits the root type and other attributes of its parent
2704
```

```
2705 parent := type_of(type_name) ;
2706 root_type(type_name) := root_type(parent) ? parent ;
2707 misc_type_attributes(type_name) :=  misc_type_attributes(parent) ;
2708 --
2709 if (l := private_kind(parent)) ≠ om then
2710    misc_type_attributes(type_name) :=
2711                        (misc_type_attributes(type_name) ? {}) with l ;
2712 end if ;
2713 --
2714
2715 end proc build_type;
2716
2717
```

ais_type

```
2719 proc ais_type(type_name) ;
2720
2721 -- This utility formats a type declaration for the interpreter. It
2722 -- is called for type declarations and for anonymous type processing
2723 -- (see procedure new_type_list ). Note that in the case of types
2724 -- for which no specific reformating is provided here, the routine
2725 -- returns the symbol table entry for the type, which is thus assumed
2726 -- to already be in the format the interpreter requires.
2727
2728
2729  case nature(type_name) of
2730
2731    ('array'):
2732      ais_rep :=  [ 'array_former', index_types(type_name),
2733                                          component_type(type_name)] ;
2734
2735    ('record'):
2736      -- Anonymous types generated during the elaboration of component
2737      -- declarations are emitted inside the record declaration. They
2738      -- may depend on discriminants and must appear in the proper scope.
2739
2740      anon := new_type_list() ;
2741      NEWTYPES with:= [ ] ;
2742
2743      ais_rep := ['record_former', [anon + invariant_part(type_name),
2744                                          variant_part(type_name)]];
2745    ('type', 'task_type', 'task_type_spec') :                    --
2746      ais_rep := type_of(type_name);
2747      if ais_rep = '$FIXED' then
2748         ais_rep := ['fixed_former'] + signature(type_name)(2..) ;
2749      end if ;
2750    ('enum') :
2751      [-,lo, hi] := signature(type_name) ;
```

```
2752      enum_map := { [v, nam]: [nam, v] ∈ literal_map(type_name) } ;
2753      ais_rep := ['enum_former', lo(2), hi(2), enum_map] ;
2754
2755    ('subtype'):
2756      if is_array(type_name) then
2757        ais_rep := ['subtype', base_type(type_name),
2758                                  ['index', index_types(type_name)]] ;
2759
2760      elseif is_access(type_name) then
2761          ais_rep := ['access_former', designated_type(type_name)];
2762
2763      else
2764         [-, typ, sig] := SYMBTAB(type_name) ;
2765          if sig = signature(typ) then          -- subtype is redundant
2766             ais_rep := typ;
2767          else
2768             ais_rep := ['subtype', typ, sig] ;
2769          end if;
2770      end if ;
2771
2772    ('access'):
2773      ais_rep := ['access_former', designated_type(type_name)];
2774
2775    else
2776      ais_rep := SYMBTAB(type_name) ;
2777
2778  end case ;
2779  if (pr := private_ancestor(type_name)) ≠ om and pr ≠ type_name then
2780     return ['delayed_type', type_name, ais_rep, pr] ;
2781  else
2782     return ['type', type_name, ais_rep] ;
2783  end if ;
2784 end proc ais_type ;
2785
```

subtype_decl

```
2787 proc subtype_decl(id,subt);
2788
2789
2790
2791 -- Process a subtype declaration. id is the source id of the new
2792 -- entity, and subt is the subtype description.If the subtype indication
2793 -- does not include a constraint, subt is either an anonymous array, or a
2794 -- subtype indication that fucntions as a renaming of a type. In that
2795 -- case the new id denotes the same entity, and does not needs a new
2796 -- symbol table entry, except that for conformance purposes it is not
2797 -- equivalent to the original type. For now we only introduce a new sub-
2798 -- type in the case of scalar types.
```

```
2799
2800    name := find_new(id) ;
2801    if is_identifier(subt) then
2802        if is_array(subt) and is_anonymous(subt) then
2803            SYMBTAB(name) := SYMBTAB(subt) ;
2804
2805        elseif is_scalar_type(subt) then
2806            SYMBTAB(name) := SYMBTAB(subt) ;
2807            nature(name) := 'subtype' ;
2808        else
2809            declared(scope_name)(id) := subt ;
2810            return [ ] ;                        -- no declaration needed
2811        end if ;
2812
2813    else
2814        -- an explicit constraint on a type mark.
2815        SYMBTAB(name) := subt ;
2816    end if;
2817
2818    parent := type_of(name) ;
2819    root_type(name) := root_type(parent) ;
2820    misc_type_attributes(name) := misc_type_attributes(parent) ;
2821
2822    return [ais_type(name)] ;
2823
2824 end proc subtype_decl;
```

make_subtype

```
2826 proc make_subtype(subtype_indic) ;
2827
2828
2829
2830
2831 -- Process a subtype indication.
2832
2833 [name, constraint] := subtype_indic ;
2834
2835
2836 type_mark := find_old(name) ;
2837
2838 if type_mark = '?' then
2839    return [ ] ;
2840 end if ;
2841
2842
2843 if not is_identifier(type_mark) or not is_type(type_mark) then
2844
2845    if type_of(type_mark) ≠ 'any' then
```

```
2846          errmsg('Invalid type_mark in subtype indication','','3.3') ;
2847      -- else it is an undeclared identifier, and was reported as
2848      -- such already.
2849      end if ;
2850      return 'any' ;
2851
2852  elseif base_type(type_mark) ∈ open_scopes                    --
2853  and ( not is_task_type(type_mark)                            --
2854        or original_name(type_mark) =
2855           (if is_identifier(name) then name
2856                                   elseif name(1) = '.' then name(3)
2857                                   else name end))
2858  then
2859      -- Component of record is subtype of record type itself, or task type
2860      -- is used within its own body, rather than a defined subtype.
2861
2862      errmsg('invalid use of type ' + original_name(type_mark) ,
2863                              ' within its definition or body', '3.3,9.1');
2864      return type_mark ;
2865
2866  elseif constraint = [ ] then
2867      check_incomplete(type_mark) ;
2868      return type_mark ;
2869
2870  else
2871      -- If the type is a access type, the constraint applies to the type
2872      -- being accessed. We create the corresponding subtype of it, promote
2873      -- it to an anonymous type, and return an access to it.
2874
2875      if (is_acc:= is_access(type_mark)) then
2876         acc_t := type_mark ;
2877         type_mark := designated_type(type_mark) ;
2878      end if ;
2879
2880
2881      if is_acc
2882      and not is_array(type_mark) and not is_record(type_mark) then
2883          errmsg('Invalid constraint on access type','','3.8') ;
2884          return type_mark ;
2885      end if ;
2886
2887       case nature(type_mark) of
2888
2889      ('type'):   if is_scalar_type(type_mark) then
2890                     subtype := constrain_scalar(type_mark, constraint) ;
2891                  else       -- Private type with discriminants, hopefully.
2892                     subtype := constrain_record(type_mark, constraint) ;
2893                  end if ;
2894
```

```
2895    ('enum'):    subtype := constrain_scalar(type_mark, constraint) ;
2896
2897    ('array') : subtype := constrain_array(type_mark, constraint) ;
2898
2899    ('record'): subtype := constrain_record(type_mark, constraint) ;
2900
2901    ('subtype'): if is_array(type_mark) or is_record(type_mark) then
2902                      errmsg('Invalid subtype indication: ',
2903                                  'type is already constrained',
2904                             '3.6.1, 3.7.2') ;
2905                      subtype := SYMBTAB(type_mark) ;
2906                 else
2907                      subtype := constrain_scalar(type_mark,constraint);
2908                 end if ;
2909
2910    else
2911         errmsg('Invalid type mark in subtype indication: ',
2912                                  original_name(type_mark),'3.3, 3.6.1') ;
2913
2914         subtype := type_mark ;
2915    end case ;
2916
2917 end if ;
2918
2919
2920 return if is_acc then ['subtype', acc_t, promote_subtype(subtype) ]
2921                       else  subtype end ;
2922
2923 end proc make_subtype ;
```

derived_type

```
2925 proc derived_type(type_name, def_node) ;
2926
2927 const unconstrained_natures =
2928         {'enum', 'array', 'record', 'task_type', 'task_type_spec'} ;
2929
2930
2931 [-, subtype] := adasem(def_node) ;
2932 if is_identifier(subtype) then
2933     old_type := subtype ;
2934     constraint := signature(old_type) ;     -- Inherited by new type
2935 else
2936     [-, old_type, constraint] := subtype ;  -- Subtype indication.
2937 end if ;
2938
2939 if old_type = subtype and
2940   (nature(subtype) ∈  unconstrained_natures
2941   or is_access(subtype) or type_of(subtype) ∈ priv_types)
```

```
2942 then
2943     new_type := type_name ;
2944
2945 else
2946     -- If the derived type definition includes a constraint, or if the
2947     -- old type is constrained, we first derive an anonmyous type, and
2948     -- then construct a subtype of it.
2949     new_type := named_type(original_name(type_name) + "'base') ;
2950     top(NEWTYPES) with:= new_type ;
2951 end if ;
2952
2953 parent_type := base_type(old_type) ;
2954 -- A derived type defined in a package specification cannot be used for
2955 -- further derivation until the end of its visible part. This means that
2956 -- derivation in the private part of the package is allowed.
2957
2958 if is_derived_type(parent_type)
2959   and (exists i ∈ [1..#open_scopes] |
2960             open_scopes(i) = scope_of(parent_type))
2961   and nature(open_scopes(i)) ∈ ['package_spec', 'generic_package_spec']
2962   and nature(open_scopes(i-1 max 1)) ≠ 'private_part'
2963 or  is_incomplete_type(old_type)
2964 then
2965     errmsg('premature derivation of derived or private type ',
2966                           original_name(parent_type), '3.4, 7.4.1') ;
2967 end if ;
2968
2969 case nature(parent_type) of
2970
2971   ('type'):
2972             new_def := ['type', old_type, signature(parent_type)] ;
2973
2974   ('array'):
2975             array_info := signature(old_type) ;
2976             new_unconstrained_array(new_type, array_info ) ;
2977             new_def := SYMBTAB(new_type) ;
2978
2979
2980   -- A derived record type has the same fields and types as the parent.
2981   -- All we need to do is introduce an aggregate under the new type mark.
2982   -- The declaration may have a discriminant part, in which case they
2983   -- must conform to the discriminants of the parent type. We assume that
2984   -- the discriminant names, types, and default values must be the same.
2985
2986   ('record'):
2987             if is_record(new_type) then
2988                 dl1 := discriminant_list(new_type) ;
2989                 dl2 := discriminant_list(parent_type) ;
2990                 if #dl1 ≠ #dl2 or
```

```
2991                exists i ∈ [1..#dl1] |
2992                  (type_of(d1 := dl1(i)) ≠ type_of(d2 := dl2(i))
2993                  or default_expr(d1) ≠ default_expr(d2)
2994                  or original_name(d1) ≠ original_name(d2)) then
2995                   errmsg('discriminant mismatch in declaration',
2996                                                        '','3.8');
2997              end if ;
2998          end if ;
2999          new_def := [nature(parent_type), new_type,
3000                          record_declarations(parent_type)]; --
3001          declared(new_type) := declared(parent_type) ;
3002          new_agg_or_access('aggregate', new_type);
3003
3004 -- A derived enumeration type has the literals of its parent, but these
3005 -- are marked as derived, to enforce the overloading rules.
3006
3007 ('enum'):
3008          lit_map := literal_map(parent_type) ;
3009          (forall nam ∈ domain lit_map)
3010              new_lit_name :=
3011              chain_overloads(nam, ['literal',new_type, [ ], nam]);
3012              ALIAS(new_lit_name) := nam ;
3013              -- NOT QUITE EXACT: SHOULD BE UNIQUE NAME OF PARENT.
3014          end forall ;
3015          bounds := SIGNATURE(old_type) ;
3016          new_def := ['enum', new_type, bounds, lit_map] ;
3017
3018 ('access'):
3019          new_def := SYMBTAB(old_type);
3020          new_agg_or_access('access', new_type) ;
3021
3022 ('task_type', 'task_type_spec'):
3023          new_def := SYMBTAB(old_type) ;
3024         new_def(1) := 'task_type' ;              --
3025          declared(new_type) := declared(parent_type) ;
3026
3027  else
3028          errmsg('cannot obtain derived type from ',
3029                              original_name(old_type),'3.4') ;
3030          new_def := symbtab(old_type) ;  -- To have something.
3031
3032 end case ;
3033
3034 SYMBTAB(new_type) := new_def ;
3035 root_type(new_type) := root_type(old_type) ;                    --
3036 if nature(parent_type) ≠ 'enum' then
3037   misc_type_attributes(new_type) :=
3038      if (l := private_kind(old_type)) ≠ om then {l}          --
3039                                          else { } end ;
```

```
3040 -- otherwise the attribute is the literal map.
3041 end if ;
3042
3043 if type_name ≠ new_type then
3044     SYMBTAB(type_name) := ['subtype', new_type, constraint] ;
3045 end if ;
3046
3047 derive_subprograms(old_type, new_type) ;
3048
3049 end proc derived_type ;
3050
```

is_derived_type

```
3052 proc is_derived_type(mark) ;
3053 --
3054 return base_type(mark) ≠ root_type(mark) and not is_generic_type(mark);
3055
3056 -- Incomplete for composite types.
3057
3058 end proc is_derived_type ;
```

derive_subprograms

```
3060 proc derive_subprograms(original_type, d_type) ;
3061
3062 -- If a type is defined in a package specification, and derived after it,
3063 -- then the derived type inherits all subprograms defined in the package
3064 -- that mention the parent type.
3065 -- We introduce in the symbol table the new subprograms with the derived
3066 -- signature, but obviously emit no code for them. We produce instead a
3067 -- mapping from the inherited subprogram to its ancestor, and replace
3068 -- the name at the point of call, in macro-like fashion.
3069 -- We must verify that the original type is defined in the visible part
3070 -- of a package, that its base type is in the same package , and that the
3071 -- derived type is not defined in the same package specification.
3072 -- If the type is derived in the private part of the package, and the
3073 -- parent type is a private type which is a first-named subtype, then its
3074 -- base type is defined in the scope of the private part, and this must
3075 -- checked for separately.
3076
3077 parent_scope := scope_of(original_type) ;
3078
3079 if nature(parent_scope) ∈ {'package', 'package_spec'}
3080 and
3081 -- The type must be defined in the package, and not be a subtype or a
3082 -- renaming of a type defined outside of the package.
3083 ((( base_scope := scope_of(base_type(original_type))) = parent_scope)
3084     or
```

```
3085    nature(base_scope) = 'private_part'
3086    and base_scope = scope_name )
3087 and
3088  (v := visible(parent_scope)) ≠ om and
3089  (exists [-,t] ∈ v | t = original_type)          --
3090  and
3091  -- We are no longer in the visible part of the package specification.
3092  (nature(parent_scope) = 'package' or parent_scope ≠ scope_name)
3093 then                                            -- calculate inheritance.
3094    (forall [id,obj] ∈ v | (nat := nature(obj)) ∈
3095        {'procedure', 'function', 'procedure_spec', 'function_spec'})
3096       derive1_subprogram(id, obj, original_type, d_type, obj) ;
3097    end forall ;
3098
3099 elseif parent_scope ≠ 'STANDARD#0' then
3100
3101    -- If the original type is a derived type, its derived subprograms
3102    -- are further derived. If such exist, they are aliased subprograms
3103    -- declared in the same scope as the parent type.
3104
3105    (forall [id,obj] ∈ declared(parent_scope) |
3106       (nat := nature(obj)) ∈
3107           {'procedure', 'function', 'procedure_spec', 'function_spec'}
3108        and is_derived_subprogram(obj))
3109
3110       derive1_subprogram(id, obj, original_type, d_type, ALIAS(obj)) ;
3111    end forall ;
3112
3113 end if ;
3114
3115 end proc derive_subprograms ;
3116
```

derive1_subprogram

```
3118 proc derive1_subprogram(id, obj, original_type, d_type, parent_subp) ;
3119
3120 -- obj is a subprogram declared in the same scope as original_type. If
3121 -- some type in its profile is compatible with original_type, produce
3122 -- a derived subprogram by replacing the original_type with the derived
3123 -- one, and introduce a subprogram declaration with this new profile.
3124 -- The parent subprogram is either obj itself, or ALIAS(obj) if obj is
3125 -- already a derived subprogram.
3126
3127 new_sig := signature(obj) ;
3128 new_typ := type_of(obj) ;
3129
3130 (forall [-,-,typ] = new_sig(i))
3131    if compatible_types(typ, original_type) then          --
```

```
3132          new_sig(i)(3) := d_type ;
3133      end if ;
3134  end forall ;
3135
3136  if compatible_types(new_typ, original_type) then              --
3137      new_typ := d_type ;
3138  end if ;
3139
3140  if new_sig ≠ signature(obj) or new_typ ≠ type_of(obj) then
3141      -- subprogram is derived. Insert it in current scope, with
3142      -- the new signature, and recall its lineage.
3143      -- Its nature is a subprogram, not a spec, because no body
3144      -- will be defined for it.
3145
3146      nat := nature(obj) ;
3147      rmatch(nat, '_spec') ;
3148  --
3149      id := original_name(obj) ;
3150      new_proc:= chain_overloads(id,[nat, new_typ, new_sig, parent_subp]);
3151      if new_proc ≠ '' then                  -- There is no explicit sub-
3152          ALIAS(new_proc) := parent_subp ;    -- program with that name.
3153      end if ;
3154  end if ;
3155
3156  end proc derive1_subprogram ;
3157
```

is_derived_subprogram

```
3159  proc is_derived_subprogram(name) ;
3160
3161
3162  return (s := ALIAS(name)) ≠ om
3163      and original_name(s) = original_name(name) ;
3164
3165  end proc is_derived_subprogram ;
```

new_enum_type

```
3167  proc new_enum_type(type_name, def_node) ;
3168
3169  [-, literals] := adasem(def_node) ;
3170
3171  lit_map := {} ;
3172  new_lit := {} ;
3173
3174  (forall i ∈ [1..#literals])
3175      -- insert each literal in symbol table, as an overloadable identifier
3176      -- Each enumeration type is mapped  into a sequence  of integers, and
```

```
3177    -- each literal is defined as a constant with integer value.
3178
3179    new_lit(i) := chain_overloads(literals(i),
3180                                  ['literal', type_name, [ ] ]) ;
3181    lit_map(literals(i)) := i-1 ;
3182 end forall ;
3183
3184 SYMBTAB(type_name) := ['enum', type_name, ['range', ['ivalue', 0],
3185                                  ['ivalue', #literals-1]], lit_map] ;
3186
3187 end proc new_enum_type ;
```

new_integer_type

```
3189 proc new_integer_type(type_name, def_node) ;
3190
3191
3192
3193
3194 -- Create a new integer, and apply the constraint to obtain a
3195 -- subtype of it.
3196
3197 [-,constraint] := adasem(def_node) ;
3198 newtype := anonymous_type() ;
3199 SYMBTAB(newtype) := SYMBTAB('INTEGER') ;
3200 root_type(newtype) := 'INTEGER' ;                          --
3201 [-,lo, hi] := constraint ;
3202 lo := check_type('integer_type', lo) ;
3203 hi := check_type('integer_type', hi) ;
3204
3205 lo := specialize(lo, 'INTEGER') ;          -- if universal
3206 hi := specialize(hi, 'INTEGER') ;
3207
3208 if not is_value(lo) or not is_value(hi) then
3209    errmsg('bounds in an integer type definition must be static',
3210                                        '','3.5.4') ;
3211 elseif not is_integer(lo(2)) or not is_integer(hi(2)) then
3212    errmsg('bounds in an integer type definition must be ',
3213                'of some integer type', '3.5.4') ;
3214 end if ;
3215
3216 SYMBTAB(type_name) := ['subtype', newtype, ['range', lo, hi]] ;
3217 end proc new_integer_type;
3218
```

new_floating_type

```
3220 proc new_floating_type(type_name, def_node) ;
3221
3222 -- Process a floating point declaration.
3223 [-, accuracy] := adasem(def_node) ;
3224 [-, precision,opt_range] := accuracy ;
3225
3226 -- The range constraint is optional. If absent, the range is that of
3227 -- the parent type. If present, the bounds need not be of the same
3228 -- type, but of some real type, and static. Their resolution is done
3229 -- in procedure real-bound.
3230
3231 newtype := anonymous_type() ;
3232 SYMBTAB(newtype) := SYMBTAB('FLOAT') ;
3233 root_type(newtype) := 'FLOAT' ;                              --
3234 SYMBTAB(type_name) := SYMBTAB(newtype) ;          -- by default.
3235 precision := check_type('integer_type', precision) ;
3236 p_type := expr_type(precision) ;
3237
3238 if precision = [ ] then                                  -- Type error.
3239     return ;
3240 elseif not is_value(precision) then
3241     errmsg('Expect static expression for ', 'digits', '3.5.7') ;
3242     return ;
3243 elseif p_type = 'universal_integer' then
3244     precision := specialize(precision, 'INTEGER') ;
3245 end if ;
3246
3247 digits := precision(2) ;
3248 if digits < 1 then
3249     errmsg('Invalid digits value in ', ' real type declaration',
3250                              '3.5.7') ;
3251     return ;
3252 elseif digits > ADA_REAL_DIGITS then
3253     errmsg('Precision not supported ',' by implementation','none') ;
3254     return ;
3255 end if ;
3256
3257
3258 if opt_range = [ ] then                          -- Use system FLOAT.
3259     constraint := signature('FLOAT') ;
3260     constraint(4) := precision ;-- and stated precision.
3261 else
3262     [-, lo, hi] := opt_range ;
3263     lo := real_bound(lo, 'FLOAT') ;
3264     if lo = [ ] then return ; end if ;
3265
```

```
3266    hi := real_bound(hi, 'FLOAT') ;
3267    if hi = [ ] then return ;.end if ;
3268    constraint := ['digits',lo,hi, precision] ;
3269 end if ;
3270
3271 SYMBTAB(type_name) := ['subtype', newtype, constraint] ;
3272
3273 end proc new_floating_type;
3274
```

new_fixed_type

```
3276 proc new_fixed_type(type_name, def_node) ;
3277
3278 -- Process a fixed point declaration. Similar to floating case.
3279 [-, accuracy] := adasem(def_node) ;
3280 [-, precision,opt_range] := accuracy ;
3281
3282 newtype := anonymous_type() ;
3283 SYMBTAB(newtype) := SYMBTAB('$FIXED') ;
3284 root_type(newtype) := '$FIXED' ;                          --
3285 SYMBTAB(type_name) := SYMBTAB(newtype) ;
3286
3287 precision := check_type('real_type', precision) ;
3288 p_type := expr_type(precision) ;
3289
3290 if precision = [ ] then                                   -- Type error.
3291    return ;
3292 elseif not is_value(precision) then
3293    errmsg('Expect static expression for ', 'delta', '3.5.7') ;
3294    return ;
3295 elseif root_type(p_type) = 'FLOAT' then
3296    precision := ['ivalue', rat_frr(precision(2))] ;
3297 end if ;
3298
3299 if opt_range = [ ] then
3300    errmsg ('Missing range in Fixed type declaration','','3.5.9');
3301    return ;
3302 else
3303    [-, lo, hi] := opt_range ;
3304    lo := real_bound(lo, '$FIXED') ;
3305    if lo = [ ] then return ; end if ;
3306
3307    hi := real_bound(hi, '$FIXED') ;
3308    if hi = [ ] then return ; end if ;
3309
3310    constraint := ['delta', lo, hi, precision] ;
3311 end if ;
3312
```

```
3313 SYMBTAB(type_name) := ['subtype', newtype, constraint] ;
3314 signature(newtype) := constraint ;                    --
3315
3316 end proc new_fixed_type ;
3317
```

real_bound

```
3319 proc real_bound( bound, kind) ;
3320
3321
3322
3323 -- Verify that the bound of a range constraint in a real type definition
3324 -- is a real type, and convert it to or from universal when needed.
3325
3326 bound := check_type('real_type', bound) ;                --
3327 b_type := expr_type(bound) ;
3328 r_type := root_type(b_type) ;
3329
3330 if not is_value(bound) then
3331     errmsg('Bound in range constraint of type definition ',
3332                                     'must be static','3.5.7, 3.5.9') ;
3333     return [ ] ;
3334 elseif r_type = kind then
3335     return bound ;
3336 --
3337
3338 elseif kind = 'FLOAT' then          -- Fixed or universal bound.
3339     return ['ivalue', rat_tor(bound(2),ADA_REAL_DIGITS)] ;
3340
3341 elseif kind = '$FIXED' then
3342    if r_type = 'FLOAT' then
3343          return ['ivalue', rat_frr(bound(2))] ;
3344    else  return bound ;
3345    end if ;
3346 end if ;
3347 --
3348 end proc real_bound ;
3349
```

constrain_scalar

```
3351 proc constrain_scalar(type_mark, constraint) ;
3352
3353
3354
3355 -- Constrain a discrete type or a floating type .
3356
3357     constr_type := constraint(1);
```

```
3358    base_mark := base_type(type_mark) ;
3359
3360    case constr_type of
3361
3362    ('range'):
3363        -- In this case, the bounds expressions have not been
3364        -- type-checked yet. Do it now that the desired base
3365        -- type is known.
3366
3367          bt := root_type(type_mark) ;
3368          if not is_scalar_type(bt) then
3369                errmsg('Invalid RANGE constraint for type',
3370                                ",'3.3, 3.6.1') ;
3371                return type_mark ;
3372          end if ;
3373          [-,lo,hi] := constraint ;
3374          if bt ∈ {'FLOAT', '$FIXED'} then
3375              -- Range constraint on floating type. Use
3376              -- precision that was specified for type itself.
3377
3378              [constr, -, -, precision] := signature(type_mark) ;
3379              new_c := [constr, check_type(base_mark,lo),
3380                          check_type(base_mark,hi), precision] ;
3381
3382          else          -- Discrete type.
3383              new_c :=  ['range', check_type(base_mark, lo),
3384                                  check_type(base_mark, hi)] ;
3385          end if ;
3386
3387    ('digits', 'delta') :
3388          kind := if constr_type = 'digits' then 'FLOAT'
3389                                              else '$FIXED' end;
3390          if root_type(type_mark) /= kind then
3391              errmsg('Invalid constraint for type',",'3.3') ;
3392              return type_mark ;
3393          end if ;
3394
3395          if is_generic_type(base_mark) then
3396              errmsg('accurracy constraint cannot depend on ',
3397                      'a generic type', '12.1') ;
3398              return type_mark ;
3399          end if ;
3400
3401          [-, precision, [-, lo, hi]] := constraint ;
3402          precision :=
3403              check_type(if kind='FLOAT' then 'INTEGER'
3404                                          else '$FIXED' end,
3405                      precision);
3406          if precision /= [ ] then
```

```
3407                if kind= 'FLOAT' then
3408                   if not is_value(precision) then
3409                   errmsg('Expect static expression for',
3410                              'DIGITS','3.5.7') ;
3411                   else digits := precision(2) ;
3412                     if digits < 1 then
3413                         errmsg('Invalid value for ','DIGITS','3.5.7') ;
3414                     elseif digits > ADA_REAL_DIGITS then
3415                          errmsg('Implementation only supports ',
3416                                       'digits 6','none') ;
3417                     end if ;
3418                   end if ;
3419               end if ;
3420           end if ;
3421
3422           if lo /= om then
3423               lo := check_type(base_mark, lo) ;
3424               hi := check_type(base_mark, hi) ;
3425           else
3426               -- Only the precision was given in the constraint.
3427               -- Obtain the bounds from the type itself.
3428
3429               [-,lo,hi] := signature(type_mark) ;
3430           end if ;
3431
3432           new_c := [constr_type, lo, hi, precision] ;
3433
3434   ('name'):
3435
3436       -- The constraint is given by a RANGE attribute.
3437      -- We get the bounds of the range to construct the new
3438       -- subtype.
3439      range_expr := find_old(constraint) ;
3440      if not is_range_attribute(range_expr) then
3441          errmsg('Invalid expression for range constraint','','3.3');
3442          return type_mark ;
3443      else
3444          range_expr := check_type(base_mark, range_expr) ;
3445          new_c := apply_range(range_expr) ;                    --
3446      end if ;
3447
3448   else
3449       if #constraint = 1 and constraint(1)(1) = 'name' then
3450          -- Case of an index constraint given by a subtype name.
3451          subtype := find_old(constraint(1)(2)) ;
3452          if not is_type(subtype) or
3453              not compatible_types(type_mark, subtype) then    --
3454              errmsg('Invalid subtype name in  constraint',
3455                              '','3.3, 3.6.1');
```

```
3456                return [ ] ;
3457            else
3458                -- TBSL. What should this be ?
3459              pass ;
3460            end if ;
3461        else
3462            errmsg('invalid constraint ',' for type','3.3');
3463            new_c := [ ] ;
3464        end if ;
3465    end case;
3466 --
3467    -- Verify that a discriminant does not appear in a scalar constraint.
3468    -- This must be special-cased because discriminants are otherwise
3469    -- allowed to appear in index and discriminant constraints, and in
3470    -- initial values, i.e. arbitrary expressions.
3471    check_discriminant(new_c) ;
3472 --
3473    return ['subtype', type_mark, new_c] ;
3474
3475 end proc constrain_scalar ;
```

apply_range

```
3477 proc apply_range(range_expr) ;
3478
3479 -- A'RANGE is equivalent to A'FIRST..A'LAST. When the range attribute
3480 -- is used as a constraint, the bounds are expressed according to the
3481 -- above equivalence. This is not strictly correct if the elaboration
3482 -- of A has side-effects, but we ignore this detail for now.
3483
3484 if range_expr = [ ] then                    -- previous type error.
3485    return signature('INTEGER') ;            -- format is right.
3486 end if ;
3487 if range_expr(1) = 'qual_range' then        -- discard check for now
3488    [-, typ, range_expr] := range_expr ;     -- (could apply to each)
3489 end if ;
3490
3491 [-, attr, arg1, arg2, typ] := range_expr ;
3492
3493 -- The attribute is either O_RANGE or T_RANGE, according as arg1 is an
3494 -- object or a type. FIRST and LAST must be marked accordingly.
3495 attr_prefix := attr(1..2) ;
3496 return ['range', ['"', attr_prefix + 'FIRST', arg1, arg2, typ],
3497                  ['"', attr_prefix + 'LAST',  arg1, arg2, typ] ];
3498
3499 end proc apply_range ;
```

array_typedef

```
3501 proc array_typedef(index_list, element_type) ;
3502
3503
3504
3505 -- Validate an array type definition.
3506
3507 if exists indx ∈ index_list | is_tuple(indx) and indx(1) = 'box'
3508 then
3509    -- Unconstrained array . Verify that all indices are unconstrained.
3510    if exists indx1 ∈ index_list|not is_tuple(indx1) or indx1(1) /='box'
3511      then
3512          errmsg('Constraints apply to all indices ', 'or none','3.6.1') ;
3513    end if ;
3514
3515 end if ;
3516
3517 if is_unconstrained(element_type)                          --
3518 then
3519       errmsg('Unconstrained element type in array declaration',
3520                                        '', '3.6.1, 3.7.2') ;
3521 end if ;
3522 check_fully_declared2(element_type) ;                      --
3523
3524 return ['array', [index_list, element_type] ] ;
3525
3526 end proc array_typedef;
3527
```

new_array_type

```
3529 proc new_array_type(array_type, def_node) ;
3530
3531
3532
3533 -- This procedure is called whenever an array type is created.
3534 -- For each new array type we create a corresponding sequence type,
3535 -- which is an unconstrained array. Unconstrained array types have
3536 -- nature 'array', while constrained arrays have nature 'subtype'.
3537
3538 [-, array_info] := adasem(def_node) ;
3539 [indx, comp] := array_info ;
3540 if is_tuple(indx(1)) and indx(1)(1) = 'box' then
3541    -- Unconstrained array definition. In this case, introduce only the
3542    -- unconstrained type, and ignore the actual array type.
3543    indx := [i(2) : i ∈ indx] ;
3544    array_info := [indx, comp] ;
```

```
3545     new_unconstrained_array(array_type, array_info) ;
3546 else
3547     new_constrained_array(array_type, array_info) ;
3548 end if ;
3549 --
3550  end proc new_array_type;
```

new_unconstrained_array

```
3552 proc new_unconstrained_array(sequence_type, array_info) ;
3553
3554 [indx, comp] := array_info ;
3555
3556 SYMBTAB(sequence_type) := ['array', sequence_type, array_info] ;
3557 -- Mark the type as limited if the component type is.
3558 misc_type_attributes(sequence_type) :=
3559     if is_access(comp) then {}                                    --
3560     elseif (l:= private_kind(comp)) /= om then {l}
3561     else {}
3562 end ;
3563 root_type(sequence_type) := sequence_type ;
3564
3565 -- For each unconstrained array type, we introduce an instance of the
3566 -- 'aggregate' pseudo-operator for that array.
3567
3568 new_agg_or_access('aggregate', sequence_type) ;
3569 end proc ;
```

new_constrained_array

```
3571 proc new_constrained_array(array_type, array_info) ;
3572
3573 -- Construct meaningful name for anonymous parent type.
3574 nam := original_name(array_type) ;
3575 if nam = '' then nam := 'anonymous_array' ; end if ;
3576
3577 sequence_type := nam + ''base' + str newat  ;
3578 declared(scope_name)(sequence_type) := sequence_type ;
3579 scope_of(sequence_type) := scope_of(array_type) ;          --
3580 -- emit sequence type as an anonymous type. It is used in aggregates
3581 -- that are assigned to slices, and in other unconstrained contexts.
3582 -- (This should only be needed for one dimensional arrays).
3583 top(NEWTYPES) with:= sequence_type ;
3584 new_unconstrained_array(sequence_type, array_info) ;
3585
3586 -- Make the actual array type into a subtype of the unconstrained one
3587
3588 SYMBTAB(array_type) := ['subtype', sequence_type, array_info] ;
3589 misc_type_attributes(array_type) := misc_type_attributes(sequence_type);
```

```
3590 root_type(array_type) := sequence_type ;
3591
3592 end proc ;
```

anonymous_array

```
3594 proc anonymous_array(node) ;
3595
3596
3597
3598 -- Process an array definition in an object or constant declaration.
3599 -- The node is an array_type node.
3600
3601 typ := find_new('anon' + str newat) ;              -- Create a name for it
3602 new_array_type(typ, node) ;                  -- elaborate definition
3603 top(NEWTYPES) with:= typ ;                        -- Insert into type stack
3604
3605 return typ ;
3606
3607 end proc anonymous_array ;
3608
```

constrain_array

```
3610 proc constrain_array(type_mark, constraint) ;
3611
3612
3613
3614 -- Apply index constraints to array type.
3615
3616 if not can_constrain(type_mark) then
3617     errmsg('Array type is already constrained','','3.6.1') ;
3618     return type_mark ;
3619 end if ;
3620
3621 if #constraint /= no_dimensions(type_mark) then
3622     errmsg('Incorrect no. of index constraints for type ',
3623                                 original_name(type_mark),'3.6.1') ;
3624     return type_mark ;
3625 end if ;
3626
3627 if constraint = [.] then
3628     new_array := type_mark;
3629 else
3630     -- apply constraints to each index type.
3631     indices := index_types(type_mark) ;
3632     new_indices := [constrain_index(indices(i), constraint(i)):
3633                                                 i ∈ [1..#constraint]] ;
3634 end if;
```

```
3635
3636 new_array := promote_subtype(['subtype', type_mark,
3637                             [new_indices, component_type(type_mark)]]) ;
3638 return new_array ;
3639
3640 end proc constrain_array ;
3641
```

make_index

```
3643 proc make_index(subtype) ;
3644
3645 -- Process and index  in an array declaration,  an entry family declara-
3646 -- tion, or a loop iteration. The index is given by an index declaration
3647 -- ( a 'box' ),  or by a discrete range. The later can be  the name of a
3648 -- discrete type, or a subtype indication.
3649
3650
3651
3652
3653 if is_tuple(subtype) and subtype(1) = 'name' then
3654     -- The discrete range is the name of a discrete (sub)type, or a range
3655     -- attribute (syntactically, a name also).
3656     subtype := find_old(subtype) ;
3657 end ;
3658
3659 if is_identifier(subtype) then
3660     new_sub := subtype ;
3661
3662 elseif subtype(1) = 'box' then
3663     -- Unconstrained index definition. verify that the type_mark is
3664     -- discrete.
3665
3666     type_mark := find_old(subtype(2)) ;
3667     if not is_type(type_mark) or not is_discrete_type(type_mark) then
3668         errmsg('Invalid type mark in subtype definition',
3669             original_name(type_mark), '3.6.1') ;
3670     end if ;
3671     return ['box', type_mark] ;
3672 .
3673 elseif is_range_attribute(subtype) then
3674     range_expr := check_type('discrete_type', subtype) ;
3675     typ := expr_type(range_expr) ;
3676     new_sub := ['subtype', typ, apply_range(range_expr)];   --
3677
3678 elseif subtype(1) = 'subtype' then
3679
3680     -- the index is given by a subtype with a range constraint.
3681
```

```
3682    [-,type_name, constraint] := subtype ;
3683
3684    if type_name = [ ] then
3685       new_sub := check_type('discrete_type', subtype) ;
3686
3687    else                          -- Type name is an identifier.
3688       if constraint = [ ] then          -- Discrete range is a type mark.
3689          new_sub := type_name ;
3690       else                              -- Check bounds of constraint.
3691          new_sub  := check_type(base_type(type_name), subtype) ;
3692       end if ;
3693
3694    end if;
3695 else
3696
3697    errmsg('Invalid expression for index definition','','3.6.1');
3698    return 'any' ;
3699 end if ;
3700
3701 if is_identifier(new_sub) then
3702    typ := new_sub ;
3703 elseif new_sub /= om then                    -- new_sub is a subtype
3704   [-,typ,-] := new_sub ;
3705 end if ;
3706
3707 -- Check that a type for the range was found, and that it is
3708 -- discrete, and generate an anonymous type for it.
3709
3710 if noop_error then
3711    -- Error message was emitted already.
3712    return 'any' ;
3713
3714 elseif typ = om then
3715    errmsg('Invalid bounds expressions in range','','3.3, 3.6.1') ;
3716    return 'any' ;
3717
3718 elseif not is_discrete_type(typ) then
3719    errmsg('expect discrete type  in discrete range','','3.3, 3.6.1') ;
3720    return 'any' ;
3721 end if;
3722
3723 return promote_subtype(new_sub) ;
3724
3725 return sub_name ;
3726
3727
3728 end proc make_index;
3729
3730
```

constrain_index

```
3732 proc constrain_index(index, constraint) ;
3733
3734
3735
3736 -- Process an index constraint in a constrained array declaration.
3737 -- The constraint can be a subtype name, or a range with or without
3738 -- an explicit type mark. The index has been obtained from the signature
3739 -- of the unconstrained array.
3740
3741 base_index := base_type(index) ;
3742 case constraint(1) of
3743
3744    ('name'):
3745      new_index := find_old(constraint(2)) ;
3746
3747      if is_range_attribute(new_index) then
3748        -- Obtain subtype of range, but construct as non-static.
3749          range_expr := check_type('discrete_type', new_index) ;
3750        new_index := expr_type(range_expr) ;
3751          sig := apply_range(range_expr) ;                    --
3752
3753      else                                          -- named subtype.
3754          sig := signature(new_index) ;
3755      end if ;
3756
3757      if not compatible_types(index, new_index) then          --
3758          errmsg('Invalid index constraint for ',
3759          original_name(index),'3.6.1') ;
3760          return base_index ;
3761      else
3762          -- Apply bounds of given subtype to original index type.
3763        new_index := ['subtype', index, sig] ;
3764      end if ;
3765
3766    ('subtype'):
3767      -- The type name in the given constraint must be the same as the
3768      -- original unconstrained index.
3769      [-, type_name, cons] := constraint ;
3770      if base_type(type_name) /= base_index then
3771          errmsg('Invalid index constraint for ',
3772          original_name(index),'3.6.1') ;
3773          return base_index ;
3774      end if ;
3775      new_index := check_type(index,                          --
3776                              ['subtype', type_name, cons]) ;
3777      -- If type_name is not the index itself, the constraint must be
```

```
3778      -- be checked for compatibility against both.
3779      if type_name /= index then
3780         anon_index := promote_subtype(new_index) ;
3781         new_index := ['subtype', index, signature(anon_index)] ;
3782      end if ;
3783    ('range'):
3784      -- The expressions defining the range must be compatible with
3785      -- the unconstrained index.
3786
3787      new_index := check_type(base_index,['subtype',index,constraint]);
3788
3789 else
3790         errmsg('Invalid index constraint for ',
3791         original_name(index),'3.6.1') ;
3792         return base_index ;
3793
3794 end case ;
3795
3796 if new_index /= [ ] then
3797    return promote_subtype(new_index) ;
3798 end if ;
3799
3800 errmsg('Invalid index constraint for ', original_name(index),'3.6.1') ;
3801 return base_index ;
3802
3803 end proc constrain_index ;
3804
```

record_decl

```
3806 proc record_decl(type_name, opt_disc, type_def) ;
3807
3808
3809
3810 -- Records constitute a scope for the component declarations within.
3811 -- The scope is created prior to the processing of these declarations.
3812 -- Discriminants are processed first, so that they are visible when
3813 -- processing the other components. After the discriminants have been
3814 -- processed we set the nature of the type to 'record'.
3815 --
3816
3817 if record_declarations(type_name) /= om then
3818    -- an incomplete or private type declaration was already given for
3819    -- the type, and the discriminants have been declared. Install scope.
3820    newscope(type_name) ;
3821 else
3822    process_discr(type_name, opt_disc) ;
3823 end if ;
3824 nature(type_name) := 'record' ;
```

```
3825 type_of(type_name) := type_name ;                -- its own base type
3826
3827 -- Now process remaining field declarations.
3828 [-, invar_part, var_part] := ADASEM(type_def) ;
3829
3830 invariant_part(type_name) +:= invar_part ;
3831 variant_part(type_name) := var_part ;
3832 declared_components(type_name) := declared(scope_name) ;
3833 --
3834 misc_type_attributes(type_name) := {} ;
3835 if exists [-,comp] ∈ declared(scope_name) |
3836   (not is_access(type_of(comp))                                    --
3837   and  (l := private_kind(type_of(comp))) /= om)                   --
3838 then misc_type_attributes(type_name) with:= l ;
3839 end if ;
3840 --
3841
3842 -- The nature of the record components is given as 'field' while the
3843 -- record is being processed, in order to catch invalid dependencies
3844 -- among component declarations. Reset the nature  of each to 'obj'
3845 -- (except for discriminants of course).
3846
3847 (for [-,n] ∈ declared(scope_name) | nature(n) = 'field')
3848     nature(n) := 'obj' ;
3849 end for ;
3850
3851 popscope() ;                              -- Exit record scope.
3852
3853 -- For each record type we create an aggregate of the corresponding
3854 -- type.
3855 new_agg_or_access('aggregate', type_name) ;
3856
3857
3858 end proc record_decl ;
3859
```

is_record

```
3861 proc is_record(typ) ;
3862
3863 -- This predicate is used to validate selected component notation and
3864 -- the examination of discriminant lists.
3865
3866 return    nature(r := root_type(typ)) = 'record'
3867       or (type_of(r) ∈ incp_types and has_discriminants(r)) ;
3868
3869 end proc is_record ;
3870
```

process_discr

```
3872 proc process_discr(type_name, opt_disc) ;
3873
3874
3875 -- Process discriminants, or reprocess them in a full type declaration.
3876 -- Introduce the record scope . It is exited after the call, in type_decl
3877 -- or record decl.
3878
3879 newscope(type_name) ;
3880
3881    record_declarations(type_name) := [[ ],[ ]] ;
3882
3883    [disc_decls, discr_decl_tree(type_name)] := discr_decl(opt_disc) ;
3884    invariant_part(type_name) := disc_decls ;
3885    disc_names := [ ] +/ [ d(2): d ∈ disc_decls] ;
3886
3887    -- Check that all discriminants have default values, or none.
3888
3889    if exists d1 ∈ disc_names | default_expr(d1) /= om and
3890       exists d2 ∈ disc_names | default_expr(d2)  = om then
3891       errmsg('Incomplete specification of default values ',
3892                                        ' for discriminants','3.7.1') ;
3893    end if ;
3894
3895    discriminant_list(type_name) := disc_names ;
3896
3897    -- Make names of discriminants visible at this point, because they
3898    -- may be used in constraints to access types of the current record
3899    -- type.
3900    declared_components(type_name) := declared(scope_name) ;
3901
3902 end proc process_discr ;
3903
```

discr_decl

```
3905 proc discr_decl(discr_list) ;
3906
3907
3908 -- Process discriminant declarations. Discriminants are processed like
3909 -- variable declarations, except that the type of a discriminant must be
3910 -- discrete, and the nature of a discriminant is, naturally enough
3911 -- 'discriminant' . This insures that discriminants cannot appear on the
3912 -- left of an assignment, nor in expressions. The original tree is also
3913 -- saved to check for conformance upon redeclaration.
3914
3915 decls := [ ] ;
```

```
3916 discr_tree := [ ] ;
3917
3918 (forall discr_spec ∈ discr_list)
3919
3920    [id_list, type_id, opt_init] := discr_spec ;
3921    nam_list := [find_new(id): id ∈ id_list] ;
3922    type_mark := find_old(type_id) ;
3923
3924    if not is_discrete_type(type_mark) then
3925        errmsg('Discriminant must have discrete type','','3.7.1') ;
3926       type_mark := 'any' ;
3927    end if ;
3928
3929     init_val  := if opt_init = [ ] then om
3930                  else check_type(type_mark, opt_init) end ;
3931
3932    (forall n ∈ nam_list)
3933        SYMBTAB(n) := ['discriminant', type_mark, init_val] ;
3934       decls with:=   ['object', [n], type_mark, init_val] ;    --
3935    end ;
3936
3937    init_norm := normalize(type_mark, opt_init) ;
3938    discr_tree with:= discr_spec + [type_mark, init_norm];
3939
3940 end forall ;
3941 return [decls, discr_tree] ;
3942
3943 end proc discr_decl ;
```

discr_redecl

```
3945 proc discr_redecl(type_name, opt_disc) ;
3946
3947 -- Verify conformance of discriminant part on redeclarations of types.
3948
3949
3950 old_decl := discr_decl_tree(type_name) ;
3951 if #old_decl /= #opt_disc then
3952    conformance_error ;
3953    return ;
3954 end if ;
3955
3956 (forall i ∈ [1..#old_decl])
3957    [id_list, type_id, opt_init] := opt_disc(i) ;
3958    [oid_list, otype_id, oopt_init, otype_mark, oinit_norm] :=
3959                                                  old_decl(i) ;
3960
3961    if id_list /= oid_list or not conform(type_id, otype_id) then
3962        conformance_error ;
```

```
3963     end if ;
3964
3965     type_mark := find_old(type_id) ;
3966     init_norm := normalize(type_mark, opt_init) ;
3967
3968     if init_norm /= oinit_norm
3969       or type_mark /= otype_mark
3970       or not conform(opt_init, oopt_init)
3971     then
3972         conformance_error ;
3973     end if ;
3974 end forall ;
3975
3976 -- Discard discriminant declaration tree after verifying conformance.
3977 discr_decl_tree(type_name) := om ;
3978
3979 end proc discr_redecl;
```

conformance_error

```
3981 proc conformance_error ;
3982 errmsg('non conformance to previous declaration','','6.3.1') ;
3983 end proc conformance_error;
```

comp_decl

```
3985 proc comp_decl(id_list, type_mark, opt_init);
3986
3987 -- Process record component declaration.
3988 -- Verify that the type is a constrained one, or that default values
3989 -- exist for the discriminants of the type.
3990
3991
3992 nam_list := [find_new(id) : id ∈ id_list] ;
3993 check_fully_declared2(type_mark) ;                    --
3994
3995
3996 -- Type-check the initial value, if provided.
3997
3998     if opt_init = [ ] then
3999         if is_access(type_mark) then
4000             init_val := 'NULL' ;
4001         else
4002           init_val := om ;
4003         end if ;
4004     else
4005 --
4006         init_val := check_type(type_mark, opt_init) ;
4007         if is_tuple(init_val)
```

```
4008            and init_val(1) ∈ {'qual_discr', 'qual_length'} then
4009                    init_val := init_val(3) ;
4010            end if ;
4011        end if ;
4012
4013    -- Try to catch self-reference within a record type (a frequent mistake).
4014        if type_mark ∈ open_scopes then
4015            errmsg('Invalid self-reference in definition of ',
4016                                    original_name(type_mark),'3.1') ;
4017        end if ;
4018        if is_unconstrained(type_mark)                          --
4019        then
4020            errmsg('Unconstrained '+nature(type_mark),
4021                            ' in component declaration', '3.6.1, 3.7.2');
4022        end if ;
4023
4024    (forall n ∈ nam_list)
4025        SYMBTAB(n) := ['field', type_mark] ;
4026    end ;
4027    --
4028    return [['field', [nam], type_mark, init_val]: nam ∈ nam_list] ;
4029
4030    end proc comp_decl ;
```

constrain_record

```
4032    proc constrain_record(type_mark, constraint) ;
4033
4034
4035
4036    -- Process discriminant constraints of record type.
4037    -- Verify that values have been provided for all discriminants, that
4038    -- the original type is unconstrained, and that the types of the
4039    -- supplied expressions match the discriminant types.
4040    if not is_record(type_mark) then
4041        errmsg('Invalid type for constraint', '','3.3, 3.7.2') ;
4042        return type_mark ;                                      --
4043    end if ;
4044
4045    d_list := discriminant_list(type_mark) ;
4046
4047    if d_list = [ ] or d_list = om then
4048        errmsg('Invalid constraint: ',
4049                    'Record type has no discriminant','3.7.1, 3.7.2') ;
4050        return type_mark ;                                      --
4051    end if;
4052
4053    d_seen := {} ;                  -- To verify that all discriminants were
4054                                    -- given values.
```

```
4055
4056
4057 -- Look for named associations in discriminant constraint list.
4058
4059 if exists ct = constraint(i) | is_tuple(ct)
4060                                  and ct(1) = 'choice_list' then
4061    first_named := i ;
4062    if exists nam ∈ constraint(i+1..) | is_tuple(nam)
4063                                  and nam(1) /= 'choice_list' then
4064       errmsg('Positional associations after named ones','','3.7.2') ;
4065       return type_mark ;                                          --
4066    end if ;
4067 else first_named := #constraint + 1 ;
4068
4069 end if ;
4070
4071 named_constraint := constraint(first_named..) ;
4072
4073 discr_map := {} ;
4074
4075 (for i ∈ [1..first_named-1])
4076
4077    d_name := d_list(i) ;
4078    if d_name = om then                    -- Exhausted discriminant list
4079       errmsg('Too many constraints for record type', '','3.7.2') ;
4080       return type_mark ;                                          --
4081    end if ;
4082
4083    d_val := check_type(type_of(d_name), constraint(i));
4084    check_discriminant(d_val) ;
4085    if d_val = [ ] then          -- Type error occurred.
4086       return type_mark ;                                          --
4087    else
4088       discr_map([d_name]) := d_val ;
4089    end if ;
4090    d_seen with:= d_name ;
4091 end for ;
4092
4093
4094
4095 (for [-, choice_list, expn] ∈ named_constraint)
4096    c_list := [ ] ;
4097
4098    (for [choice_type, name] ∈ choice_list)
4099       if choice_type /= 'simple_choice' then
4100             errmsg('Expect discriminant names only ',
4101                   ' in discriminant constraint','3.7.1, 3.7.2') ;
4102             return  type_mark ;                                   --
4103       elseif  not is_identifier (d_id := name(2)) or
```

```
4104          not is_map(comps := declared_components(type_mark)) or
4105          (d_name := comps(d_id)) = om or
4106            nature(d_name) /= 'discriminant' then
4107                errmsg('Invalid discriminant name ',
4108                       'in discriminant constraint','3.7.1, 3.7.2') ;
4109                return type_mark ;                              --
4110          elseif d_name ∈ d_seen then
4111                errmsg('Duplicate constraint for discriminant ',
4112                                d_id,'3.7.1, 3.7.2') ;
4113          else   c_list with:= d_name ;
4114               d_seen with:= d_name ;
4115             TO_XREF(d_name) ;
4116
4117                expn2 := check_type(type_of(d_name), expn) ;
4118                check_discriminant(expn2) ;
4119                if expn2 = [ ] then
4120                       return type_mark ;                       --
4121                end if ;
4122
4123        end if ;
4124      end for ;
4125
4126      discr_map(c_list) := expn2 ;
4127
4128 end for ;
4129
4130
4131 if #d_seen = #d_list then              -- All discriminants were ok.
4132     return  ['subtype', type_mark,  ['discr', discr_map]] ;
4133 else
4134     errmsg('Missing constraints ', 'for discriminants','3.7.2') ;
4135 end if ;
4136
4137     return type_mark ;                                         --
4138
4139
4140 end proc constrain_record ;
```

check_discriminant

```
4142 proc check_discriminant(expn) ;
4143
4144
4145 -- Verify that when a discriminant appears in an index constraint or a
4146 -- discriminant constraint, it appears by itself and not as part of a
4147 -- larger expression.
4148
4149
4150
```

```
4151 if not is_tuple(expn) then return false ;
4152
4153 elseif expn(1) = 'discr_ref'
4154 or  (expn(1) = 'qual_range' and expn(3)(1) = 'discr_ref')    --
4155 then return true ;
4156
4157 elseif exists sub_expn ∈ expn | check_discriminant(sub_expn) then
4158    errmsg('a discriminant appearing in a subtype indication ',
4159          'must appear by itself', '3.7.1') ;
4160    return false ;                          -- No need to propagate error.
4161 else return false ;
4162
4163 end if ;
4164
4165 end proc check_discriminant ;
```

variant_decl

```
4167 proc variant_decl(id, variant_list) ;
4168
4169
4170 discr_name := find_old(id) ;
4171 if nature(discr_name) /= 'discriminant' then
4172    errmsg('Invalid discriminant name in variant part',
4173                                                  '','3.7.1, 3.7.3') ;
4174    return [ ] ;
4175
4176 elseif (dtyp := type_of(discr_name)) = om then
4177    return [ ] ;
4178 else
4179
4180    case_list :=
4181         [ [[adasem(c): c ∈ choices], adasem(comps)] :
4182                                    [choices, comps] ∈ variant_list] ;
4183    [variants, others] := process_case(dtyp, case_list) ;
4184 end if ;
4185
4186 if others /= [ ] then
4187    variants with:= [{'others'}, others] ;                    --
4188 end if ;
4189
4190 -- Replace 'null' marker with empty component list.
4191 (forall v ∈ domain variants | variants(v) = ['null'])
4192    variants(v) := [[ ], [ ]] ;
4193 end ;
4194
4195 return [discr_name, variants] ;
4196
4197 end proc variant_decl ;
```

```
4198

      incomplete_decl

4200 proc incomplete_decl(id, opt_disc) ;
4201
4202 -- Process an incomplete declaration. The identifier must not have
4203 -- been declared already in the scope. However, an incomplete declaration
4204 -- may appear in the private part of a package, for a private type that
4205 -- has already been declared. In this case, the discriminants (if any)
4206 -- must match.
4207
4208 if declared(scope_name)(id) = om then
4209
4210    if nature(scope_name) /= 'private_part'
4211       -- Or it is not declared in the visible part.
4212       or (old_name := declared(open_scopes(2))(id)) = om then
4213
4214          name := find_new(id) ;
4215        type_of(name) := 'incomplete' ;
4216          root_type(name) := name ;
4217        process_discr(name, opt_disc) ;
4218          nature(name) := 'type' ;
4219        popscope() ;
4220
4221    elseif type_of(old_name) ∈ priv_types then
4222       -- redeclaration of private type in private part.
4223       nature(old_name) := 'record' ;
4224          discr_redecl(old_name, opt_disc) ;                    --
4225          name := old_name ;
4226
4227    else
4228          errmsg('invalid redeclaration of ' + id,
4229                                    ' in private part', '8.2(b)') ;
4230       return 'any' ;                                           --
4231    end if ;
4232
4233 else
4234    errmsg('invalid redeclaration of' ,id, '3.8, 8.2') ;
4235 end if ;
4236
4237 return name ;
4238
4239 end proc incomplete_decl ;
4240
4241 proc check_incomplete(type_mark) ;
4242
4243 -- Called to verify that an incomplete type is not used prematurely.
4244
```

```
4245 if type_of(base_type(type_mark)) = 'incomplete' then
4246     errmsg('Invalid use of type ' + original_name(type_mark),
4247                             ' before its full declaration', '3.8.1') ;
4248 end if ;
4249 end proc check_incomplete ;
```

promote_subtype

```
4251 proc promote_subtype(subtype) ;
4252
4253
4254
4255 -- This procedure is called when a subtype must be made into an
4256 -- anonymous type. This occurs when processing an object, constant or
4257 -- subtype declaration,
4258 -- when processing the formals of a procedure or entry, or the range
4259 -- of a task or entry family. If the subtype is already a type name,
4260 -- it is returned as is.If a previous subtype with the same structure was
4261 -- already promoted, then that one is returned. (this is the case when
4262 -- promoting a subtype appearing in the parameter list of a procedure
4263 -- body. The procedure spec. contained the same subtype, for which a
4264 -- promoted anonymous type was already created). If this is the first
4265 -- occurence of this subtypoe in this scope, we produce an anonymous
4266 -- type whose symbol table entry is the subtype structure.
4267
4268 if is_identifier(subtype) then
4269     return subtype ;
4270 elseif exists [id, anon] ∈ declared(scope_name) |
4271                             (id = anon) and SYMBTAB(anon) = subtype
4272     then
4273         return anon ;
4274 else
4275     new_name := anonymous_type() ;
4276     SYMBTAB(new_name) := subtype ;
4277     parent_type := type_of(new_name) ;
4278     root_type(new_name) := root_type(parent_type) ;                    --
4279     misc_type_attributes(new_name) := misc_type_attributes(parent_type);
4280     return new_name ;
4281 end if ;
4282
4283 end proc promote_subtype ;
```

subtype_expr

```
4285 proc subtype_expr(name) ;
4286
4287
4288
4289 -- Called to obtain the subtype expression of a  type name. This  is done
```

```
4290 -- for allocators and slices when a subtype indication appear in an expr.
4291 -- introduce an anonymous type for it, to simplify the type resolution of
4292 -- the complete expression. The code emitted however must include the
4293 -- subtype expression, to insure that the constraint is evaluated at the
4294 -- proper place.
4295
4296 if original_name(name) /= '' then
4297     -- Not an anonymous name. return as is.
4298     return name ;
4299
4300 elseif is_scalar_type(name) or is_record(name) then
4301     return symbtab(name) ;
4302
4303 elseif is_access(name) then
4304     return ['access', subtype_expr(designated_type(name))] ;
4305
4306 elseif is_array(name) then
4307     -- Collect subtype expression for each index, and emit constrained
4308     -- subtype.
4309
4310     return ['subtype', base_type(name),
4311       ['index', [subtype_expr(indx) : indx ∈ index_types(name)]],
4312                                           component_type(name)] ;
4313 end if ;
4314
4315 end proc subtype_expr ;
4316
```

Miscellanous type predicates.

is_type

```
4319 proc is_type(name) ;
4320
4321 const type_natures =
4322         {'type', 'subtype', 'array', 'record', 'enum', 'access',
4323                                     'task_type', 'task_type_spec'} ;
4324 return nature(name) ∈ type_natures ;
4325
4326 end proc ;
```

is_character_type

```
4328 proc is_character_type(name) ;
4329
4330 -- An enumeration type is a character type if it contains at least one
4331 -- character literal.
4332
```

```
4333 return root_type(name) = 'CHARACTER'
4334 or nature(bt := base_type(name)) = 'enum' and exists [c,-]
4335      ∈ literal_map(bt) | #c = 3 and c(1) = "'" and c(3) = "'" ;
4336 end proc is_character_type ;
```

is_discrete_type

```
4338 proc is_discrete_type(name);
4339
4340 const discrete_types =
4341             { 'INTEGER', 'universal_integer', 'discrete_type', 'any'} ;
4342
4343
4344
4345
4346 btype := (if TYPE_OF(name) /= om then root_type(name)   else om end) ;
4347
4348 return (btype ∈ discrete_types or
4349          (if btype = om then false else nature(btype) = 'enum' end)) ;
4350
4351 end proc is_discrete_type;
```

is_numeric

```
4353 proc is_numeric(name) ;
4354
4355 const numeric_types = {'INTEGER', 'FLOAT', '$FIXED','universal_integer',
4356                        'universal_fixed', 'universal_real'} ;
4357
4358 return root_type(name) ∈ numeric_types ;
4359
4360 end proc is_numeric ;
```

is_scalar_type

```
4362 proc is_scalar_type(name) ;
4363
4364 const scalar_types =
4365    {'INTEGER','FLOAT', '$FIXED', 'universal_integer','universal_real',
4366     'universal_fixed', 'discrete_type'} ;
4367
4368
4369
4370 root := root_type(name) ;
4371 return
4372   root ∈ scalar_types
4373  or
4374    nature(root) = 'enum' ;
4375
```

```
4376 end proc is_scalar_type ;
4377
```

is_incomplete_type

```
4379 proc is_incomplete_type(t) ;
4380
4381 -- A type is incomplete if only an incomplete type declaration for it
4382 -- has been seen, or if one of its subcomponents is an incomplete private
4383 -- type (because of other rules, a subcomponent can never have an incomp-
4384 -- lete type).
4385
4386 return type_of(b := base_type(t)) = 'incomplete'
4387 or private_ancestor(b) /= om ;
4388
4389 end proc is_incomplete_type;
```

is_unconstrained

```
4391 proc is_unconstrained(typ) ;
4392
4393
4394
4395 return  (nature(typ) = 'array'
4396       or
4397         (nature(typ) = 'record' or type_of(typ) ∈ incp_types) and
4398           -- Some discriminant has no default value.
4399         exists discr ∈ discriminant_list(typ) | signature(discr) = om);
4400
4401 end proc is_unconstrained ;
```

can_constrain

```
4403 proc can_constrain(d_type) ;
4404
4405 -- Determine whether an object, actual parameter, type def, etc. can
4406 -- receive a constraint.The predicate -is_unconstrained- used in decla-
4407 -- rations is too weak here, because it returns false on discriminated
4408 -- records with default values.
4409
4410 return (nature(d_type) = 'array')  or
4411       (is_record(d_type) and nature(d_type) /= 'subtype'
4412                                          and has_discriminants(d_type)) ;
4413
4414 end proc can_constrain;
```

base type

```
4416 proc base_type(name);
4417 -- The base-type of a type-mark is itself, unless the type-mark denotes
4418 -- a subtype. In that case the nature of name is 'subtype', unless it is
4419 -- an unconstrained subtype, in which case its base type is its type_of.
4420
4421 return
4422     if nature(name) = 'subtype' then
4423         if type_of(name) = name then name else base_type(type_of(name))
4424        end
4425     elseif nature(name) ∈ ['record', 'array'] then type_of(name)
4426     else
4427         name
4428     end ;
4429 end proc base_type;
```

new_type_list

```
4431 proc new_type_list() ;
4432
4433
4434
4435 -- This procedure is invoked at the end of each declarative part,
4436 -- to format the new type declarations before insertion into the
4437 -- intermediate code.
4438 -- The type definitions for new types are in the map TYPE_OF.
4439 -- In the case of new and derived types, all that matters to the
4440 -- interpreter is the original type. Therefore , for types whose
4441 -- structure is ['new', old_type]  we simply emit : old_type together
4442 -- with any constraint on it.
4443 --
4444 -- For arrays, their internal representation is that of a subtype of
4445 -- some internal 'sequence' object, which lacks index constraints.
4446 -- This dependence is irrelevant to the interpreter. We emit the
4447 -- signature of the type, which is the pair [index, component].
4448
4449 -- For records, the type field holds the declarations for the comps.
4450 -- and the signature holds declared map from selector names to their
4451 -- unique name. This map is only used internally to validate the
4452 -- names of selectors in qualified names and aggregates. It is not passed
4453 -- to the interpreter.
4454 -- This formatting is also done for user-defined types. Routine ais_type
4455 -- performs this function.
4456
4457 t_list frome NEWTYPES ;
4458
4459
```

```
4460 return [ais_type(type_name) : type_name ∈ t_list] ;
4461
4462 end proc new_type_list ;
```

named_type

```
4464 proc named_type(name) ;
4465
4466
4467
4468 -- This procedure is invoked when an anonymous type can be given a name
4469 -- that relates to its nature (e.g the base type of a derived type).
4470
4471 type_name :=  prefix + name + suffix ;
4472 declared(scope_name)(name) := type_name ;
4473 scope_of(type_name) := scope_name ;
4474 return type_name ;
4475
4476 end proc named_type ;
```

anonymous_type

```
4478 proc anonymous_type ;
4479
4480
4481
4482 -- This procedure is called to produce a new identifier for an anonymous
4483 -- type. The new identifier is inserted into the symbol table, and into
4484 -- the type stack.
4485
4486 new_name := named_atom('') ;
4487
4488 declared(scope_name)(new_name) := new_name ;
4489 scope_of(new_name) := scope_name ;
4490 top(NEWTYPES) with:= new_name ;
4491
4492 return new_name ;
4493
4494 end proc anonymous_type ;
4495
4496 proc named_atom(id);
4497
4498 -- This procedure uses the unique name generated for a compilation
4499 -- unit to produce new names that will be unique throughout a library,
4500 -- especially one containing more than one AIS file.
4501
4502     return
4503
4504         if unit_name(1) = 'body' then 'UB:' else '' end
```

```
4505        +/[unit_name(i) + '.' : i ∈ [#unit_name,#unit_name-1..3]]
4506        + unit_name(2)
4507        + if unit_name(2) = '' then '' else '.' end
4508        + id
4509        + str newat;
4510
4511 end proc named_atom;
```

4. Names and Expressions

check_type

```
4514 proc check_type(context_type, expn) ;
4515
4516
4517 -- This procedure performs type checking and operator disambiguation.
4518 -- -expn- is an expression tree, which must have the type -context_type-.
4519 -- This procedure is called from all reductions where the type of
4520 -- an expression is known a priori : assignments, conditionals, etc.
4521 -- The procedure returns -expn-, suitably modified if disambiguation
4522 -- had to take place.
4523
4524 --
4525 -- Some contexts require that a type belong to a class of types instead
4526 -- of one  specific type. For example, a condition must be of a boolean
4527 -- type, not just BOOLEAN.
4528
4529 const type_classes =
4530       ['boolean_type', 'discrete_type',    'integer_type',
4531        'real_type',    'universal_type'] ;
4532 --
4533
4534
4535 noop_error := false ;
4536 tree1 := resolve1(expn) ;              -- Bottom-up pass.
4537
4538 if noop_error then                     -- No valid types
4539    noop_error := false ;               -- error emitted already
4540    return [ ] ;
4541 end if ;
4542
4543 --
4544 [-,types] := tree1 ;
4545 old_context := context_type ;                  -- save for test below
4546 if context_type ∈ type_classes then
4547    -- Keep only those that belong to this class.
4548    types := {t ∈ types | compatible_types(t, context_type)} ; --
4549
4550    if # types > 1 then
4551       -- May be overloaded operator: user_defined one hides predefined.
4552       types -:= univ_types ;
4553    end if ;
```

```
4554
4555    if # types = 1 then
4556        context_type := arb types ;
4557    else
4558        type_error({'any'}, context_type, #types) ;
4559    end if ;
4560 end ;
4561 --
4562
4563 tree2 := resolve2(tree1, context_type) ;
4564 if tree2 = [ ] then return [ ] ; end if ;
4565
4566 -- Now emit a constraint qualification if needed, and try to fold it.
4567 if old_context   type_classes then
4568    tree2 := apply_constraint(tree2, context_type) ;
4569    tree2 := eval_static(tree2) ;                              --
4570 end if ;
4571
4572 -- Add type to resulting expression, if not present. (temporary)
4573 if is_value(tree2) and #tree2 = 2 then
4574    tree2 with:= context_type ;
4575 end if ;
4576
4577 return tree2 ;
4578
4579 end proc check_type ;
```

constraint_kind

```
4581 proc constraint_kind(typ) ;
4582
4583
4584 if is_unconstrained(typ) or typ ∈ univ_types then return '' ;
4585 else  return
4586         (if is_scalar_type(typ)  then 'qual_range'
4587        elseif   is_array(typ)  then 'qual_length'
4588        elseif   is_record(typ) then
4589           (if nature(typ) = 'subtype' then 'qual_discr' else '' end)
4590         elseif   is_access(typ) then
4591           (if is_scalar_type(d := designated_type(typ))
4592                                then 'qual_arange'
4593            elseif is_unconstrained(d) then ''                --
4594            elseif is_array(d)  then 'qual_alength'
4595            elseif is_record(d) then
4596              (if nature(d) = 'subtype' then 'qual_adiscr' else '' end)
4597            else '' end)
4598         else '' end) ;
4599 end if ;
4600
```

```
4601 end proc constraint_kind ;
```

apply_constraint

```
4603 proc apply_constraint(expn, typ) ;
4604 -- Apply a constraint check to an expression, in a context that imposes
4605 -- it.
4606 constraint := constraint_kind(typ) ;
4607
4608 if constraint = ''                          -- unconstrained context
4609   or is_identifier(expn) and type_of(expn) = typ
4610   -- expression has type of context
4611   or is_tuple(expn) and expn(1) ∈ ['new', '()'] then
4612       -- constraint has already been applied to the inner expression.
4613       return expn ;
4614 else
4615     if is_access(typ) then
4616           typ := designated_type(typ) ;
4617     end if ;
4618     expn := [constraint, typ, expn] ;
4619
4620 end if ;
4621
4622 return expn ;
4623
4624 end proc apply_constraint ;
4625
4626 --
```

expr_type

```
4628 proc expr_type(expn) ;
4629
4630 -- Utility to obtain the type of an expression, after type resolution.
4631 -- If the expression is not an identifier, the type appears explicitly
4632 -- within  the expression; otherwise retrieve type from symbol table.
4633
4634 return if is_identifier(expn) then type_of(expn)
4635 elseif expn = [ ] then 'any'
4636 elseif expn(1) ∈ qualifiers then expn(2)
4637 else top(expn) end ;
4638
4639 end proc expr_type;
```

find_check_type

```
4641 proc find_check_type(expn) ;
4642
4643 -- This procedure is invoked to determine the type of an expression
```

```
4644 -- whose type cannot be found from context. This is the case for
4645 -- the left-hand side of an assignment, for a case expression, and
4646 -- an index constraint.
4647 -- We proceed as in check_type, but save and return the (unique )
4648 -- type yielded by resolve1.
4649
4650 noop_error := false ;
4651
4652 tree1 := resolve1(expn) ;
4653
4654
4655 if noop_error then return [ ] ; end if ;
4656
4657 [-,types] := tree1 ;
4658
4659 -- The expression must be unambiguous, because there is no context
4660 -- for further determination. However, the expression may be universal,
4661 -- and also contain user-defined operators that yield non-universal
4662 -- types. In the context of a constant definition we must opt for
4663 -- the universal type.
4664
4665 if #types > 1 then
4666     types *:= univ_types ;
4667 end if ;
4668
4669 if #types /= 1 then
4670     -- Now there is an error.
4671     return [ ] ;
4672 else
4673     typ := arb types ;
4674 end if ;
4675
4676
4677 tree2 := resolve2(tree1, typ) ;
4678
4679
4680 return [tree2, typ]  ;
4681
4682 end proc find_check_type ;
4683
```

- RESOLVE1 -

```
4685 proc resolve1(expn) ;
4686
4687 -- This procedure performs the first, bottom-up pass of the type
4688 -- checking and overload resolution. It rebuilds the expression tree
4689 -- for -expn-, and returns a pair : [ new expn, set of types of expn].
4690
```

```
4691 -- If a type error is encountered, this procedure returns the following
4692 -- template :
4693 const error_expn = [ [ ], {}] ;
4694
4695
4696
4697
4698
4699 if (noop_error ? false) then return error_expn ; end if ;
4700
4701 if is_identifier(expn) then
4702
4703     id_type := type_of(expn) ;
4704     return
4705     case nature(expn) of
4706
4707 --
4708         ('obj', 'constant', 'in', 'inout', 'out'): [expn, {id_type}],
4709 --
4710         ('type', 'subtype', 'enum', 'record', 'array', 'access'):
4711               pass1_error('Invalid use of type '+ original_name(expn),
4712                                         ' in expression', '4.4'),
4713
4714         ('task_obj', 'task_obj_spec', 'task_type', 'task_type_spec'):
4715                                   [expn, {id_type}],
4716
4717 --
4718         ('discriminant'): [['discr_ref', expn, id_type], {id_type}],
4719
4720         ('void'): pass1_error('premature use of identifier ',
4721                                   original_name(expn), '8.3')
4722     else
4723           pass1_error('Invalid use of identifier ',
4724                       original_name(expn), '4.4')
4725     end ;
4726
4727 elseif is_overloaded(expn) then
4728     -- An overloaded name is either an enumeration literal, or a sub-
4729     -- program call without parameters. Rebuild the expression in the
4730     -- format of a nullary operation.
4731     return resolve1( [expn]) ;
4732 else
4733     op_name := expn(1) ;                    -- Expression with arguments.
4734
4735
4736     if is_overloaded(op_name) then
4737         arg_list := [resolve1(expn(i)) : i ∈ [2..#expn]] ;
4738
4739         exp1 := result_types(op_name, arg_list) ;
```

```
4740        if noop_error then return error_expn ;
4741        elseif exp1(2) = {} then          -- No valid types.
4742           type_error(op_name, om, 0) ;
4743            return error_expn ;
4744        else return exp1 ;
4745        end if;
4746
4747
4748    else
4749        -- Operator is not overloaded : it is a basic  operation
4750        -- on arrays, entries, records, aggregates, or  attributes.
4751
4752    case op_name of
4753
4754        -- A name in an expression appears in the format :
4755        --  ['name',  name]
4756        -- We must first perform name resolution on it, and then
4757        -- validate types.
4758
4759    ('name') : return resolve1(find_old(expn(2))) ;
4760
4761    ('int_literal') :   return [expn, {'universal_integer'}] ;
4762
4763    ('real_literal'):   return [expn, {'universal_real'}] ;
4764
4765    ('string_literal'): return [expn, {'string_type'}] ;
4766
4767    ('null'):                 return [expn, find_access_types()] ;  --
4768
4769    ('aggregate'):
4770
4771        -- Verify that the list of choices is properly formatted. The
4772        -- types of the individual choices are not used to resolve the
4773        -- aggregate type. The possible types of the aggregate are all
4774       -- composite types in scope.
4775
4776       arg_list := [resolve1(expn(i)): i ∈ [2..#expn]] ;
4777
4778        if (exists arg = arg_list(i) | arg(1) = 'choice_list' )
4779           and (exists arg2 ∈ arg_list(i+1..) |
4780                                    arg2(1) /= 'choice_list')
4781        then
4782           return pass1_error('positional associations must appear',
4783                                        ' first in aggregate', '4.3') ;
4784        else
4785          return [['aggregate', arg_list], find_agg_types()]; --
4786          --- TBSL: introduce generic type 'composite_type' to simplify.
4787        end if ;
4788
```

```
4789
4790    ('[ ]') :
4791        [-, a_expn, i_expn] := expn ;
4792
4793        expn1 := valid_array_expn(a_expn, i_expn) ;
4794        if expn1 = error_expn then
4795            return pass1_error('type mismatch in indexing', '','4.1.1');
4796        end if ;
4797         return expn1 ;
4798
4799
4800    ('[..]'):
4801        -- Slicing operations are equivalent to indexing operations,
4802        -- for type checking purposes. We simply reformat the result
4803        -- of type checking, so that the result type of the slice is
4804        -- the base type of the array expression. If this type is an
4805        -- access type, we must of course dereference it.
4806
4807        [-, a_expn, discr] := expn ;
4808        expn1 := valid_array_expn(a_expn, [discr]) ;
4809        if expn1 = error_expn then
4810            return pass1_error('type mismatch in slice','', '4.1.2') ;
4811         else
4812            [ [-, [arrays, array_types], indx], -] := expn1 ;
4813
4814            return [['[..]', [arrays, array_types], indx(1)],
4815             {base_type(if is_access(t) then designated_type(t)
4816                                        else t end)  : t ∈ array_types}] ;
4817        end if ;
4818
4819
4820    ('.'):
4821        [-, obj, selector] := expn ;
4822
4823        return valid_selected_expn(obj, selector) ;              --
4824
4825
4826 --
4827    ('in', 'notin'):
4828        -- The second argument of membership operators can be a type_mark
4829        [op_name, arg1, arg2] := expn ;
4830        arg11 := resolve1(arg1) ;
4831        if arg2(1) = 'name' and is_identifier(id := arg2(2)) then
4832            -- This can be a qualified name also. TBSL
4833            nam := find_old(id) ;
4834            if (t := type_of(nam)) ∈ priv_types then
4835                t := nam ;
4836            end if ;
4837            arg21 := [ nam, {t}] ;
```

```
4838          else
4839             arg21 := resolve1(arg2) ;
4840          end if ;
4841
4842 --
4843          [nam, ts] := arg21 ;
4844
4845          if is_tuple(nam) then
4846             if nam(1) = '"' and nam(2) = 'BASE' then
4847                errmsg('invalid use of attribute BASE', '', 'Annex A') ;
4848             elseif nam(1) = 'subtype' then
4849                ts := {t ∈ ts | is_scalar_type(t)} ;
4850                if ts = {} then return
4851                  pass1_error('bounds of range for membership op ',
4852                                                'must be scalar', '4.4') ;
4853                else arg21(2) := ts ;
4854                end if ;
4855             end if ;
4856          end if ;
4857
4858          -- Now resolve the expression as for any other operator.
4859          exp1 := result_types({op_name}, [arg11,arg21]) ;
4860          if noop_error then return error_expn ;
4861          elseif exp1(2) = {} then
4862             type_error({op_name}, om, 0) ;
4863            return error_expn;                --
4864          else return exp1 ;
4865          end if ;
4866
4867 --
4868    ('@'):
4869          -- A dereference operation must apply to objects of access type.
4870          [-, ac_expn] := expn ;
4871       expn1 := resolve1(ac_expn) ;
4872       desig_types := {designated_type(t): t ∈ expn1(2)
4873                                                | is_access(t)} ;
4874       if desig_types = {} then
4875          return pass1_error('prefix of ALL must belong ',
4876                                        'to an access type', '4.1.3') ;
4877       else
4878          return [ ['@', expn1], desig_types ] ;
4879       end if ;
4880
4881    ('new'):
4882       -- 'new' labels an allocator given with a type mark and an
4883       -- optional constraint.
4884
4885         [-, type_id, constraint] := expn ;
4886
```

```
4887        type_mark := find_old(type_id) ;
4888
4889        if constraint = [ ] and is_unconstrained(type_mark)     --
4890        then
4891            return  pass1_error('Constraint required in allocator ',
4892                                'when initialization is absent', '4.8') ;
4893        else
4894            desig_type := make_subtype([type_id, constraint]) ;
4895
4896        end if ;
4897
4898        -- if the designated type is a subtype expression, make it
4899        -- into an anonymous type for simplicity of processing.
4900
4901        if is_tuple(desig_type) then
4902            anon := named_atom('') ;
4903            symbtab(anon) := desig_type ;
4904            root_type(anon) := root_type(type_mark) ;                --
4905            desig_type := anon ;
4906        end if ;
4907        check_fully_declared(desig_type) ;
4908       a_types := {a ∈ find_access_types() |
4909             compatible_types(desig_type, designated_type(a))}; --
4910        return [['new', [ ], desig_type], a_types] ;
4911
4912
4913   ('new_init'):
4914        -- Allocator given by a type mark and an explicit aggregate.
4915
4916        [-, type_id, aggregate] := expn ;
4917        desig_type := find_old(type_id) ;
4918
4919        if not is_type(desig_type) then
4920            return pass1_error('invalid type mark in allocator', '',
4921                                                          '4.8') ;
4922       elseif is_limited_type(desig_type) then                   --
4923          return pass1_error('initial value not allowed ',
4924                      'on an allocator for a limited type', '7.4.4') ;
4925       end if ;
4926
4927        expn1 := resolve1(aggregate) ;
4928       a_types := {a ∈ find_access_types() |
4929             compatible_types(desig_type, designated_type(a))}; --
4930       return [['new', expn1, desig_type], a_types] ;
4931
4932
4933   (''''):
4934        return valid_attribute(expn) ;
4935
```

```
4936    ('choice_list'):
4937        -- If the choice is a simple choice of an identifier, this may
4938        -- denote a formal name or a component name. Do not resolve
4939        -- them until the second pass.
4940
4941        [-, choices, value] := expn ;
4942
4943        choice_list := [ ] ;
4944
4945        (for [choice_type, indx] ∈ choices)
4946
4947         case choice_type of
4948
4949         ('simple_choice') :
4950          if is_tuple(indx) and indx(1) = 'name' then
4951             if (is_identifier(choice := indx(2))) then
4952                indx1 := [choice, {'any'} ] ;      -- Type found later
4953
4954             elseif is_range_attribute(choice)
4955             then
4956                  choice_list with:= ['range_choice',
4957                                      resolve1(find_old(choice))] ;
4958                  continue for ;
4959
4960             else                          -- not a simple name: can resolve
4961                indx1 := resolve1(indx) ;
4962             end if ;
4963          else                  -- expression: can resolve.
4964              indx1 := resolve1(indx) ;
4965          end if ;
4966
4967          if is_tuple(indx) and indx(1) = 'subtype'
4968               then
4969                 choice_list with:= ['range_choice', indx1] ;
4970               else
4971                  choice_list with:= [choice_type, indx1] ;
4972
4973          end if ;
4974
4975          -- For a slice choice, the index is a subtype. If this sub-
4976          -- type is an explicit range construct, its bounds have to be
4977          -- checked. Otherwise, the name of the subtype yields the
4978          -- type.
4979
4980         ('range_choice'):
4981
4982          indx1 := resolve1(indx) ;
4983          choice_list with:= [choice_type, indx1] ;
4984
```

```
4985
4986          ('others_choice'):
4987            -- Verify that this appears by itself in the choice list.
4988
4989            if #choice_list > 1 then
4990                pass1_error('OTHERS choice must appear by itself',
4991                                    '', '3.7.3, 4.3, 5.4') ;
4992            else
4993                -- Others is an index type that matches everything.
4994               choice_list with:= [choice_type,[ [ ], {'any'}]] ;
4995            end if ;
4996
4997
4998          end case ;
4999        end for ;
5000
5001        return [ 'choice_list', choice_list, resolve1(value)] ;
5002
5003
5004
5005      ('qual_range'):
5006          -- When qual_range appears in an expression, the bounds have
5007          -- been type-checked. Simply extract the known result type.
5008          return [expn, {expn(2)}] ;
5009
5010      ('convert'):
5011          -- the result type is the type mark of the conversion.
5012        [-, target_type, arg] := expn ;
5013          if is_tuple(arg) and arg(1) ∈
5014              ['null', 'new', 'new_init', 'aggregate', 'string_literal']
5015        then return
5016            pass1_error('invalid expression for conversion','','4.6(3)');
5017        else
5018           expn1 := resolve1(arg) ;
5019          end if ;
5020          return if expn1 = error_expn
5021            then error_expn
5022          else [['convert', target_type, expn1], {target_type}]
5023        end;
5024
5025      ('qualify'):
5026          [- , to_type, arg] := expn ;
5027
5028          if not is_type(to_type) then return
5029            pass1_error('Expect type mark in qualified expression',
5030                                                        '','4.7');
5031          elseif to_type ∈ open_scopes and is_task_type(to_type) then
5032        return
5033          pass1_error('invalid use of type ' + original_name(to_type),
```

```
5034                                    ' within its own body', '9.1') ;
5035          else
5036            expn1 := resolve1(arg) ;
5037          end if ;
5038
5039          [-, types] := expn1 ;
5040
5041          return if exists t ∈ types | compatible_types(to_type,t)
5042            then  [ ['qualify', to_type, expn1], {to_type}]
5043            else  pass1_error('Expression has wrong type ',
5044                                          'for qualification', '4.7')
5045          end ;
5046
5047    ('subtype') :
5048          -- For a subtype expression, the bounds expressions must be
5049          -- checked against the specified type, if any, or against the
5050          -- type required by context.
5051          -- In the case of floating point types, the specified precision
5052          -- is a static expression which has been checked already.
5053
5054          [-, typ, [kind, low, high, precision]] := expn ;
5055
5056          [l1, ltypes] := low1 := resolve1(low) ;
5057          [h1, htypes] := high1:= resolve1(high) ;
5058
5059          if typ = [ ] then
5060             -- Case of a range expression with no named type. Validate
5061             -- the bounds against each other, and return the possible
5062             -- types.
5063            possible_types := {it : t1 ∈ ltypes, t2 ∈ htypes |
5064                                    (it := intersect_types(t1,t2)) /= om};
5065
5066           if # possible_types = 0 then
5067             return pass1_error('Incompatible expressions in range',
5068                                    '', '3.5') ;
5069             else return [ ['subtype',[ ],[kind, low1, high1, precision]],
5070                                          possible_types] ;
5071             end if ;
5072          else
5073             -- Subtype of a specified type. Validate the bounds against
5074             -- it, and return the named type.
5075                                                                    --
5076             if exists t1 ∈ low1(2) | compatible_types(typ, t1)
5077           and  exists t2 ∈ high1(2)| compatible_types(typ, t2)
5078             then return
5079                  [ ['subtype', typ, [kind, low1, high1,precision]],
5080                                        {typ} ] ;
5081             else return
5082                  pass1_error('Invalid types in bounds for range',
```

```
5083                                        '', '3.5, 4.1.2') ;
5084              end if ;
5085            end if ;
5086
5087    ('()'):
5088        -- A parenthesised expression carries a special operator, in
5089        -- order to distinguish it from a variable. (Thus (X) is not a
5090        -- valid OUT parameter for a procedure, and (D) is not a valid
5091        -- use of a discriminant name).
5092
5093        [-, e] := expn ;
5094      expn1 := resolve1(e) ;
5095        return [['()', expn1], expn1(2)] ;
5096 --
5097    ('call_or_index'):
5098      -- In the presence of calls to parameterless functions, the
5099        -- parse tree itself must be considered overloaded, and each
5100        -- of its interpretations resolved separately.
5101
5102      [-, obj_name, arg_list] := expn ;
5103        -- resolve as function call.
5104      call1 := result_types(obj_name,[resolve1(arg):arg ∈ arg_list]);
5105        -- And try for indexing on parameterless call.
5106        array1 := valid_array_expn([obj_name], arg_list) ;
5107        trees1 := {call1, array1} ;
5108        types1 := call1(2) + array1(2) ;
5109      return [[op_name, trees1], types1] ;
5110 --
5111    ('range'):                                  -- A frequent error.
5112      return pass1_error('Invalid use of discrete range ',
5113                                          'in expression', '4.4') ;
5114
5115        else
5116          if op_name = om then
5117            -- usually a previous error; often an invalid selected
5118            -- component name.
5119            noop_error := true ;
5120            return error_expn ;
5121        else
5122            return
5123          pass1_error('Invalid operator in expression: ',
5124                                  op_name, '4.4, 4.5');
5125        end if ;
5126        end case ;
5127    end if ;
5128
5129
5130 end if ;
5131 end proc resolve1;
```

```
5132

      -RESOLVE2-

5134 proc resolve2(typed_expn, context_typ) ;
5135
5136 -- This procedure performs the second, top-down pass of the
5137 -- type validation and overloading resolution. The first argument
5138 -- is a pair : [expression, possible types for expression] . The
5139 -- second argument is the type which the expression must yield.
5140 -- If the expression is overloaded, only one of its instances must
5141 -- yield -context_typ-. Once this is ascertained, the known types of the
5142 -- formals for the top level operator in expression , are propagated
5143 -- downwards to the actuals.
5144
5145
5146 if (noop_error ? false) then return [ ] ; end if ;
5147
5148 [expn, types ] := typed_expn ;
5149
5150 if expn = [ ] then return [ ] ; end if ;
5151
5152 if is_identifier (expn) then
5153     -- If constant, get its value, and if universal constant,
5154     -- convert when necessary.
5155     type2 := type_of(expn) ;
5156     expn2 := specialize(const_val(expn), context_typ) ;
5157
5158 --
5159     if not compatible_types(context_typ, type2) then
5160         errmsg(original_name(expn) + ' has incorrect type.Expect ',
5161                         full_type_name(context_typ),'none') ;
5162     return [ ] ;
5163 --
5164     elseif nature(expn) = 'out' and not out_context then
5165         errmsg('invalid reading of out parameter ', original_name(expn),
5166                                                         '6.2') ;
5167     end if ;
5168     return expn2 ;
5169 --
5170

      -RESOLVE2- expression

5172 else                        -- We have an expression
5173     op_name := expn(1) ;
5174     if is_overloaded(op_name) then
5175
5176         arg_list := expn(2..) ;
```

```
5177        -- Find instance of operator that yields type imposed by context.
5178        valid_ops :=  [ name ∈ op_name |
5179              compatible_types(context_typ, type_of(name))] ; --
5180
5181        if #valid_ops > 1 then
5182          valid_ops := disambiguate(valid_ops, arg_list, context_typ) ;
5183        end if ;
5184
5185        if #valid_ops > 1 then
5186          -- try removing implicit conversions of universal quantities.
5187          [expn, types] := remove_conversions(valid_ops, arg_list) ;
5188          if #expn(1) = 1 then       -- Disambiguation succeeded.
5189            valid_ops := [arb expn(1)] ;
5190          end if ;
5191        end if ;
5192
5193        -- Now there should be only one possibility left.
5194      if #valid_ops /= 1 then
5195
5196          type_error(op_name, context_typ, #valid_ops) ;
5197          return [ ] ;
5198        else
5199          op_name := valid_ops(1) ;
5200          type2 := type_of(op_name) ;
5201        end if ;
5202      -- Downward propagation to the arguments must be done on a case
5203      -- by case basis, depending on the nature of the operator.
5204
5205      case nature(op_name) of
5206
5207        -- For a predefined operator, the type imposed by context fixes
5208        -- the types of the arguments. The signature of a predefined op.
5209        -- contains only classes of types, and it ignored in this pass.
5210
5211    ('op') :
5212        expn2 := complete_op_expr(op_name,
5213                                  base_type(context_typ), arg_list) ;
5214
5215      -- For a procedure or function, the signature imposes a type on
5216      -- each actual parameter present, and specifies a default value
5217      -- for the ones that are absent. If the function is aliased (ie
5218      -- a renaming or derivation) the parent subprogram is called.
5219
5220    ('procedure', 'procedure_spec', 'function', 'function_spec') :
5221 --
5222        formals := signature(op_name) ;
5223        arg_list := complete_arg_list(formals, arg_list) ;
5224        subp_name := ALIAS(op_name) ? op_name ;
5225        expn2 := ['call', subp_name, arg_list] ;
```

```
5226        TO_XREF(op_name) ;
5227 --
5228    ('entry', 'entry_family'):
5229
5230        formals := signature(op_name) ;
5231          arg_list := complete_arg_list(formals, arg_list) ;
5232          expn2 := ['ecall', scope_of(op_name), op_name, arg_list] ;
5233          TO_XREF(op_name) ;
5234
5235
5236        -- Resolved enumeration literals are returned as themselves.
5237
5238    ('literal'): expn2 := op_name ; TO_XREF(op_name) ;
5239
5240        end case ;
5241
5242    else case op_name of                    -- Basic operation
5243
5244    ('int_literal'):
5245    -- If the context  type is not universal, the literal must be trans-
5246    -- formed to its short SETL form.
5247
5248        target_typ := if context_typ = 'universal_integer'
5249            then 'universal_integer'
5250            else 'integer'
5251        end ;
5252
5253        lv := ADAVAL(target_typ, expn(2));
5254        if lv = 'OVERFLOW' then
5255            expn2 := numeric_error() ;
5256        else
5257            expn2 := ['ivalue', lv] ;
5258        end if;
5259
5260        type2 := 'universal_integer' ;
5261
5262    ('real_literal'):
5263    -- If the context is not universal, or is not a fixed type, then
5264    -- convert the literal to a SETL floating number.
5265
5266        target_typ := if context_typ = 'universal_real'
5267                          or root_type(context_typ) = '$FIXED'
5268            then 'universal_real'
5269              else 'float'
5270        end ;
5271        lv := ADAVAL(target_typ, expn(2));
5272        if lv = 'OVERFLOW' then
5273            expn2 := constraint_error() ;
5274        else
```

```
5275          expn2 := ['ivalue', lv] ;
5276       end if;
5277
5278       type2 := 'universal_real' ;
5279
5280    ('string_literal'):
5281
5282       if is_array(context_typ) then
5283          if context_typ = 'string_type' then
5284             -- verify that only one string type is visible.
5285             context_typ := 'STRING' ;
5286          elseif is_fully_private(context_typ) then
5287             premature_access(context_typ) ;
5288          end if ;
5289
5290          expn2 := complete_string_literal(expn(2),
5291             index_types(context_typ), component_type(context_typ)) ;
5292
5293       else     -- Some type error. Will be caught at end
5294          expn2 := [ ] ;
5295       end if ;
5296
5297       type2 := 'string_type' ;
5298
5299    ('null'):                                                    --
5300    if is_access(context_typ) then
5301       expn2 := 'NULL';
5302       type2 := context_typ ;
5303    else
5304       errmsg('invalid context for NULL',",'3.8') ;
5305       return [ ] ;
5306    end if ;
5307
5308
5309    ('aggregate') :
5310    -- Resolve it using the  context type, and apply constraint if any.
5311    -- The possible types include all visible composite types,and there
5312    -- should be one of them compatible with the context.
5313
5314    arg_list := expn(2) ;
5315    type2 := context_typ;
5316    expn2 := complete_aggregate(context_typ, arg_list) ;
5317    comp_check := constraint_kind(context_typ) ;
5318    if comp_check = '' then comp_check := 'yes' ; end if ;  --- clean up.
5319
5320    -- For arrays, obtain required index type from type of array
5321    -- expression, and complete the determination of both.
5322
5323    ('[ ]') : [-, array1, index_list] := expn ;
```

```
5324    [expn2, array_type] :=
5325        complete_array_expn(array1, #index_list, context_typ) ;
5326    if expn2 = om then return [ ] ;
5327    else
5328      -- Complete resolution of each index.
5329 --
5330      -- The  index expression is  a context  in which  out parameters
5331      -- cannot  be  read. This  has to  be  special-cased  because an
5332      -- indexed  expression on  the lhs  of an assignment is a  valid
5333      -- context for an out parameter, and the global flag out_context
5334      -- is set accordingly in  processing assignments.
5335
5336      out_c := out_context ;
5337      out_context := false ;
5338
5339      index2 := [resolve2(indx, index_types(array_type)(i)) :
5340                                              indx = index_list(i)] ;
5341      expn2 := ['[ ]', expn2, index2] ;
5342      out_context := out_c ;
5343 --
5344    end if ;
5345
5346    type2 := component_type(array_type);
5347
5348
5349
5350    -- For slices, obtain array type, and apply its index type to the
5351    -- subtype expression for the discrete range.
5352
5353    ('[..]') : [-, array1, discr1] := expn ;
5354    [expn2, array_type] := complete_array_expn(array1, 0, context_typ);
5355    if expn2 = om then
5356        return [ ] ;
5357    else
5358        expn2 :=
5359            ['[..]', expn2, resolve2(discr1, index_type(array_type))] ;
5360       type2 := array_type ;
5361    end if ;
5362
5363
5364    ('.') : [-, obj1, selector ] := expn ;
5365 --
5366    [expn2, type2] :=
5367                complete_selected_expn(obj1, selector, context_typ) ;
5368
5369    if expn2 = om then return [ ] ; end if ;
5370
5371    -- For a parenthesised expression, resolve the expression, and keep
5372    -- the parenthesis, to distinguish them from variables. The possible
```

```
5373    -- constraint of the context is not propagated to the expression, but
5374    -- a constraint check may have to be performed on the whole.
5375
5376    ('()'):
5377    e := resolve2(expn(2),-base_type(context_typ)) ;
5378    expn2 := ['()', apply_constraint(e, context_typ)] ;          --
5379    type2 := context_typ ;
5380
5381    -- For a dereference operation, find the access type whose designated
5382    -- type fits the context.
5383    ('@') : [-, ac_expn] := expn ;
5384    a_types := {t ∈ ac_expn(2) | is_access(t) and
5385            compatible_types(context_typ, designated_type(t))} ; --
5386    if #a_types /= 1 then
5387        type_error({'@'}, t, #a_types) ;
5388       return [ ] ;
5389    else
5390       access_type := arb a_types ;
5391    end if ;
5392    -- It is always illegal to dereference an out parameter.
5393    out_c := out_context ;
5394    out_context := false ;
5395    type2 := designated_type(access_type) ;
5396    expn2 := [ '@', resolve2(ac_expn, access_type) ] ;
5397    out_context := out_c ;
5398
```

-RESOLVE2- allocator

```
5400    -- For an allocator, we obtain the type of the access object
5401    -- by dereferencing the access type. The final expression however
5402    -- gives the access type, together with the validated access object.
5403
5404    ('new'): [-, expn1, alloc_type] := expn ;
5405    if not is_access(context_typ) then
5406        errmsg('Context of allocator ', ' must be an access type',
5407                                '4.8, 3.8') ;
5408        return [ ] ;
5409    end if ;
5410
5411    accessed_type := designated_type(context_typ) ;
5412
5413    -- Verify that the allocator matches the context.
5414
5415    -- The (possibly unconstrained) access type is the one given by the
5416    -- context (eg. declaration). If the allocator provides a constraint
5417    -- rather than an aggregate, then a subtype has been created, and the
5418    -- access type is an access to this constrained type. The constraint
5419    -- must then be emitted so that it is evaluated at the proper time.
```

```
5420    -- (The subtype is not an anonymous type, and is introduced only to
5421    -- simplify type checking).
5422    -- The converse may also occur: the context is constrained, but the
5423    -- allocator type is unconstrained. In that case, use the context
5424    -- context type as the type of the expression.
5425    -- Finally, the context may be an unconstrained array type, whose
5426    -- index type is nevertheless bounded. When the allocator is
5427    -- initialized with an aggregate, the bounds of the aggregate must be
5428    -- compatible with that index type.
5429                                                              --
5430    if not compatible_types(accessed_type, alloc_type) then
5431        errmsg('Invalid type for allocator. Expect ',
5432                        full_type_name(accessed_type),'3.8, 4.8') ;
5433        return [ ] ;
5434
5435    elseif accessed_type = alloc_type
5436       -- no constrained type was created in allocator.
5437
5438    or is_unconstrained(accessed_type) then
5439        -- Alloc_type is a subtype of the original designated type.
5440       obj_type := alloc_type ;
5441
5442    else                              -- designated type is constrained
5443        obj_type := accessed_type ;
5444    end if ;
5445
5446    type_expr := subtype_expr(alloc_type) ;
5447
5448    if expn1 = [ ] then
5449       access_object := [ ] ;
5450    else
5451        access_object := res2_check(expn1, obj_type) ;
5452 --
5453       if is_array(accessed_type) and can_constrain(accessed_type)
5454         then
5455            -- bounds of the aggregate will have to be shown to be
5456          -- compatible with the (unconstrained) designated type.
5457            access_object :=
5458                ['qual_sub',index_types(accessed_type),access_object] ;
5459       end if ;
5460    end if ;
5461
5462    expn2 := ['new', context_typ, type_expr, access_object] ;
5463    type2 := context_typ ;                    -- No further constraints
5464
5465
```

-RESOLVE2- attribute

```
5467    -- For an attribute, we complete the type checking of the right
5468    -- argument, and if it must be a static expression, we perform
5469    -- the appropriate check and extract the attribute.
5470
5471    ("'"): [-, attribute, arg11, arg21] := expn ;
5472
5473    -- The type of the right argument is determined by the attribute,
5474    -- and has already been evaluated in the case of array attributes.
5475
5476    [arg1, types1] := arg11 ;
5477    [arg2, types2] := arg21 ;
5478    type1 := arb types1 ;
5479
5480    case attribute of
5481
5482    ('PRED', 'SUCC', 'POS', 'IMAGE'):                              --
5483        t2 := base_type(type1) ;
5484
5485    ('VALUE'):
5486        t2 := 'STRING' ;
5487
5488    ('VAL'):
5489        types2 := {t ∈ types2 | compatible_types(t, 'integer_type')} ;
5490      if types2 = {} then
5491          errmsg('Second argument of VAL must be ',
5492                                      'of some integer type', 'Annex A') ;
5493          return [ ] ;
5494        elseif # types2 = 1 then
5495          t2 := arb types2 ;
5496        elseif 'universal_integer' ∈ types2 then
5497          t2 := 'universal_integer' ;
5498        else
5499          errmsg('ambiguous argument for attribute VAL','','Annex A');
5500          return [ ] ;
5501        end if ;
5502
5503    else
5504        t2 := 'INTEGER' ;
5505
5506    end case ;
5507
5508    arg2 := if is_value(arg21) then arg21
5509                                else resolve2(arg21, t2)
5510    end ;
5511    if t2 = 'universal_integer' then                -- possible for VAL
5512        arg2 := specialize(arg2, 'INTEGER') ;
```

```
5513    end ;
5514
5515    -- If the left argument is a type, or if it is a constrained
5516    -- object, then evaluate the attribute on the type, statically if
5517    -- possible.
5518
5519    -- All attributes, except those that are functions, can be applied
5520    -- to an out parameter, because they do not require reading of the
5521    -- object, or read only its bounds. On the other hand, if the pre-
5522    -- fix is an access type, it cannot be an an out parameter (4.1(4)).
5523
5524    out_c := out_context ;                          -- Save current setting
5525    out_context := attribute   base_attributes
5526                                        and not is_access(type1) ;
5527    itype1 := type1;
5528    if is_array(type1) and
5529      attribute ∈ {'RANGE', 'FIRST', 'LAST', 'LENGTH'} then
5530
5531        -- The second argument indicates the dimension whose attribute
5532        -- is sought. It must be a static integer (this has been checked
5533        -- already).
5534        if not is_value(arg2) then
5535            dim := 1 ;                  -- By default.
5536        else
5537            dim := arg2(2) ;
5538        end if ;
5539        ctype1 := type1;
5540        itype1 := index_types(type1)(dim) ;
5541    end if ;
5542
5543    if is_identifier(arg1) and is_type(arg1)                    --
5544    then
5545        expn2 := ["'", attribute, itype1, arg2] ;
5546
5547    elseif attribute = 'COUNT' then
5548        -- entry name is fully resolved in first pass.
5549        expn2 := ["'", attribute, arg1, arg2] ;
5550
5551    elseif attribute = 'ADDRESS' then
5552        if is_overloaded(arg1) then             -- subprogram or entry
5553            arg1 := arb arg1;
5554        elseif not is_identifier(arg1) then     -- subcomponent of entity
5555            arg1 := resolve2(arg11, type1) ;
5556        end if ;                                -- else declared entity
5557        expn2 := ["'", attribute, arg1, arg2] ;
5558
5559    else
5560      expn2 := ["'", attribute, resolve2(arg11, type1), arg2] ;
5561    end if ;
```

```
5562    out_context := out_c ;              -- restore              -
5563
5564 --
5565    if type_of(attribute) ∈ univ_types
5566      and is_value(e := eval_static(expn2)) then              --
5567        -- Specialize value if context is not universal.
5568        expn2 := specialize(e, context_typ) ;
5569    end if ;
5570    [-,-,arg1] := expn2 ;
5571    if attribute ∈ ['FIRST_BIT', 'LAST_BIT'] and not is_component(arg1)
5572    then
5573        errmsg('argument of ' + attribute, ' must be record component',
5574                                                    'Annex A');
5575    end if ;
5576    type2 := arb types ;
5577
```

-RESOLVE2- conversion

```
5579    -- A conversion may imply a run-time action, or may be used
5580    -- between types of the same structure to achieve type consistency.
5581    -- In the later case, do not emit any conversion.
5582    -- In both cases however, a range check may be needed.
5583
5584    ('convert'): [-, target_type, expn1] := expn ;
5585    types := expn1(2) ;
5586
5587    -- Apply the preference rule to choose a universal meaning for
5588    -- the expression in case of overloading of operators.
5589
5590    if # types > 1 and # (tu := types * univ_types) = 1
5591    then
5592        types := tu ;
5593    end if ;
5594    -- Verify that original expression is unambiguous.
5595    if #types /= 1 then
5596        errmsg('ambiguous expression for conversion','','4.6') ;
5597        return [ ] ;
5598    else
5599        t := arb types ;
5600
5601        if is_numeric(t) and is_numeric(target_type)
5602          -- conversions between any  two numeric types are allowed.
5603      then
5604          expn2 :=
5605          eval_static(['convert', t, target_type, resolve2(expn1,t)]);
5606
5607        elseif root_type(target_type) = root_type(t)
5608          -- conversions among types derived from a common root. In the
```

```
5609            -- absence  of representation specifications, this is a noop,
5610            -- indicated (arbitrarily) by the same type on both sides.
5611        then
5612            expn2 := ['convert', t, t, resolve2(expn1, t)] ;
5613
5614        elseif is_array(target_type)
5615            -- Conversions between array types  are allowed, if types of
5616           -- indices are convertible and component types are the same.
5617        then
5618            if is_array(t) and
5619             no_dimensions(t) = no_dimensions(target_type)
5620             and
5621             (forall i ∈ [1..no_dimensions(t)] |
5622               root_type(index_types(target_type)(i)) =
5623                                        root_type(index_types(t)(i)))
5624             and
5625               base_type(component_type(target_type)) =
5626                                        base_type(component_type(t))
5627            then
5628                expn2 := resolve2(expn1,t) ;
5629            else
5630                errmsg('Invalid array conversion','','4.6') ;
5631                return [ ] ;
5632            end if;
5633
5634        else
5635            errmsg('conversion not possible to type ',
5636                                original_name(target_type), '4.6') ;
5637        end if ;
5638        type2 := target_type ;
5639    end if ;
5640
5641    expn2 := apply_constraint(expn2, target_type) ;          --
5642
5643    ('qualify'):
5644    [-, to_type, expn1] := expn ;
5645    if is_incomplete_type(type2) then                        --
5646        premature_access(type2) ;
5647    end if ;
5648 --
5649    -- No sliding for aggregates here.
5650    may_others := full_others ;
5651    full_others := true ;
5652    expn2 := apply_constraint(resolve2(expn1, to_type), to_type);
5653    full_others := may_others ;                                  --
5654
5655    if is_identifier(expn2) then          -- a qualified expression
5656        expn2 := ['()', expn2];                        -- is not a variable.
5657    end if ;
```

```
5658    type2 := to_type ;
5659
```

-RESOLVE2- subtype

```
5661    -- For a subtype, complete the evaluation of the bounds.
5662    -- If the bounds are literal,the type may be a universal one.
5663    -- replace it now by the corresponding non-literal type.
5664
5665    ('subtype'):
5666    [-,typ1, [kind, low1, high1,precision]] := expn ;
5667
5668    -- If the bounds are overloaded, the subtype itself may be an
5669    -- overloaded expression. Extract the type(s) that are compatible
5670    -- with context .
5671
5672    Types := {t:t ∈ types | compatible_types(context_typ, t)} ;--
5673
5674    -- Make sure that only one type is possible.
5675    if #types > 1 then types -:= univ_types ; end if ;          --
5676
5677    if #types /= 1 then
5678        type_error({'subtype'}, context_typ, #types) ;
5679        return [ ] ;
5680    else
5681        b_type := base_type(arb types) ;
5682    end if ;
5683    -- In the case of a range in a membership op, the type may be a real
5684    -- one, in which case the precision is inherited from the context .
5685
5686    if kind = 'range' then
5687        case root_type(context_typ) of
5688
5689            ('FLOAT', 'universal_real'):
5690                kind := 'digits' ;
5691               precision := signature(context_typ)(4) ;
5692
5693            ('$FIXED'):
5694                kind := 'delta' ;
5695                precision := signature(context_typ)(4) ;
5696        else
5697            pass ;                                  -- Discrete type.
5698        end case ;
5699    end if ;
5700
5701    if typ1 /= [ ] then b_type := typ1;
5702    else case(kind) of
5703
5704        ('range'):
```

```
5705            if b_type = 'universal_integer' then
5706                b_type := 'INTEGER';
5707                [l1, -] := low1 ;
5708                [h1, -] := high1 ;
5709                if context_typ = 'universal_integer' and
5710                  -- I.e. index constraint or iteration.
5711                  (is_tuple(l1) and is_overloaded(l1(1)) or
5712                   is_tuple(h1) and is_overloaded(h1(1))) then
5713                   -- Not a literal, named number or attribute.
5714                   errmsg('Invalid universal expression ',
5715                                  'in discrete range', '3.6.1') ;
5716                end if ;
5717            end if ;
5718
5719        ('delta'):  b_type := '$FIXED' ;
5720        ('digits'): b_type := 'FLOAT' ;
5721
5722        end case ;
5723    end ;
5724    -- If the type name was not specified, then it is the type
5725    -- of the bounds.
5726
5727    if typ1 = [ ] then typ1 := b_type ; end if ;
5728
5729
5730    low := resolve2(low1, b_type) ;
5731    high:= resolve2(high1,b_type) ;
5732
5733    -- For a subtype, the check for an empty range preceeds the
5734    -- check for the bounds being individually in range. Therefore
5735    -- we suppress range check on the upper bound expression.
5736
5737    --- if low(1) = 'qual_range' then low := low(3) ; end if ;
5738    if high(1)= 'qual_range' then high:= high(3); end if ;
5739
5740    -- An array constraint may be expressed by a discriminant name.
5741    -- Verify that if a discriminant appears, it is by itself, and not as
5742    -- part of a larger expression.
5743    check_discriminant(low) ;
5744    check_discriminant(high);
5745    -- The type of the subtype construct is the unique type of the bounds
5746    type2 :=  context_typ := b_type ;
5747    expn2 := ['subtype', typ1, [kind, low, high, precision]] ;
5748
5749    ('call_or_index'):
5750    -- Find the tree which has a type compatible with the context, and
5751    -- resolve it.
5752    [-, trees1] := expn ;
5753    valid_trees := { tree ∈ trees1 | (exists t ∈ tree(2) |
```

```
5754                          compatible_types(context_typ, t)) } ; --
5755    if # valid_trees = 1 then
5756       return resolve2(arb valid_trees, context_typ) ;
5757    elseif # valid_trees > 1 then
5758       errmsg('invalid or ambiguous call',",") ;
5759       return [ ] ;
5760    end if ;
5761    else                    -- Other operators require no propagation
5762     expn2 := expn ;
5763     type2 := arb types ;
5764    end case;
5765    end if ;
5766 end if ;
5767
```

-RESOLVE2- constant fold and note type of expression

```
5769
5770 if compatible_types(context_typ, type2) then                    --
5771    e := eval_static(expn2) ;
5772    return if is_tuple(e) then (e with context_typ) else e end ;
5773 else
5774    errmsg( 'Incorrect type for expression. Expect ' ,
5775                                  full_type_name(context_typ),'none') ;
5776    return [ ] ;
5777 end if ;
5778
5779 end proc resolve2;
```

result_types

```
5781 proc result_types(op_name, arg_list) ;
5782
5783 -- This procedure performs the first pass of type resolution on over-
5784 -- loadable constructs : operators, subprograms, aggregates, and
5785 -- literals.
5786
5787 const error_expn = [[ ], {}] ;
5788
5789
5790
5791 -- Check for previous type error.
5792
5793 if (noop_error ? false) then return error_expn ; end if ;
5794
5795 ops := types := {} ;
5796
5797 (forall opn ∈ op_name )
5798
```

```
5799  case nature(opn) of
5800
5801   ('op'):
5802       [opns, valid] := valid_op_types(opn, arg_list) ;
5803
5804       -- A predefined operator is usable if its resulting types appears
5805       -- in a lexically open scope, or a used package.
5806
5807       usable := {t ∈ valid | (sct := scope_of(t)) ∈ open_scopes
5808                                          or  sct ∈ used_mods} ;
5809
5810       if usable = {} and #valid = 1 and #op_name = 1 then
5811         return pass1_error('operator ' + opn + ' on type ' +
5812                                        original_name(arb valid) ,
5813              ' is not directly visible', '6.6, 8.3, 8.4') ;
5814       else
5815          ops +:= opns ;
5816          types +:= usable ;
5817       end if ;
5818
5819   ('procedure', 'procedure_spec', 'function', 'function_spec',
5820                                  'entry',   'entry_family'   ):
5821
5822      typ := valid_arg_list(opn, arg_list) ;
5823       if typ /= om then
5824          types with:= typ ;
5825          ops with:= opn ;
5826       end if ;
5827
5828
5829   ('literal') :
5830       -- A literal may overload a function. The literal is valid only
5831       -- if the argument list is empty.
5832
5833       if is_empty(arg_list) then
5834          types with:= type_of(opn) ;
5835          ops with:= opn ;
5836       end if ;
5837
5838   else
5839       -- If the overloaded object has no given nature, it is an infamous
5840       -- use of P.'op' for a predefined operator. Resolve as any operator,
5841      -- and verify that the resulting type is indeed defined inside P,
5842       -- or is the predefined BOOLEAN.
5843
5844      desig := rbreak(opn, '.') ;
5845       package := opn(1..#opn-1) ;
5846
5847      [opns, valid] := valid_op_types(desig, arg_list) ;
```

```
5848 --
5849          usable := {t ∈ valid | scope_of(t) = package or t = 'BOOLEAN'};
5850          -- Should check that the type of an argument is in named scope.
5851          ops +:= opns ;
5852          types +:= usable ;
5853
5854 end case ;
5855 end forall ;
5856
5857 if types = {}
5858     and exists arg ∈ arg_list | 'universal_fixed' ∈ arg(2) then
5859          errmsg('Missing explicit conversion ',
5860                              'from universal fixed value','3.5.9, 4.5.5') ;
5861     noop_error := true ;
5862 -- other errors are caught at the point of call of this procedure.
5863 end if ;
5864
5865
5866 return [ [ops] + arg_list, types] ;
5867
5868 end proc result_types ;
```

disambiguate

```
5870 proc disambiguate(valid_ops, arg_list, context_typ) ;
5871
5872 -- Called from resolve2, when more than one operator or function is
5873 -- compatible with the context type. Apart from true ambiguity, this
5874 -- can also happen if both a predefined and a user-defined operator are
5875 -- visible. This is because all predefined operators in the language have
5876 -- generic signatures (e.g. universal_ integer rather than INTEGER) and
5877 -- as a result, a user-defined operator does not hide the corresponding
5878 -- operator (they do not have the same signature). The solution is to
5879 -- choose in favor of the user-defined op. if it is defined in the sa-
5880 -- me package as the type, or in an open scope, and in favor of the pre-
5881 -- defined one otherwise. For comparison operators which yields the pre-
5882 -- defined type BOOLEAN, the above reasoning applies to the type of its
5883 -- formals and not to the boolean context.
5884 --
5885 -- On the other hand, a predefined operator of (generic) type o_t may be
5886 -- compatible with arguments of type a_t and with the context c_t, while
5887 -- a_t is in fact not compatible with c_t. To catch that case, we check
5888 -- valid_op_types again to verify that the result is compatible with the
5889 -- context.
5890 --
5891 -- A final wrinkle: if the context is universal, as in a number declara-
5892 -- tion, then the predefined operator is used even if a user-defined one
5893 -- is in scope.
5894 valid_ops := [opn ∈ valid_ops | (nature(opn) /= 'op')
```

```
5895                          or compatible_op(opn, arg_list, context_typ) ] ;
5896 if context_typ ∈ univ_types then
5897    valid_ops := [opn ∈ valid_ops | type_of(opn) = context_typ] ;
5898
5899 elseif exists nam ∈ valid_ops |
5900  nature(nam) /= 'op' and
5901 ((sc := scope_of(nam)) = scope_of(context_typ)
5902  or sc ∈  open_scopes
5903  -- maybe a comparison op. Check against scope of type of first formal.
5904  or (type_of(nam) = 'BOOLEAN'
5905    and sc = scope_of(type_of(signature(nam)(1)(1)))))
5906 then
5907    -- user-defined operator(s) hide derived operator.
5908    valid_ops := [nam ∈ valid_ops | nature(nam) /= 'op'] ;
5909
5910 elseif exists nam ∈ valid_ops | nature(nam) = 'op'
5911    then
5912    -- It will have precedence over imported user-defined functions
5913    valid_ops := [ nam ∈ valid_ops | nature(nam) = 'op'] ;
5914
5915    if root_type(context_typ) = '$FIXED' then
5916      -- remove mixed floating operators, that yield universal real, but
5917      -- are not compatible with a fixed type context
5918      valid_ops := [nam ∈ valid_ops |type_of(nam) /= 'universal_real'];
5919    end if ;
5920
5921 end if ;
5922
5923 return valid_ops;
5924 end proc disambiguate ;
```

compatible_op

```
5926 proc compatible_op(opn, arg_list, context_typ) ;
5927
5928
5929 -- In most cases, binary operators are homogenenous: the type of their
5930 -- arguments is also  the type of the result. We get the types  of the
5931 -- arguments to perform this test:
5932
5933 [-, types1] := arg_list(1) ;
5934 if #arg_list = 2 then [-, types2] := arg_list(2) ; end if ;
5935
5936 -- For comparison operators, the types of the operands are known to be
5937 -- compatible and unrelated to the boolean result.
5938
5939 return (exists t ∈ types1 | compatible_types(t, context_typ))  --
5940       or opn ∈ comparison_ops
5941
```

```
5942        or opn ∈ ['*ifl', '*ifx'] and
5943        -- For these ops, the second argument yields the result type.
5944        (exists t ∈ types2 | compatible_types(t, context_typ))   --
5945
5946        or opn = '&' and  --
5947        -- For concatenation, the first argument may be an array component
5948        (exists t ∈ types1 | is_array(context_typ) and
5949              compatible_types(t, component_type(context_typ))); --
5950
5951 end proc compatible_op;
```

remove_conversions

```
5953 proc remove_conversions(valid_ops, arg_list) ;
5954
5955 -- If after the previous procedure an expression is still ambiguous, this
5956 -- may be due to an implicit conversion of a universal quantity. This can
5957 -- only happen in the presence of user-defined operators. We therefore
5958 -- attempt to resolve the expression again, after removing user-defined
5959 -- operators from the tree, whose arguments are universal quantities.
5960 -- A full disambiguation would require that we try to remove these selec-
5961 -- tively. Here we simply remove all of them, and give up if the result
5962 -- is still ambiguous.
5963
5964
5965 new_arg_list := [ ] ;
5966 (for arg ∈ arg_list)
5967
5968     [arg_expn, arg_types] := arg;
5969     if # arg_types < 2 then pass;                -- unambiguous.
5970     else
5971         [arg_op, -] := arg_expn ;
5972         a_list := arg_expn(2..) ;
5973
5974         if not is_overloaded(arg_op) then pass;
5975           -- Incomplete: could be an indexing on an overloaded call!
5976
5977         elseif original_name(arb arg_op)   op_designators then
5978           -- May be overloaded because some of its arguments are.
5979           arg := remove_conversions(arg_op, a_list) ;
5980
5981         elseif exists [-, ts] ∈ a_list | ts * univ_types /= {}
5982         then
5983           -- Some argument is universal. Resolve as predefined operator
5984           arg_op := {n ∈ arg_op | nature(n) = 'op'} ;
5985           arg := result_types(arg_op, a_list) ;
5986         end if ;
5987
5988     end if ;
```

```
5989     new_arg_list with:= arg ;
5990 end for ;
5991
5992 -- Use the pruned argument list to resolve again the expression.
5993 return result_types(valid_ops, new_arg_list);
5994
5995 end proc remove_conversions;
```

valid_op_types

```
5997 proc valid_op_types(opn, arg_list) ;
5998 const    unary_sig  = [['RIGHT', 'in', 'any', [ ]] ] ,
5999          binary_sig = [['LEFT',  'in', 'any', [ ]],
6000                        ['RIGHT', 'in', 'any', [ ]] ],
6001         unary_ops  = ['+', '-', 'abs', 'not'] ;
6002
6003
6004 -- This procedure is invoked during the bottom-up pass of type
6005 -- resolution. It determines the possible result types of predefined
6006 -- operators, using the possible types of their arguments.
6007 -- All arithmetic operators have special rules that apply within literal
6008 -- expressions. They are all treated in routine valid_arith_ops.
6009 -- For other operators, the following rule applies:
6010 -- Binary operators yield the intersection of the types of their two
6011 -- arguments, provided that they are boolean (For boolean operators),
6012 -- discrete (for ordering operators) , etc.
6013 -- The concatenation operator provides an exception : it will
6014 -- concatenate and array with an object of the component type, either
6015 -- on the left or right.
6016
6017
6018 if opn ∈ unary_ops and #arg_list = 1 then
6019     arg_list := order_arg_list(arg_list, unary_sig) ;
6020 else
6021     arg_list := order_arg_list(arg_list, binary_sig) ;
6022 end if ;
6023 if arg_list = om then
6024          return [{}, {}] ;                    -- parameter list mismatch.
6025 end if ;
6026
6027 if type_of(opn) = 'numeric' then
6028     return valid_arith_types(opn, arg_list) ;
6029 end if ;
6030
6031 if #arg_list = 1 then
6032     [-, typ1] := arg_list(1) ;
6033     possible_types := {base_type(t): t ∈ typ1} ;
6034
6035 else
```

```
6036    -- Binary operator.
6037
6038    [ [-, typ1], [-, typ2]] := arg_list ;
6039
6040    if opn = '&' then
6041
6042      -- If the first argument is an array, the second must be the same
6043      -- array, or have the component type of array.
6044
6045      types := { base_type(t1) :t1 ∈ typ1 | is_array(t1) and
6046                   (exists t2 ∈ typ2 |
6047                     compatible_types(t1,t2) or
6048                     compatible_types(component_type(t1),t2))}      --
6049
6050      -- The second may be an array, and the first have its component
6051      -- type.
6052
6053            + { base_type(t2): t2 ∈ typ2 | is_array(t2) and
6054                 (exists t1 ∈ typ1 | not is_array(t1) and      --
6055                   compatible_types(component_type(t2),t1))}      --
6056
6057      -- If neither is an array, then the result is some one-dimensional
6058      -- array type with known component type.
6059
6060            + if exists t1 ∈ typ1, t2 ∈ typ2 |
6061                      not is_array(t1) and not is_array(t2) then
6062                 {t ∈ find_agg_types() | nature(t) = 'array'
6063                    and no_dimensions(t) = 1
6064                    and (exists t1 ∈ typ1, t2 ∈ typ2 |
6065                      compatible_types(component_type(t),t1)  --
6066                     and
6067                      compatible_types(component_type(t),t2))}
6068               else {} end ;
6069        types := {t ∈ types | not is_limited_type(t)} ;    --
6070        return [{opn}, types] ;
6071    else
6072
6073        -- All other binary operators are homogeneous : the arguments
6074        -- must have compatible types,
6075        possible_types:= {base_type(t) : t ∈ typ1 |
6076              (exists t2 ∈ typ2 | compatible_types(t,t2))}; --
6077
6078    end if ;
6079 end if;
6080 -- Remove array types with incomplete private components.
6081 possible_types := {t ∈ possible_types |
6082              not is_array(t) or not is_fully_private(t) } ;    --
6083
6084 case type_of(opn) of
```

```
6085  ('BOOLEAN') :              -- equality and membership operators.
6086
6087      types := case opn of
6088
6089          ('=', '/='):
6090              (if exists t ∈ possible_types | not is_limited_type(t)
6091                  then {'BOOLEAN'} else {} end)
6092
6093          else
6094              (if possible_types /={} then {'BOOLEAN'} else {} end)
6095          end ;
6096
6097  ('boolean_type'):          -- Boolean and short circuit operators.
6098      types := case opn of
6099
6100          ('andthen', 'orelse'):
6101              { t : t ∈ possible_types | root_type(t) = 'BOOLEAN'}
6102
6103          else
6104            { t : t ∈ possible_types | root_type(t) = 'BOOLEAN' or
6105            is_array(t) and no_dimensions(t) = 1
6106                        and root_type(component_type(t)) = 'BOOLEAN'}
6107      end ;
6108  ('order_type'):            -- Comparison operators.
6109
6110      if exists t ∈ possible_types | (is_scalar_type(t)
6111          or is_array(t) and no_dimensions(t) = 1
6112                        and is_discrete_type(component_type(t)))
6113      then types := {'BOOLEAN'} ;
6114      else types := {} ;
6115      end if ;
6116
6117  else errmsg('system error: strange op type ', type_of(opn),'none') ;
6118
6119  end case ;
6120
6121  return if types /= {} then [{opn}, types] else [{}, {}] end ;
6122
6123 end proc valid_op_types;
```

valid_arith_types

```
6125 proc valid_arith_types(opn, arg_list) ;
6126
6127 -- Bottom-up pass over arithmetic expressions. return the pair:
6128 -- [possible operators, possible result types] .
6129
6130 macro i ; 'INTEGER'        endm ;
6131 macro fl; 'FLOAT'          endm ;
```

```
6132 macro fx; '$FIXED'          endm ;
6133 macro ui ; 'universal_integer' endm ;
6134 macro ur ; 'universal_real'    endm ;
6135 macro ufx; 'universal_fixed'   endm ;
6136
6137 const numeric_types = {i,fl,fx,ui,ur} ,
6138     universal_types = {ui, ur} ,
6139
6140     adding_types = { [i , i ], [fl, fl], [fx, fx], [ui, i],
6141                      [ui, ui], [ur, ur], [ur, fx], [ur, fl]} ,
6142
6143     mult_types   = { [i , i ], [fl, fl], [fx, fx], [ui, i ],
6144                      [ui, ui], [ur, ur], [ur, fl], [ur, fx] } ,
6145
6146     mixed_mult_types = { [fx, i], [fx, ui], [ur, ui], [ur, i]} ,
6147
6148     mod_types    = { [i, i], [ui, i], [i, ui], [ui, ui]} ,
6149
6150     expon_types    = { [i , i ], [fl, i ], [ur, i ],[ui, i ],
6151                        [i , ui], [fl, ui], [ur, ui],[ui, ui]  } ,
6152
6153
6154     op_suffix    = { [i, 'i'], [ui, 'i'], [fl, 'fl'], [ur, 'fl'],
6155                      [fx, 'fx'] , [ufx, 'fx']} ;
6156
6157
6158
6159
6160
6161 if  #arg_list = 1 then                          -- Unary ops return the type
6162                                                 -- of their argument.
6163     [-,possible_types] := arg_list(1) ;
6164 --
6165     types := {base_type(t) : t ∈ possible_types |
6166                                          root_type(t) ∈ numeric_types} ;
6167
6168     -- Construct the unary version of the operator name.
6169     if opn ∈ {'+','-'} then opn +:= 'u' ; end if;
6170
6171     ops := { opn + op_suffix(root_type(t)): t ∈ types} ;
6172     return [ ops, types] ;
6173 else
6174     [ [-, typ1], [-, typ2]] := arg_list ;
6175
6176     ops := types := {} ;
6177
6178     (forall t1 ∈ typ1, t2 ∈ typ2)
6179
6180         sfx := '' ;                 -- Suffix to designate type of op.
```

```
6181        r_type := om ;          -- Will indicate type found.
6182
6183        bt1 := root_type(t1) ;
6184        bt2 := root_type(t2) ;
6185
6186        case opn of
6187
6188          ('+', '-'):
6189
6190             if [bt1, bt2] ∈ adding_types or
6191               [bt2, bt1] ∈ adding_types then
6192                    r_type := intersect_types(t1,t2) ;
6193             end if ;
6194
6195          ('*', '/'):
6196
6197             if [bt1, bt2] ∈ mult_types or
6198               [bt2, bt1] ∈ mult_types  then
6199                    if '$FIXED' ∈ {bt1,bt2} then
6200                         r_type := 'universal_fixed' ;
6201                    else r_type := intersect_types(t1,t2) ;
6202                    end if ;
6203             else -- Mixed mode operation on fixed types, or
6204                         -- literal expression.
6205
6206                  if [bt1, bt2] ∈ mixed_mult_types then
6207
6208                     if {bt1,bt2} = universal_types then        -- Literal expr.
6209                        r_type := ur ;
6210                          sfx := 'fli' ;          -- Compile-time op.
6211
6212                     elseif base_type(t2) = 'INTEGER' then
6213                          -- Mixed mode operation with a fixed type. If
6214                          -- the first argument is universal, the result is
6215                          -- -FIXED, i.e any fixed type.
6216
6217                        r_type := if t1 = ur then fx else t1 end ;
6218                          sfx := 'fxi' ;          -- Run-time operation.
6219
6220                     elseif bt2 = ui then-- Specific fixed on left
6221                          r_type := t1 ;
6222                          sfx    := 'fxi' ;
6223                     end if ;
6224
6225                  elseif [bt2, bt1] ∈ mixed_mult_types and opn = '*'
6226                  then
6227                  -- Mixed modes are not commutative for division.
6228
6229                     if {bt1, bt2} = universal_types then
```

```
6230                        r_type := ur ;
6231                        sfx := 'ifl' ;
6232
6233                    elseif base_type(t1) = 'INTEGER' then
6234                        -- FIXED, or the specific fixed type t2.
6235                        r_type := if t2 = ur then fx else t2 end ;
6236                        sfx := 'ifx' ;
6237
6238                    elseif bt1 = ui then-- Specific fixed right
6239                        r_type := t2 ;
6240                        sfx    := 'ifx' ;
6241                    end if ;
6242                end if ;
6243
6244            end if ;
6245
6246        ('mod', 'rem'):
6247
6248            if [bt1,bt2] ∈ mod_types then
6249                r_type := intersect_types(t1,t2) ;
6250            end if ;
6251
6252        ('**'):
6253            -- The result of an exponentiation has the type of the first
6254            -- argument.
6255
6256            if [bt1, bt2] ∈ expon_types then
6257                r_type := t1 ;
6258            end if ;
6259
6260        end case ;
6261
6262        if r_type /= om then            -- Pair of matching types found.
6263
6264            -- The result type of an arithmetic operation does not carry
6265            -- the constraint (if any) of the arguments. Therefore, drop
6266            -- the constraint on the result if it appears as a subtype.
6267
6268            types with:= base_type(r_type) ;
6269
6270            -- Append to the operator name a suffix that specifies the
6271            -- type of its arguments and the type returned.
6272            if sfx = '' then sfx := op_suffix(root_type(r_type)) ; end ;
6273            ops with:= opn + sfx ;
6274        end if ;
6275    end forall ;
6276
6277 end if ;
6278
```

```
6279 return [ops, types] ;
6280
6281 drop i, fl, fx, ui, ur, ufx ;
6282
6283 end proc valid_arith_types ;
6284
```

intersect_types

```
6286 proc intersect_types(t1,t2) ;
6287
6288 -- Find the more specific of two numeric types, if they are compatible.
6289 -- In particular, if only one of them is universal, return the other.
6290 -- Called to validate arithmetic arguments and bounds of subtypes.
6291 -- Also used in aggregate resolution, when an aggregate is being
6292 -- compared with an expression of some composite type.
6293
6294 const universal_types =
6295    {'universal_integer', 'universal_real', '$FIXED', 'composite_type'} ;
6296
6297 return
6298     if compatible_types(t1,t2) then                    --
6299          (if    t1 ∈ universal_types then t2
6300          elseif t2 ∈ universal_types then t1
6301          else t1 end)
6302     else om
6303 end ;
6304
6305 end proc intersect_types ;
```

valid_arg_list

```
6307 proc valid_arg_list(proc_name, arg_list) ;
6308
6309 -- First pass resolution over subprogram calls. If the argument list is
6310 -- compatible with the parameter profile, return the type of the sub-
6311 -- program. Otherwise return om.
6312
6313
6314 formals := signature(proc_name) ;
6315
6316 arg_list := order_arg_list(arg_list, formals) ; -- Normalize arguments
6317
6318 if arg_list = om then          -- wrong number of actuals, or error.
6319     return om ;
6320 end if ;
6321
6322 -- Traverse signature and actuals, and verify that types match.
6323
```

```
6324 (forall [-, -, f_type, -] = formals(i))
6325     actual := arg_list(i) ;
6326     if actual = [ ] then continue forall ;        -- Default value exists.
6327     else [-, a_types] := actual ;
6328     end if ;
6329     if exists t ∈ a_types | compatible_types(t, f_type) then    --
6330         continue ;                                -- argument is valid
6331     else
6332         return om ;                               -- type mismatch.
6333     end if ;
6334 end forall ;
6335 -- On exit from the loop, all arguments have a match.
6336 return type_of(proc_name) ;
6337
6338 end proc valid_arg_list;
6339
6340
```

complete_arg_list

```
6342 proc complete_arg_list(formals, arg_list) ;
6343
6344 -- Second pass over argument list. Complete resolution of each argument,
6345 -- and apply required constraints on actuals.
6346
6347
6348 arg_list := order_arg_list(arg_list, formals) ;  -- Normalize arguments.
6349 complete_args := [ ] ;
6350
6351 -- Complete type resolution of each actual, and insert default expression
6352 -- for those that are missing; default expressions are known to exist.
6353
6354 (forall [-,f_mode, f_type, f_init] = formals(i))
6355     actual := arg_list(i) ;
6356
6357     if actual = [ ] then                    -- Insert default expression
6358         complete_args with:= f_init ;
6359     else
6360         arg := bind_arg(actual, f_type, f_mode) ;
6361         if f_mode /= 'in' and not is_variable(arg) then
6362             errmsg(f_mode + ' actual parameter no.' + str i,
6363                         ' in call is not a variable', '6.4.1') ;
6364         end if ;
6365         complete_args with:= arg ;
6366     end if ;
6367 end forall ;
6368
6369 return complete_args ;
6370 end proc complete_arg_list ;
```

6371

order_arg_list

```
6373 proc order_arg_list(arg_list, sig) ;
6374
6375 -- Normalize an argument list (possibly containing named associations)
6376 -- according to the signature -sig-. Called for subprogram and operators.
6377
6378
6379 if exists actual = arg_list(i) | actual(1) = 'choice_list' then
6380     first_named := i ;
6381     if exists actual ∈ arg_list(i+1..) | actual(1) /= 'choice_list'
6382     then
6383         errmsg('No positional arguments can appear after named ones',
6384                                             '','6.4') ;
6385         return om ;
6386     end if ;
6387 else
6388     first_named := #arg_list + 1 ;          -- i.e. all positional
6389 end if ;
6390
6391 new_list := arg_list(1..first_named-1) ;    -- In their proper place.
6392 named_args := arg_list(first_named..) ;
6393 actuals_seen := first_named-1;                      -- Seen so far.
6394
6395 (forall [-,choice_list,-] ∈ named_args)
6396     if #choice_list > 1 or choice_list(1)(1) /= 'simple_choice' then
6397         errmsg('Invalid format for argument association','','6.4') ;
6398         return om  ;
6399     end if ;
6400 end forall ;
6401
6402 (forall i ∈ [first_named..#sig])
6403     [f_name,-,-,f_init] := sig(i) ;
6404     if exists
6405         [-, [[choice_kind, [a_name,-]]], a_expr] ∈ named_args
6406                                 | original_name(f_name) = a_name
6407     then
6408         new_list with:= a_expr ;
6409         actuals_seen +:= 1;
6410         check_void(a_name) ;                        --
6411
6412     elseif f_init /= [ ] then
6413         new_list with:= [ ] ;           -- marker to indicate default
6414     else                            -- Name is not present
6415         return om ;
6416     end if;
6417 end forall ;
```

```
6418
6419 if actuals_seen = #arg_list          -- all actuals processed.
6420   and #new_list = #sig then          -- all formals matched.
6421     return new_list;
6422 else return om;
6423 end if;
6424
6425 end proc order_arg_list;
6426
```

complete_op_expr

```
6428 proc complete_op_expr(op_name, ctx_type, arg_list) ;
6429
6430
6431 -- Complete the top-down pass of an expression with a predefined
6432 -- operator.
6433 -- For predefined operators, the signature of the operator does not
6434 -- fix the type of the arguments, because it only specifies a class
6435 -- of types. The precise type to be used is either imposed by context
6436 -- (this is the argument ctx_type) or is found by requiring consistency
6437 -- between the possible types of the arguments themselves.
6438 const    unary_sig  = [['RIGHT', 'in', 'any', [ ]] ] ,
6439          binary_sig = [['LEFT',  'in', 'any', [ ]],
6440                        ['RIGHT', 'in', 'any', [ ]] ] ;
6441
6442
6443
6444
6445 if #arg_list = 2 then              -- Binary operators.
6446
6447   arg_list := order_arg_list(arg_list, binary_sig) ;
6448
6449   [arg1, arg2] := arg_list ;
6450   t_left := arg1(2) ;                    -- possible types of arg1.
6451   t_right:= arg2(2) ;
6452
6453   case type_of(op_name) of
6454
6455        ('universal_integer', 'universal_real', 'universal_fixed',
6456                                                  '$FIXED') :
6457
6458       ctx_root := root_type(ctx_type) ;
6459
6460       if ctx_type = 'universal_fixed' then
6461           -- Must have appeared in a conversion. Replace the universal
6462           -- fixed type with the generic FIXED, to resolve each arg.
6463           ctx_type := '$FIXED' ;
6464       end if ;
```

```
6465
6466            case op_name of
6467
6468                ('*fxi', '*ifx', '/fxi','**i', '**fl'):
6469                -- For mixed mode fixed operations and exponentiation,
6470                -- the type from context is imposed on the first
6471                -- argument. The second one must be INTEGER.
6472
6473                if op_name = '*ifx' then          -- permute arguments.
6474                    [arg1, arg2] := [arg2, arg1] ;
6475                    t_right := t_left ;
6476                    op_name := '*fxi' ;
6477                end if ;
6478
6479                if op_name = '**fl' and ctx_root = '$FIXED' then
6480                    -- universal expression in fixed context: no ** .
6481                    type_error({'**'}, om, 0) ;
6482                    return [ ] ;
6483                end if ;
6484
6485                e1 := resolve2(arg1, ctx_type);
6486                e2 := resolve2(arg2, 'INTEGER') ;
6487
6488                ('*fli', '*ifl', '/fli'):
6489
6490                -- These mixed mode operations appear in number declara-
6491                -- tions, in which case they are universal, or in a fixed
6492                -- type context.
6493
6494                if op_name = '*ifl' then      -- permute arguments.
6495                    [arg1, arg2] := [arg2, arg1] ;
6496                    t_right := t_left ;
6497                    op_name := '*fli' ;
6498                end if ;
6499
6500                if ctx_root = 'universal_real' then
6501                    t2 := 'universal_integer' ;
6502
6503                elseif ctx_root = '$FIXED' then
6504                    -- universal expression in fixed context.
6505                    t2 := 'INTEGER' ;
6506                else
6507                    errmsg('Invalid context ',
6508                            'for mixed mode operation','4.5.5, 4.10') ;
6509                    return [ ] ;
6510                end if ;
6511
6512                e1 := resolve2(arg1, ctx_type);
6513                e2 := resolve2(arg2, t2) ;
```

```
6514              ('*fl', '/fl'):
6515
6516              -- These floating point operation may appear in some
6517              -- fixed type context if their constituents are
6518              -- literals. This is an error because the operation
6519              -- yields in fact a universal_fixed quantity that
6520              -- should be converted.
6521              e1 := resolve2(arg1, ctx_type) ;
6522              e2 := resolve2(arg2, ctx_type) ;
6523
6524              if ctx_root = '$FIXED' then
6525                 e := literal_expression([op_name, e1, e2]) ;
6526                 if ctx_type /= '$FIXED' then
6527                      -- Some specific fixed type: the conversion
6528                      -- from universal fixed was missing.
6529                    errmsg('Missing explicit conversion from',
6530                           ' universal_fixed value ','4.5.5') ;
6531                      return [ ] ;
6532                 else
6533                      -- The presence of the generic type -FIXED
6534                      -- indicates that a conversion was present.
6535                      return e;
6536                 end if ;
6537              end if ;
6538
6539           else
6540              -- For other arithmetic operators, propagate context
6541              -- type to arguments. This is in accordance with final
6542              -- Ada, and the absence of literal expressions embedded
6543              -- in other expressions.
6544
6545              e1 := resolve2(arg1, ctx_type) ;
6546              e2 := resolve2(arg2, ctx_type) ;
6547           end case ;
6548
6549           -- If the context is universal, evaluate the corresponding
6550           -- literal expression.
6551
6552 --
6553           if ctx_type ∈ univ_types or
6554             (ctx_root = '$FIXED' and is_value(e1) and is_value(e2))
6555        then
6556              return literal_expression([op_name, e1,e2]) ;
6557           else
6558              return [op_name, e1, e2] ;
6559           end if ;
6560
6561        ('order_type', 'discrete_type', 'BOOLEAN') :
6562
```

```
6563        -- Equality ,set or comparison  operators. Verify that  there  is
6564        -- only one possible type choice for both arguments. If both arg.
6565      -- are universal, we must choose a universal interpretation for
6566      -- each. Otherwise, the non-universal type is applied to both.
6567       ok_types := {base_type(i) : t1 ∈ t_left, t2 ∈ t_right |
6568                                   (i := intersect_types(t1,t2)) /= om} ;
6569      if # ok_types =  1 then
6570          t1 := t2 := arb ok_types ;
6571
6572      elseif #(univ := ok_types * univ_types) = 1 then
6573          t1 := t2 := arb univ ;
6574
6575       else
6576          type_error({op_name}, tf, #ok_types) ;
6577          return [ ] ;
6578      end if ;
6579
6580       if op_name  ['in', 'notin'] then
6581        if is_limited_type(t1) then
6582           errmsg(op_name,' not available on a limited type',
6583                                                '7.4.2');
6584           return [ ] ;
6585        end if ;
6586
6587          -- Now resolve each operand independently.
6588
6589          e1 := resolve2(arg1, t1) ;
6590          e2 := resolve2(arg2, t2) ;
6591      else
6592          -- For memberships, the second argument  may be a type mark
6593          -- for which an anonymous type was created. The test itself
6594          -- carries the subtype expression of this type.
6595
6596          e1 := resolve2(arg1, t1) ;
6597          if not is_identifier(name2 := arg2(1))
6598          then
6599             e2 := resolve2(arg2, t2) ;
6600          else
6601             e2 := subtype_expr(name2) ;
6602          end if ;
6603
6604          -- The membership tests are not static, but their arguments
6605          -- may be universal. Convert them to non-universal form for
6606          -- run-time evaluation.
6607          if t2 = 'universal_integer' then
6608              e1 := specialize(e1, 'INTEGER') ;
6609          elseif t2 = 'universal_real' then
6610              e1 := specialize(e1, 'FLOAT') ;
6611          end if ;
```

```
6612
6613        end if ;
6614
6615        -- Comparison operators on  literal expressions are evaluated
6616        -- separately,  because both their arguments are universal.
6617
6618        if op_name ∈ comparison_ops and t1 = t2
6619            and t1 ∈  univ_types then
6620            return literal_expression([op_name, e1, e2]) ;
6621         end if ;
6622
6623         ('array_type'):                    -- Concatenation operator.
6624            -- If an array is being concatenated with an object of
6625            -- the component type, then we row the component.
6626
6627            if is_array(arb t_left) then
6628                e1 := resolve2(arg1, ctx_type) ;
6629            else
6630                e1 := ['row', resolve2(arg1, component_type(ctx_type)),
6631                                                          ctx_type] ;
6632            end if ;
6633
6634            if is_array(arb t_right) then
6635                e2 := resolve2(arg2, ctx_type) ;
6636            else
6637                e2 := ['row', resolve2(arg2, component_type(ctx_type)),
6638                                                        ctx_type] ;
6639            end if ;
6640    else
6641         -- Other binary operators.
6642         e1 := resolve2(arg1, ctx_type) ;
6643        e2 := resolve2(arg2, ctx_type) ;
6644
6645    end case ;
6646
6647    return [op_name, e1, e2] ;
6648 else
6649
6650    arg_list := order_arg_list(arg_list, unary_sig) ;
6651
6652    -- Unary operator. Type is always imposed by context.
6653    arg1 := arg_list(1) ;
6654    e1 := [op_name, resolve2(arg1, ctx_type)] ;
6655    if ctx_type ∈ univ_types then
6656        return literal_expression(e1) ;
6657    else
6658        return e1 ;
6659    end if ;
6660
```

```
6661 end if ;
6662
6663
6664 end proc complete_op_expr;
6665
```

specialize

```
6667 proc specialize(u_expr, ctx_type) ;
6668
6669
6670 -- Convert a universal numeric into a specific one, if the context impo-
6671 -- ses a non-universal numeric type.
6672
6673 if not is_value(u_expr) or not is_numeric(ctx_type) then
6674     return u_expr;
6675 end ;
6676
6677 [-,v, t] := u_expr ;
6678 if ctx_type   univ_types then
6679     if is_universal_integer(v) then
6680         v := int_toi(v) ;        -- Check bounds.
6681       t := 'INTEGER' ;
6682     elseif is_universal_real(v) and root_type(ctx_type) /= '$FIXED'
6683     then
6684         v := rat_tor(v, ADA_REAL_DIGITS) ;
6685         t := 'FLOAT' ;
6686     end if ;
6687 end if ;
6688
6689 return if v = 'OVERFLOW' then constraint_error()
6690     else ['ivalue', v, t]
6691 end ;
6692
6693 end proc specialize ;
```

literal_expression

```
6695 proc literal_expression(expn) ;
6696
6697 -- Use the arbitrary precision arithmetic package to evaluate an arith-
6698 -- metic expression whose arguments are literal. This routine is called
6699 -- in contexts that require a universal value, i.e. constant definitions.
6700 -- If the constituents are not universal, the expression is returned as
6701 -- is.
6702 -- Several attributes deliver a universal value, but are nevertheless
6703 -- evaluated at run-time. If these attributes are companion operands of
6704 -- literals, then these literals must be converted to non-universal form.
6705 -- It is an open question to which integer type they should converted.
```

```
6706 -- For the time being we choose INTEGER.
6707
6708 macro ada_bool(x) ;
6709         (if x then 1 else 0 end)
6710 endm;
6711
6712
6713
6714 if #expn = 3 then              -- binary operation.
6715    [op_name, e1, e2 ] := expn ;
6716
6717    if is_value(e1) then
6718       op1 := e1(2) ;
6719       if is_value(e2) then
6720          op2 := e2(2) ;
6721
6722          -- In the case of mixed mode operations on fixed types, the
6723          -- second argument is already folded to INTEGER. If a static
6724          -- evaluation is possible, make it into a universal object
6725          -- again.
6726          --- if is_integer(op2) and
6727          --- (is_universal_real(op1) or op_name = '**i')
6728          --- then
6729          ---   op2 := int_fri(op2) ;
6730          --- end if ;
6731       else
6732          -- op2 is attribute expr. If first operand is integer, check
6733          -- its bounds . If it is a mixed operation, convert the first
6734          -- operand to the most precise floating type available.
6735
6736          e1 := specialize(e1, if is_integer(e1) then 'INTEGER'
6737                                                 else 'FLOAT' end) ;
6738          return [op_name, e1, e2] ;
6739       end if ;
6740
6741    else                                  -- op1 is attribute expr.
6742       if is_value(e2) then
6743          e2 := specialize(e2, 'INTEGER') ;
6744          return [op_name, e1, e2] ;
6745       else                                       -- They both are.
6746          return expn ;
6747       end if ;
6748    end if ;
6749
6750 else
6751    [op_name, e1 ] := expn ;
6752    if not is_value(e1) then
6753       return expn ;
6754    else
```

```
6755        op1 := e1(2) ;
6756     end if;
6757 end ;
6758
6759 is_int := is_universal_integer(op1) ;
6760  if not is_int and not is_universal_real(op1)
6761  or
6762   #expn = 3 and not is_universal_integer(op2) and
6763                    not is_universal_real(op2)
6764 then
6765     return expn ;
6766 end if ;
6767
6768 ivalue := case op_name of
6769
6770     ('+i'):        (op1 + op2) ,                          --
6771
6772     ('+fl','+fx'): rat_add(op1, op2) ,
6773
6774     ('-i'):          (op1 - op2) ,                        --
6775
6776     ('-fl','-fx'): rat_sub(op1, op2) ,
6777
6778     ('*i'):          (op1 * op2) ,                              --
6779
6780     ('*fl', '*fx'): rat_mul(op1, op2) ,
6781
6782     ('*fxi','*fli'):
6783                      rat_red((op1(1) * op2), op1(2)) ,          --
6784
6785     ('/fxi','/fli'):
6786                      rat_red(op1(1), (op1(2) * op2)) ,          --
6787
6788     ('/i'):        (op1 div op2) ,                          --
6789
6790     ('/fl','/fx'): rat_div(op1, op2) ,
6791
6792     ('remi'):      if op2 = 0 then   expr return constraint_error(); end
6793                    else
6794                      (op1 - (op1 div op2) * op2)
6795                    end,
6796
6797     ('modi'):      if op2 = 0 then   expr return constraint_error(); end
6798                    else
6799                         (if (md :=op1 mod op2) = 0 or op2 > 0 then md
6800                               else md + op2 end)            --
6801                    end,
6802
6803     ('**i'):       if (op2 < 0) then
```

```
6804                    expr
6805                        return constraint_error();
6806                                                    --
6807                    end
6808                    else
6809                        (op1 ** op2)
6810                    end ,
6811
6812    ('**fl'):     rat_exp(op1, op2) ,
6813
6814    ('=')  :      ada_bool(if is_int then (op1 = op2)
6815                                  else rat_eql(op1, op2) end),
6816
6817    ('/=') :      ada_bool(if is_int then (op1 /= op2)
6818                                  else rat_neq(op1, op2) end),
6819
6820    ('>')  :      ada_bool(if is_int then (op1 > op2)
6821                                  else rat_gtr(op1, op2) end),
6822
6823    ('<')  :      ada_bool(if is_int then (op1 < op2)
6824                                  else rat_lss(op1, op2) end),
6825
6826    ('≥') :         ada_bool(if is_int then (op1 ≥ op2)
6827                                  else rat_geq(op1, op2) end),
6828
6829    ('≤') :      ada_bool(if is_int then (op1 ≤ op2)
6830                                  else rat_leq(op1, op2) end),
6831
6832    ('+ui','+ufl', '+ufx'): op1 ,
6833
6834    ('-ui'):                  -(op1),
6835
6836    ('-ufl','-ufx'):        rat_umin(op1),
6837
6838    ('absi'):                 abs(op1),
6839
6840    ('absfl', 'absfx'):     rat_abs(op1)
6841
6842    else
6843        expr
6844            return eval_static(expn);
6845        end
6846
6847 end ;
6848
6849 return ['ivalue', ivalue] ;
6850
6851 drop ada_bool;
6852
```

```
6853 end proc literal_expression ;
6854
```

bind_arg

```
6856 proc bind_arg(actual, f_type, f_mode) ;
6857
6858
6859
6860 -- In order to indicate that constraints must be applied to the actual
6861 -- parameters of a procedure or entry, each actual that may have to be
6862 -- checked (either on entry or exit) is formatted as follows :
6863 --
6864 --        ['convert', actual_type, formal_type, actual]
6865 --
6866 -- The interpreter transforms this into a constraint check on entry
6867 -- (for IN parameters) on exit (for OUT prameters) or both. This notation
6868 -- also indicates cases when an explicit conversion has been specified,
6869 -- which may have to be applied on exit to an IN-OUT parameter.
6870 -- This formatting is done for scalar types and for access types. Arrays
6871 -- and records are handled differently, because they are passed by refe-
6872 -- rence, and they carry the explicit constraints that must be applied.
6873
6874 [-, a_types] := actual ;                    -- Get possible types of actual.
6875
6876 -- One of its possible types must be compatible with the formal.
6877 assert exists a_type ∈ a_types
6878                         | compatible_types(f_type,a_type) ;    --
6879
6880 -- An out parameter may appear as the actual for another out parameter.
6881 out_c := out_context ;
6882 out_context := (f_mode = 'out') ;
6883 ---ES 154 (
6884 -- If the actual is an aggregate, there is no sliding for it, and named
6885 -- associations can appear with "others" (cf. 4.3.2(6))
6886 may_others := full_others ;
6887 full_others := true ;
6888
6889 if is_scalar_type(f_type) then
6890
6891     arg := resolve2(actual, f_type) ;
6892     -- Convert from universal value if need be.
6893     arg := specialize(arg, f_type) ;
6894
6895     if is_tuple(arg) and arg(1) = 'qual_range' then
6896         [-,-, a_arg] := arg ;
6897         if is_tuple(a_arg) and a_arg(1) = 'convert' then
6898             -- It may have been a conversion to the base type of the formal
6899             -- type. Rather than impose the range check on the result of
```

```
6900            -- the conversion, modify  the target type of the conversion to
6901            -- be the constrained type.
6902
6903            a_arg(3) := f_type ;
6904            arg := a_arg ;
6905         else
6906            -- change the constraint check into a convert.
6907            arg := ['convert', a_type, f_type, a_arg] ;
6908         end if ;
6909
6910      -- No constraint check was needed on entry.It may still be required
6911      -- for OUT or INOUT parameters.
6912      elseif f_mode /= 'in' and not can_constrain(f_type)
6913      then
6914           arg := ['convert', a_type, f_type, arg] ;
6915      else
6916         -- formal and actual are two-way compatible.
6917         pass ;
6918      end if ;
6919
6920   elseif is_access(f_type) then
6921      arg := resolve2(actual, f_type) ;
6922
6923      -- If the  formal is constrained, then  place a constraint check  on
6924      -- the  actual. In  the  case  of an  unconstrained  formal and of a
6925      -- constrained actual (out  or in-out)  use the  'convert' marker to
6926      -- indicate that a check must be emitted after the call.
6927      u_f := can_constrain(designated_type(f_type)) ;
6928      u_a := can_constrain(designated_type(a_type)) ;
6929      case f_mode of
6930           ('in'):
6931             if not u_f and f_type /= a_type then
6932               -- No sliding semantics here: array bounds must match.
6933               if (check := constraint_kind(f_type)) = 'qual_alength' then
6934                    check := 'qual_aindex' ;
6935               end if ;
6936               arg := [check, designated_type(f_type), arg] ;
6937           end if ;
6938
6939           ('out'):
6940           if not u_a and f_type /= a_type then
6941                arg := ['convert', a_type, f_type, arg] ;
6942           end if ;
6943
6944           ('inout'):
6945             -- If one of the types is unconstrained, the corresponding test
6946             -- must be omitted. This is  indicated by using the constrained
6947             -- type  both in and out. If  they are both constrained, a test
6948             -- on entry will also suffice.
```

```
6949
6950            if not u_f and f_type /= a_type then
6951              arg := ['convert', f_type, f_type, arg] ;
6952
6953            elseif not u_a and u_f then
6954              arg := ['convert', a_type, a_type, arg] ;
6955            end if ;
6956
6957
6958      end case;
6959
6960 else                                          -- array or record type.
6961      arg := resolve2(actual, f_type) ;
6962 end if ;
6963
6964 out_context := out_c ;                        --
6965 full_others := may_others ;
6966 return arg ;
6967
6968 end proc bind_arg ;
6969
6970
6971
```

valid_array_expn

```
6973 proc valid_array_expn(a_expn, i_list) ;
6974
6975
6976
6977 -- Called to validate indexing and slicing operations. The array name may
6978 -- be overloaded, and may be an access to an array type.
6979
6980 [arrays, array_types] := resolve1(a_expn) ;
6981 -- In the case of a slice, the index list may contain a type name.
6982 index_list := [
6983   (if is_identifier(i) then [i, {type_of(i)}] else resolve1(i) end )
6984                                                  : i ∈ i_list] ;
6985
6986 -- Now select those array types that are compatible with the given indices.
6987 comp_types := {} ;                    -- For component types of array.
6988 (forall a_t ∈ array_types)
6989     t := a_t ;
6990     if is_access(t) then
6991       if is_fully_private(t) then                 --
6992         -- Cannot dereference an access to fully private type.
6993           if #array_types = 1 then return premature_access(t);
6994           else continue forall a_t ;
6995           end if ;
```

```
6996
6997        else
6998            t := designated_type(t) ;
6999        end if ;
7000    end if ;
7001    if not is_array(t) or no_dimensions(t) /= #index_list then
7002        array_types less:= a_t ;
7003        continue forall a_t ;
7004    end if ;
7005
7006    -- Now verify all indices in turn.
7007    (forall [-, i_types] = index_list(i))
7008        if not exists i_t ∈ i_types |
7009                compatible_types(i_t, index_types(t)(i)) then --
7010            array_types less:= a_t ;            -- Not a candidate.
7011            -- Examine remaining candidates.
7012            continue forall a_t ;
7013        end if ;
7014    end forall ;
7015    -- Indices match, so the resulting type is the component type.
7016    comp_types with:= component_type(t) ;
7017 end forall a_t ;
7018
7019 return if comp_types = {} then [[ ],{}]
7020 else
7021    [ ['[ ]', [arrays, array_types], index_list], comp_types]
7022 end ;
7023
7024 end proc valid_array_expn;
```

complete_array_expn

```
7026 proc complete_array_expn(a_expn, i_num, c_type) ;
7027
7028
7029
7030 -- Called to complete the validation of an index or slice expression.
7031 -- i_num is the number of indices in the expression.
7032 -- For a slice, the  i_num is zero, and the context type is the
7033 -- required type of the array. For an indexing expression, the context
7034 -- type is the component type of the array.
7035 -- The array expression may yield an access type, in which case a derefe-
7036 -- rence operation is emitted now.
7037
7038 [arrays, array_types] := a_expn ;
7039 array_type := om ;
7040
7041 -- Iterate over array types to find unique one satisfying context.
7042
```

```
7043 (forall a_t ∈ array_types)
7044     t := if is_access(a_t) then designated_type(a_t) else a_t end ;
7045     c := if i_num = 0 then t else component_type(t) end ;
7046
7047     if compatible_types(c_type, c) then                                    --
7048         if array_type = om then                    -- One match found.
7049             array_type := t ;
7050             access_type := a_t ;          -- Maybe an access.
7051         else
7052             -- If it is ambiguous, then it must an overloaded function
7053             -- that return (an access to) an array.
7054
7055             type_error(arrays(1), c_type, 2) ;
7056             return [ ] ;
7057         end if;
7058     end if ;
7059
7060 end forall ;
7061 if array_type = om then
7062     type_error({'indexing'}, c_type, 0) ;
7063     return [ ] ;
7064 end if ;
7065 --
7066 if array_type /= access_type then              -- Insert dereference.
7067     out_c := out_context ;
7068     out_context := false ;                     -- cannot be out parameter
7069     array := ['@', resolve2(a_expn, access_type), array_type] ;
7070     out_context := out_c ;
7071 else
7072     array := resolve2(a_expn, array_type) ;
7073 end if ;
7074 --
7075
7076 return [array, array_type] ;
7077
7078 end proc complete_array_expn ;
7079 --
```

valid_selected_expn

```
7081 proc valid_selected_expn(obj, selector) ;
7082
7083 -- Use the name of the selector to determine the possible types of obj,
7084 -- which may be a function returning (an access to) a record or task type
7085 -- The possible types of the expression are those of the selected comps.
7086
7087 obj1 := resolve1(obj) ;
7088 types1 := obj1(2) ;
7089 valid_t := {} ;
```

```
7090
7091 (for o_t ∈ types1)
7092     t := o_t ;
7093     if is_access(o_t) then t := designated_type(o_t) ; end if ;
7094     if is_record(t) then  decls := declared_components(base_type(t)) ;
7095
7096     elseif is_task_type(t) then  decls := declared(t) ;
7097
7098     else  continue for ;
7099     end if ;
7100
7101     if (comp := decls(selector)) /= om then
7102         if is_access(o_t) and is_fully_private(o_t)
7103             and nature(comp) /= 'discriminant' then  -- Can't dereference.
7104              if #types1 = 1 then return premature_access(o_t) ;
7105              else continue for ;
7106              end if ;
7107         else
7108              valid_t with:= type_of(comp) ;
7109         end if ;
7110
7111     end if ;
7112 end for ;
7113
7114 if valid_t = {} then
7115     return pass1_error('invalid selector name: ', selector, '4.1.3') ;
7116 else
7117     return [ ['.', obj1, selector], valid_t] ;
7118 end if ;
7119
7120 end proc valid_selected_expn ;
```

complete_selected_expn

```
7122 proc complete_selected_expn(obj1, selector, c_type) ;
7123
7124 -- Complete the  resolution of  a selected component  expression, by
7125 -- choosing the one that yields the context_type. If the type of the
7126 -- object selected from is an access type, emit a dereference.
7127
7128 [expn, types1] := obj1 ;
7129 comp_t := om ;
7130
7131 (for o_t ∈ types1)
7132
7133     t := if is_access(o_t) then designated_type(o_t) else o_t end ;
7134
7135     if is_record(t) then
7136          decls := declared_components(base_type(t)) ;
```

```
7137    elseif is_task_type(t) then
7138        decls := declared(t) ;
7139    end if ;
7140
7141    c := decls(selector) ;
7142    if compatible_types(type_of(c), c_type) then              --
7143        comp := c ;
7144        if comp_t = om then
7145            comp_t := type_of(comp) ;              -- Found a match
7146            obj_t  := o_t ;
7147        else                                   -- ambiguous call to some
7148            type_error(expn(1), t, 2) ;        -- overloaded function.
7149            return [ ] ;
7150       end if ;
7151    end if ;
7152
7153 end for ;
7154
7155 if comp_t = om then
7156    type_error({'selection'}, om, 0) ;
7157    return [ ] ;
7158 else
7159 --
7160    out_c := out_context ;
7161    -- This is a valid context for the use of an out parameter, if
7162    -- it is an assigment to a component of it, or if it is a reading
7163    -- of a discriminant, and in any case if the type is not an access.
7164
7165    out_context := (out_c or nature(comp) = 'discriminant' )
7166                                        and not is_access(obj_t) ;
7167    obj := resolve2(obj1, obj_t) ;
7168    out_context := out_c ;
7169 --
7170
7171    if is_access(obj_t) then
7172        obj := ['@', obj, designated_type(obj_t)] ;
7173    end if ;
7174
7175    return [['.', obj, comp], comp_t] ;
7176 end if ;
7177
7178 end proc complete_selected_expn;
```

is_component

```
7180 proc is_component(expn) ;
7181
7182 -- Used to validate uses of the attributes FIRST_BIT and LAST_BIT.
7183 -- An expression denotes a component if it is a qualified expression, or
```

```
7184 -- it is an identifier that renames such an expression.
7185 return
7186     is_tuple(expn) and expn(1) = '.' and is_record(scope_of(expn(3)))
7187 or  is_identifier(expn) and is_component(signature(expn)) ;
7188 end proc is_component; --
7189 --
```

complete_aggregate

```
7191 proc complete_aggregate(agg_type, arg_list) ;
7192
7193 -- Given the context type, resolve the aggregate components. For an array
7194 -- type we pass index and component types separately to the recursive
7195 -- routine complete_a_aggregate. For record types only the base type is
7196 -- needed here. Any required constraints are imposed in resolve2.
7197
7198 if is_limited_type(agg_type) then
7199     errmsg('aggregates not available for limited type',
7200                                     original_name(agg_type), '7.4.4') ;
7201     return [ ] ;
7202 end if ;
7203 if is_array(agg_type) then
7204     return  complete_a_aggregate(index_types(agg_type),
7205           component_type(agg_type), can_constrain(agg_type), arg_list);
7206
7207 elseif is_record(agg_type) then
7208     return  complete_r_aggregate(base_type(agg_type), arg_list) ;
7209
7210 else
7211     errmsg('Invalid context for aggregate','','none') ;
7212     return [ ] ;
7213 end ;
7214
7215 end proc complete_aggregate ;
7216
```

complete_a_aggregate

```
7218 proc complete_a_aggregate(indices, comp_type, is_unc, arg_list) ;
7219
7220 -- Complete processing of an array aggregate.
7221
7222     indxt := indices(1) ;
7223     other_choice := false ;
7224     pos_list := [ ] ;
7225     nam_list := [ ] ;
7226
7227     (forall arg ∈ arg_list)
7228         if other_choice then
```

```
7229        errmsg('OTHERS must be the last aggregate component','','4.3');
7230        return [ ] ;
7231       end if ;
7232       if is_tuple(arg) and arg(1) = 'choice_list' then
7233          [-, choice_list, c_expr] := arg ;
7234          choices := [ ] ;
7235          c_expr2 := complete_component(indices(2..),
7236                                        comp_type, is_unc, c_expr) ;
7237
7238          (for [choice_type, i_expr] ∈ choice_list)
7239
7240           -- If the index expression is an identifier, it has not
7241           -- been resolved yet, in order to avoid problems with
7242           -- hidden record selector names. For an array, this id.
7243           -- must be a visible object name. Resolve it now.
7244
7245           [ix, i_types] := i_expr ;
7246           if is_identifier(ix) and choice_type = 'simple_choice'
7247           then
7248                i_expr := find_old(ix) ;
7249                if is_type(i_expr) then
7250                   -- subtype name, indicating a range choice.
7251                      choice_type := 'range_choice' ;
7252                      i_expr := [i_expr, {i_expr}] ;
7253                else
7254                      i_expr := resolve1(i_expr) ;
7255                end if ;
7256           end if ;
7257           case choice_type of
7258
7259           ('simple_choice'):
7260                choices with := resolve2(i_expr, indxt) ;
7261
7262           ('range_choice'):
7263              r_c := resolve2(i_expr, indxt) ;
7264              if is_identifier(r_c) then  -- range is subtype name
7265                 choices with:= ['tname', r_c] ;
7266              elseif r_c = [ ] then       -- type error in bounds.
7267                 return [ ] ;
7268             elseif r_c(1) = '"' then   -- range attribute
7269                 [-,-, i] := r_c ;
7270                 if is_type(i) then
7271                    choices with:= ['tname', i] ;
7272                 else
7273                    choices with:= ['tname', r_c] ;
7274                 end if ;
7275              else                          -- subtype expr.: unpack
7276                 [-,type_mark, constraint] := r_c ;
7277                 choices with:= constraint ;
```

```
7278                end if ;
7279
7280             ('others_choice') :
7281               if is_unc then
7282                 errmsg('OTHERS choice not allowed in this context',
7283                                          '','4.3, 4.3.2');
7284                   return [ ] ;
7285               else
7286                    choices := 'others' ;
7287                    other_choice := true ;
7288               end if;
7289
7290             end case ;
7291          end for ;
7292
7293
7294          nam_list with:= [choices, c_expr2] ;
7295        else                                -- Positional component.
7296          pos_list with:= complete_component(indices(2..),
7297                                             comp_type, is_unc, arg) ;
7298        end if ;
7299    end forall ;
7300
7301 if other_choice and #nam_list > 1 and not full_others then
7302    errmsg('Invalid context for "others" and named associations',
7303                                                    '', '4.3.2(6)') ;
7304    return [ ] ;
7305 elseif pos_list /= [ ] and nam_list /= [ ]
7306   and (not other_choice or #nam_list > 1) then
7307    errmsg('apart from "others", associations must be ',
7308                    'all positional or all named', '4.3.2(3)') ;
7309    return [ ] ;
7310 end if ;
7311
7312 return ['array_aggregate', indices, pos_list, nam_list] ;       --
7313
7314 end proc complete_a_aggregate ;
```

complete_component

```
7316 proc complete_component(indices, comp_type, is_unc, expn) ;
7317
7318 -- Complete the resolution of a component of an array aggregate. If it
7319 -- is a multidimensional aggregate, the component itself is an array and
7320 -- a recursive call is made with the remaining indices. The aggregate
7321 -- types attached to it can be discarded, because the type of the comp-
7322 -- onent is anonymous. String literals are handled in their own routine.
7323 -- There is no sliding for sub-aggregates: their bounds must belong to
7324 -- the corresponding subtypes.
```

```
7325
7326 if indices = [ ] then return res2_check(expn, comp_type) ;
7327 else
7328    [e, types] := expn ;
7329
7330    if not is_tuple(e) or (kind := e(1))
7331                  ['aggregate', 'string_literal', '()'] then
7332       errmsg('Invalid expression for',
7333                ' component of multidimensional aggregate', '4.3.2') ;
7334        c := [ ] ;
7335
7336    elseif kind = 'aggregate' then
7337       -- There is no sliding on constructing sub-aggregates.
7338       c := eval_static(complete_a_aggregate(indices, comp_type,
7339                                                    is_unc, e(2)));
7340
7341    elseif kind = '()' then
7342        -- Context is unconstrained: no others choice is allowed.
7343        c := complete_component(indices, comp_type, true, e(2)) ;
7344
7345    else
7346        if #indices /= 1 then
7347           errmsg('Invalid use of literal in aggregate','','4.3.2') ;
7348        end if ;
7349        c := complete_string_literal(e(2), indices, comp_type) ;
7350    end if ;
7351
7352    return c ;
7353 end ;
7354
7355 end proc complete_component;
```

complete_string_literal

```
7357 proc complete_string_literal(strg, indices, comp) ;
7358
7359 -- String literals are transformed into array aggregates.
7360 -- This simplifies the application of sliding semantics. The context may
7361 -- be an array of some enumeration type, in which case we have to verify
7362 -- that each character in the string is an  enumeration literal for that
7363 -- type. For now we assume no representation specification was given for
7364 -- characters, and the ascii values are used.
7365
7366 sc := scope_of(comp) ;
7367 if sc   open_scopes and sc   used_mods then
7368    errmsg('characters in a string literal must be directly visible',
7369                                                 '','4.2(3)') ;
7370 end if ;
7371
```

```
7372 if comp ∈ {'CHARACTER','any'} then
7373 e := ['array_aggregate', indices, [ ['ivalue', abs c] : c ∈ strg],[ ]] ;
7374
7375 else                    -- Some enumeration type. Use its literal map.
7376
7377    if nature(base_type(comp)) /= 'enum' then
7378       errmsg('component type of context is not a character type ',
7379                                                          '','4.2') ;
7380       return [ ] ;
7381    end if ;
7382    lit_map := literal_map(base_type(comp)) ? {} ;
7383    arr := [lit_map('"' + c + '"') : c ∈ strg] ;
7384
7385    if exists c = strg(i) | arr(i) = om then
7386       -- Some characters are not in the component type.
7387       e := constraint_error() ;                          --
7388    else
7389       e := ['array_aggregate',indices, [['ivalue', v ]: v ∈ arr],[ ]];
7390
7391       -- The individual characters must be bounds-checked as any other
7392       -- array component.
7393       if nature(comp) = 'subtype' then
7394          [-,lo, hi] := signature(comp) ;
7395          if is_value(lo) and is_value(hi) and
7396            exists v ∈ arr | v < lo(2) or v > hi(2) then
7397               e := constraint_error() ;
7398          end if ;
7399       end if ;
7400
7401    end if ;
7402 end if ;
7403
7404 return e ;
7405
7406 end proc complete_string_literal;
7407 --
```

complete_r_aggregate

```
7409 proc complete_r_aggregate(aggregate_type, arg_list) ;
7410
7411 -- Complete resolution of components in a record aggregate. If the
7412 -- record type has discriminants, we must first complete resolution
7413 -- of the discriminant components, in order to determine the variant
7414 -- parts to which the rest of the aggregate must conform.
7415
7416
7417
7418
```

```
7419    discr_list := discriminant_list(aggregate_type) ;
7420
7421    -- Components can be given by named choices. Divide  argument list
7422    -- into positional and named components .
7423
7424    if exists arg=arg_list(i) | is_tuple(arg) and arg(1)='choice_list'
7425    then
7426        first_named := i ;
7427    else
7428        first_named := #arg_list + 1 ;
7429    end if ;
7430
7431    positional_args := arg_list(1..first_named-1) ;
7432    named_args := arg_list(first_named..) ;
7433    maybe := exists [-,choice_list,c_expr] ∈ named_args |
7434      #choice_list = 1 and choice_list(1)(1) = 'others_choice' ; --
7435
7436    others_expr := if maybe then c_expr else om end ;
7437
7438    discr_map := {} ;
7439
7440    -- Map the discriminants into the (hopefully) static expressions
7441    -- given for them.
7442
7443    i1 := #discr_list min #positional_args ;
7444
7445    (for i ∈ [1..i1])
7446
7447        discr_map(discr_list(i)) := positional_args(i) ;
7448    end ;
7449
7450    -- Now look for named discriminants among named components.
7451
7452    (for discr ∈ discr_list(i1+1..))
7453
7454        if exists [-, choice_list, c_expr] ∈ named_args |
7455            exists [-,[id,-]] ∈ choice_list | id = original_name(discr)
7456          then
7457            discr_map(discr) := c_expr ;
7458          elseif others_expr /= om then
7459            discr_map(discr) := others_expr ;
7460        else
7461            errmsg('No value supplied for discriminant ',
7462                                    original_name(discr),'4.3.1') ;
7463            return [ ] ;
7464          end if ;
7465    end for ;
7466
7467    -- Now complete type resolution of the discriminant values, and
```

```
7468    -- verify that they are static.
7469
7470    (for expn = discr_map(discr))
7471
7472       expn2 := resolve2(expn, type_of(discr)) ;
7473       discr_map(discr) := const_val(expn2) ;
7474    end for ;
7475
7476    components := invariant_part(aggregate_type) ;  -- To start with.
7477    variant := variant_part(aggregate_type) ;
7478    sel_names := declared_components(aggregate_type) ;
7479
7480    -- Now scan the variant part of the record declaration, and collect
7481    -- the types corresponding to the given discriminants.
7482
7483    (while variant /= [ ])
7484
7485       [discr, variants ] := variant ;
7486       v := discr_map(discr) ;
7487       if not is_value(v) then
7488          errmsg('Value for discriminant ' + original_name(discr),
7489                          'must be static','4.3.1') ;
7490          return [ ] ;
7491       else
7492          discr_value := v(2) ;
7493       end if ;
7494       if (exists [alt_list, decls] ∈ variants, alt ∈ alt_list |
7495             ((is_tuple(alt) and
7496             alt(2) ≤ discr_value and discr_value ≤ alt(3))
7497          or
7498             alt = discr_value))
7499       or (exists [alt_list, decls] ∈ variants |
7500             alt_list = {'others'} )                          --
7501       then
7502          if decls /= ['null'] then
7503             components +:= decls(1) ;
7504          end if;
7505       else
7506          errmsg('Discriminant value for ' + original_name(discr),
7507                          ' is out of range','3.7.2') ;
7508          return [ ] ;
7509       end if ;
7510       variant := decls(2) ;          -- Variants may be nested.
7511
7512    end while ;
7513
7514    -- Now the list of components is assembled.
7515    -- Complete the resolution of all components.
7516
```

```
7517    comp_list := [ ] ;          -- To gather list of selector assignments.
7518    comp_pos  := 1 ;            -- Index into component list.
7519
7520    -- Each component declaration declares a list of selectors with the
7521    -- same type and initial value. Flatten those lists into a single
7522    -- one.
7523    all_component_names := [ ] +/[obj(2) : obj ∈ components] ;
7524    components_seen := {} ;
7525
7526    (forall arg ∈ arg_list)
7527        if is_tuple(arg) and arg(1) = 'choice_list' then
7528           [-, choice_list, c_expr] := arg ;
7529           choices := [ ] ;
7530           (for [choice_type, [sel,-]] ∈ choice_list)
7531
7532           case choice_type of
7533
7534           ('simple_choice'):
7535
7536                selector := sel_names(sel) ;
7537                check_void(sel) ;                              --
7538                if selector = om then
7539                   errmsg('undefined component name ', sel,'4.3.1') ;
7540                   return [ ] ;
7541                end if;
7542                choices with:= selector ;
7543                if selector ∈ components_seen then
7544                   errmsg('Duplicate value for component ' + sel,
7545                                ' in aggregate','4.3.1') ;
7546                else
7547                  components_seen with:= selector ;
7548                  c_expr2 := res2_check(c_expr, type_of(selector)) ;
7549                   comp_pos +:= 1 ;
7550                end if ;
7551
7552
7553          ('range_choice'):
7554               errmsg('Range choice not allowed in record aggregate',
7555                                                  '','4.3.1');
7556                return [ ] ;
7557
7558          ('others_choice'):
7559
7560                if # components_seen = # all_component_names then
7561                   errmsg('OTHERS choice must represent at least ',
7562                                          'one component','4.3.1');
7563                else
7564                  (forall selector ∈ all_component_names |
7565                             selector  components_seen)
```

```
7566
7567                    c_expr2 := res2_check(c_expr, type_of(selector)) ;
7568                    if c_expr2 = [ ] then
7569                    errmsg('OTHERS expression incompatible with ',
7570                                original_name(selector) ,'4.3.1') ;
7571                    return [ ] ;
7572                    else
7573                        choices with:= selector ;
7574                    end if ;
7575                  end forall ;
7576                end if ;
7577
7578                components_seen := all_component_names ;
7579
7580            end case ;
7581        end for ;
7582        comp_list with:= [choices, c_expr2] ;
7583        if #{base_type(type_of(selector)) : selector ∈ choices} /= 1
7584        then
7585            errmsg('components on a choice list must have same type',
7586                                                    '','4.3.1') ;
7587        end if ;
7588        else                                -- Positional record aggregate.
7589            if comp_pos > #all_component_names then
7590                errmsg('Too many components ', ' for record aggregate',
7591                                        'none') ;
7592                return [ ] ;
7593            end if;
7594            selector := all_component_names(comp_pos) ;
7595            comp_list with:= [[selector],
7596                            res2_check(arg, type_of(selector))] ;
7597            comp_pos +:= 1 ;
7598            components_seen with:= selector;
7599        end if ;
7600    end forall ;
7601
7602
7603 if exists selector ∈ all_component_names | selector
7604                                            components_seen   then
7605    errmsg('No value supplied for component ',
7606                original_name(selector),'4.3.1');
7607    return [ ] ;
7608 end if ;
7609
7610
7611 return ['record_aggregate', aggregate_type,  comp_list] ;
7612
7613
7614
```

```
7615 end proc complete_r_aggregate ;
```

valid_task_name

```
7617 proc valid_task_name(task_name) ;
7618
7619 -- First pass over an expression that must yield a task type: called to
7620 -- resolve entry names.
7621
7622 [t1, types] := resolve1(task_name) ;
7623
7624 task_types := {t ∈ types | is_task_type(t)
7625                 or is_access(t) and is_task_type(designated_type(t))} ;
7626
7627 if task_types = {} then
7628    errmsg('expect task name ', '', '9.5') ;
7629 end if ;
7630
7631 return [t1, task_types] ;
7632
7633 end proc valid_task_name;
```

complete_task_name

```
7635 proc complete_task_name(task1, context_typ) ;
7636
7637 -- Complete resolution of task name used in an entry name.The context_typ
7638 -- is obtained from the scope of the resolved entry name. Derived task
7639 -- types have the same entries as their root type, and the unique type of
7640 -- the task name is thus the one whose root type is the context type.
7641
7642 types := task1(2) ;
7643 if exists t ∈ types | (root_type(t) = context_typ) then
7644    task_name := resolve2(task1, t) ;
7645 elseif exists t ∈ types | (is_access(t) and
7646                 root_type(designated_type(t)) = context_typ) then
7647    task_name := ['@', resolve2(task1, t)] ;
7648 else                                          -- previous error.
7649    return [ ] ;
7650 end if ;
7651
7652 -- Within the task body a task type designates the object currently exe-
7653 -- cuting that task. We replace the task type with what will be its
7654 -- run-time identity.
7655
7656 if is_identifier(task_name) and is_task_type(task_name) then
7657    if task_name ∈ open_scopes then
7658       task_name := declared(task_name)('current_task') ;
7659    else
```

```
7660        -- Use of the task type otherwise is invalid.
7661        errmsg('invalid use of task type outside of its own body', '',
7662                                                        '9.1') ;
7663     end if ;
7664  end if ;
7665
7666  return task_name ;
7667  end proc complete_task_name ;
```

is_task_type

```
7669  proc is_task_type(task) ;
7670
7671  return nature(task) ∈ ['task_type', 'task_type_spec'] ;
7672
7673  end proc is_task_type ;
```

valid_attribute

```
7675  proc valid_attribute(expn) ;
7676
7677  -- The prefix of an attribute may be a type, in which case type reso-
7678  -- lution is not applied  to it. The prefix of ADDRESS can also be an
7679  -- entity that is  not type-resolved, such as  a package. Finally the
7680  -- prefix of COUNT must be  an entry name, on which additional checks
7681  -- apply.
7682
7683  [-, attribute, arg1, arg2] := expn ;
7684
7685  if attribute = 'ADDRESS' then
7686     if is_identifier(arg1) then                    -- simple entity
7687        if is_type(arg1) or is_literal(arg1) then
7688           return pass1_error('invalid argument for ADDRESS','',
7689                                                        'Annex A') ;
7690        elseif nature(arg1) = 'out' and is_access(type_of(arg1)) then
7691           return pass1_error('invalid reading of out parameter','',
7692                                                        '4.1(4)') ;
7693        else
7694           arg11 := [arg1, {type_of(arg1) ? 'any'}] ;
7695        end if ;
7696     elseif is_overloaded(arg1) then                -- subprogram or entry
7697        arg11 := [arg1, {type_of(a): a ∈ arg1}] ;
7698     else
7699        if arg1(1)   ['.', '[ ]', '[..]', '@'] then return
7700          pass1_error('invalid argument for ADDRESS','','Annex A');
7701        else
7702          arg11 := resolve1(arg1) ;                -- name of entity
7703        end if ;
7704     end if ;
```

```
7705
7706 elseif attribute = 'COUNT' then
7707     [task1, entry_names, indx] := find_entry_name(arg1) ;
7708
7709     if task1 = om then      -- previous error.
7710         noop_error := true ;
7711         return [[ ], {}] ;
7712
7713     elseif # entry_names > 1 then
7714         errmsg('ambiguous entry name', '', '9.5') ;
7715       return [[ ], {}] ;
7716     else
7717       entry_name := arb entry_names ;
7718         task := complete_task_name(task1,type_of(scope_of(entry_name)));
7719     end if ;
7720
7721     if not is_identifier(task)
7722         -- The COUNT attribute can only be used immediately within
7723         -- the object executing the task body.
7724         or (not exists s = open_scopes(i) | task = s
7725         or original_name(task) = 'current_task' and scope_of(task) = s)
7726         or (exists s1 ∈ open_scopes(1..i-1) |
7727                     nature(s1)   {'block','entry', 'entry_family'})
7728     then
7729           return pass1_error('E''COUNT can only be used within ',
7730                       ' the body of the task containing E', '9.9') ;
7731
7732     elseif indx /= om then
7733         -- entry is member of entry family.
7734         range_typ := index_type(type_of(entry_name)) ;
7735         entry_name:=[ '[ ]', entry_name, [check_type(range_typ, indx)]] ;
7736
7737     end if ;
7738
7739     arg11 := [entry_name, {'none'}] ;
7740
7741 elseif is_type(arg1) then
7742
7743    type1 := arg1;
7744    if is_task_type(arg1)                                    --
7745     and attribute   ['BASE', 'SIZE', 'STORAGE_SIZE']  then
7746      -- may refer to current task
7747       if arg1 ∈ open_scopes then
7748           arg1 := declared(arg1)('current_task') ;
7749       else
7750          -- Use of the task type otherwise is invalid.
7751          errmsg('invalid use of task type outside of its own body',
7752                                                    '', '9.1') ;
7753       end if ;
```

```
7754     end if ;
7755     arg11 := [arg1, {type1}] ;
7756
7757 else                                    -- prefix of any other attribute
7758     arg11 := resolve1(arg1) ;
7759 end if ;
7760                                                          --
7761 [-, types] := arg11 ;
7762 if #types /= 1 then
7763    return pass1_error('Invalid argument for attribute ',
7764                                    attribute, 'Annex A, 4.1.4') ;
7765 else
7766     type1 := arb types ;
7767 end if ;
7768
7769 -- Verify that the type has received a full declaration.
7770 if is_incomplete_type(type1) then
7771     -- 'SIZE and 'ADDRESS can be applied to a deffered constant, in the
7772     -- default expression for record  components and non-generic formal
7773     -- parameters. The nature of  the current scope  is either 'record'
7774     -- or 'void' (indicating a formal part or discriminant part).
7775
7776     if not is_type(arg1)
7777       and attribute ∈ ['SIZE', 'ADDRESS']
7778       and nature(scope_name) ∈ ['void', 'record']
7779     then  pass ;
7780     else  return premature_access(type1) ;
7781     end if ;
7782
7783 end if ;
7784 if is_identifier(arg2) then             -- May also be a type name
7785     arg21 := [arg2, {type_of(arg2)}] ;
7786 else
7787     arg21 := resolve1(arg2) ;           -- Begin resolution of second arg
7788 end if ;
7789
7790 [-, types2] := arg21 ;
7791 if types2 = {} then                     -- Some type error .
7792         return arg21 ;
7793 end if ;
7794
7795 -- Verify that attributes have the proper number of arguments. If the
7796 -- second argument is absent, the internal constant I:1 is present.
7797
7798 if is_scalar_type(type1) and attribute ∈ {'FIRST','LAST'}
7799 then
7800     if arg2 /= 'I:1' then
7801                 pass1_error('Invalid second argument for attribute ',
7802                                     attribute, 'Annex A, 4.1.4') ;
```

```
7803    elseif (is_identifier(arg1) and not is_type(arg1) )
7804     or (is_tuple(arg1) and arg1(1..2) /= ["'", 'BASE']) then
7805       pass1_error(attribute,
7806                ' cannot be applied to a scalar object','Annex A') ;
7807    end if ;
7808
7809 elseif attribute ∈
7810    ['POS', 'VAL', 'PRED', 'SUCC', 'VALUE', 'IMAGE']
7811 then
7812    if arg2 = 'I:1' then
7813        pass1_error('Missing second argument for attribute ',
7814                                                attribute, 'Annex A') ;
7815    elseif not (is_type(arg1)
7816     or (is_tuple(arg1) and arg1(1..2) = ["'", 'BASE']) )
7817    then
7818        pass1_error('First argument of attribute ' + attribute,
7819                ' must denote a type', 'Annex A') ;
7820    end if ;
7821
7822 end if ;
7823
7824 if attribute ∈ ['FIRST', 'LAST', 'RANGE', 'LENGTH'] and
7825  (is_array(type1) or
7826   is_access(type1) and is_array(designated_type(type1)))
7827 then
7828    if is_access(type1) then
7829        -- If the argument is an access object, dereference now.
7830       if is_fully_private(type1) then                    --
7831           return premature_access(type1) ;
7832       end if ;
7833
7834       type1 := designated_type(type1) ;
7835       if is_type(arg1) then
7836          arg11 := [type1, {type1}] ;
7837       else                                          -- Dereference object
7838           arg11 := [ ['@', arg11], {type1}] ;
7839        end if ;
7840    end if ;
7841
7842    if is_type(arg1) and can_constrain(arg1) then
7843        return pass1_error('Unconstrained array type for attribute ',
7844                                              attribute, '3.6.2') ;
7845    end if ;
7846    -- The second argument must be a universal integer, and
7847    -- and must be static. Complete its resolution now.
7848
7849    if 'universal_integer' ∈ types2 then
7850        arg2 := resolve2(arg21, 'universal_integer');
7851        arg2 := specialize(arg2, 'INTEGER') ;
```

```
7852    else pass1_error('index number of attribute '+attribute,
7853                         ' must be universal', 'Appendix A') ;
7854    end if;
7855
7856    if arg2 = [ ] then                          -- type error.
7857        arg2 := ['ivalue', 1];                  -- assume 1
7858    end if;
7859
7860    -- arg21 is returned in the attribute expression. In the case of
7861    -- array attributes, no further resolution is needed:
7862    arg21 := arg2 ;
7863
7864    if not is_value(arg2) or
7865      not is_integer(dim := arg2(2)) then
7866        pass1_error('Second argument of '+ attribute,
7867                                ' must be static integer', '3.6.2') ;
7868        dim := 1;                               -- assume 1
7869    else
7870        dim := arg2(2) ;
7871    end if ;
7872
7873    a_type := type1;
7874    if dim > no_dimensions(type1) or dim < 1 then
7875        return  pass1_error('Invalid dimension number for array type',
7876                                        ' in attribute', '3.6.2') ;
7877    end if ;
7878
7879    if attribute = 'LENGTH' then
7880        a_type := 'universal_integer' ;
7881    else            -- Get type of index for specified dimension.
7882        a_type := index_types(a_type)(dim) ;
7883    end if ;
7884
7885 elseif attribute = 'ADDRESS' then
7886    if 'SYSTEM'  vis_mods then
7887       -- The use of ADDRESS is incorrect if its type cannot be named
7888        errmsg('use of SYSTEM.ADDRESS requires presence of package',
7889                                ' SYSTEM', '13.7.2, Annex A') ;
7890       a_type := 'INTEGER' ;                    -- Closest thing we've got.
7891    else
7892       a_type := visible('SYSTEM')('ADDRESS') ;
7893    end if ;
7894
7895 elseif type_of(attribute) /= 'overloaded' then
7896    a_type := type_of(attribute) ;
7897
7898 elseif attribute ∈ base_attributes then
7899            a_type := base_type(type1) ;
7900 else
```

```
7901    a_type := type1 ;
7902 end if ;
7903
7904 root := root_type(type1) ;
7905
7906 if is_identifier(arg1) and (arg1 ∈ open_scopes)
7907  and nature(arg1) = 'record' then                              --
7908    return pass1_error('Invalid self-reference in definition of ',
7909                                  original_name(arg1), '3.1');
7910 end if;
7911
7912 -- Final type verification for all attributes.
7913 case attribute of
7914
7915    ('ADDRESS'):
7916         type_ok := true;                          -- checked above.
7917
7918    ('BASE'):
7919         type_ok := is_type(root) ;
7920
7921    ('FIRST', 'LAST'):
7922         type_ok := is_scalar_type(root) or is_array(root) ;
7923
7924    ('VALUE'):
7925         type_ok := is_discrete_type(root)
7926           and exists t2 ∈ types2 |compatible_types('STRING',t2);
7927
7928    ('IMAGE', 'POS', 'PRED', 'SUCC'):
7929         type_ok := is_discrete_type(root)
7930           and exists t2 ∈ types2 | compatible_types(type1,t2);--
7931
7932    ('VAL'):
7933         type_ok := is_discrete_type(root)
7934            and exists t2 ∈ types2 | is_integer_type(root_type(t2));
7935
7936    ('AFT', 'BITS', 'DELTA', 'FORE'):
7937         type_ok := root = '$FIXED';
7938
7939    ('DIGITS',      'EMAX',          'EPSILON',
7940     'MACHINE_RADIX', 'MACHINE_MANTISSA', 'MACHINE_EMAX',
7941     'MACHINE_EMIN', 'SAFE_EMAX'):
7942         type_ok := root = 'FLOAT';
7943
7944    ('LARGE', 'MACHINE_ROUNDS', 'MACHINE_OVERFLOWS', 'MANTISSA',
7945     'SMALL', 'SAFE_LARGE',    'SAFE_SMALL') :
7946         type_ok := root ∈ {'$FIXED', 'FLOAT'} ;
7947
7948    ('LENGTH', 'RANGE'):
7949         type_ok := is_array(root) ;
```

```
7950
7951    ('CONSTRAINED'):
7952        type_ok := if is_type(arg1) then is_private(type1)
7953                   else is_record(root) and has_discriminants(root) end;
7954
7955    ('PRIORITY','TERMINATED','CALLABLE'):
7956        type_ok := is_task_type(root) ;
7957
7958    ('STORAGE_SIZE'):
7959        type_ok := is_task_type(root) or is_access(root) ;
7960
7961    ('WIDTH'):
7962        type_ok := is_discrete_type(root) ;
7963
7964    ('FIRST_BIT', 'LAST_BIT', 'SIZE', 'POSITION'):
7965        type_ok := true ;
7966
7967    ('COUNT'):
7968        type_ok := true ;
7969 else
7970    return pass1_error('Undefined attribute: ', attribute, 'Annex A') ;
7971 end case ;
7972
7973 return if type_ok then
7974   [["'", attribute, arg11, arg21], {a_type}]
7975 else
7976    pass1_error('Invalid argument type for attribute ',
7977                                          attribute, 'Annex A')
7978 end ;
7979
7980 end proc valid_attribute ;
7981
```

res2_check

```
7983 proc res2_check(expn2, context_type) ;
7984
7985 -- Called to impose constraints when needed, on aggregate components and
7986 -- allocated objects. These are non-sliding contexts for aggregates.
7987
7988 may_others := full_others ;
7989 full_others := true ;
7990 e := apply_constraint(resolve2(expn2, context_type), context_type) ;
7991 full_others := may_others ;
7992
7993 if is_tuple(e) and e(1) = 'qual_length' then
7994    e(1) := 'qual_index' ;                    -- indicate no sliding
7995 end if ;
7996
```

```
7997 e := eval_static(e);    --
7998 return if is_value(e) and #e = 2 then (e with context_type) else e end ;
7999
8000 end proc res2_check ;
```

compatible_types

```
8002 proc compatible_types(t_out, t_in) ;
8003
8004 -- This procedure verifies that an expression of type -t_in- can appear
8005 -- in a context requiring type -t_out-. In the case of subtypes this
8006 -- procedure indicates whether a run-time check will be necessary.
8007 -- Equality, set and comparison operators carry a special type-marker which
8008 -- is ignored on the first pass of type resolution, because the type of
8009 -- the arguments of these operators have no effect on the result type.
8010 -- On the second pass, these special type-markers are used to indicate
8011 -- the need for a consistency check among the types of the two actual
8012 -- parameters themselves.
8013
8014 const multiple_types = {'equal_type', 'order_type', 'any'} ;
8015
8016 const gen_types =
8017   {'array_type',   'boolean_type', 'character_type', 'composite_type',
8018    'discrete_type', '$FIXED',      'integer_type',  'real_type',
8019    'string_type',   'universal_integer', 'universal_real',
8020    'universal_fixed', 'universal_type'};
8021
8022 if t_in = om or t_out = om                    -- syntax error
8023  or (t_in = t_out )                           -- compatible types
8024  or t_in ∈ multiple_types or t_out ∈ multiple_types then
8025    return true ;
8026 end if ;
8027
8028 -- The generic  types 'universal_integer', 'universal_real' , etc., are
8029 -- used to indicate the type of the corresponding literals. These types
8030 -- are compatible with specific  types of the same family. On the other
8031 -- hand, the generic 'universal_fixed' is  incompatible with all types,
8032 -- and its presence in any type checking will trigger an error  message
8033 -- somewhere ( but not in this routine).
8034 -- To avoid checking  for their presence on both sides, we perform  the
8035 -- following normalization :
8036
8037 if t_in   gen_types and t_out ∈ gen_types  then
8038    [t_in, t_out] := [t_out, t_in] ;
8039 end if ;
8040
8041 return case t_in of
8042
8043 ('universal_integer'): root_type(t_out) = 'INTEGER',
```

```
8044
8045 ('universal_real'):   root_type(t_out) ∈ {'FLOAT', '$FIXED'},
8046
8047 ('universal_type'):   t_out ∈ univ_types ,
8048
8049 ('$FIXED'):           t_out = 'universal_real'
8050                              or root_type(t_out)='$FIXED' ,
8051
8052 ('boolean_type'):     root_type(t_out) = 'BOOLEAN' or is_array(t_out)
8053                       and root_type(component_type(t_out)) = 'BOOLEAN',
8054
8055 ('discrete_type'):    is_discrete_type(t_out),
8056
8057 ('integer_type'):     root_type(t_out) = 'INTEGER'
8058                              or t_out = 'universal_integer',
8059
8060 ('real_type'):        root_type(t_out) ∈
8061                              {'FLOAT', '$FIXED', 'universal_real'},
8062
8063 ('string_type'):      is_array(t_out) and no_dimensions(t_out) = 1
8064              and is_character_type(component_type(t_out)) ,
8065
8066 ('character_type'):   is_character_type(t_out),
8067
8068 ('array_type'):       is_array(t_out),
8069
8070 ('composite_type'):   nature(root_type(t_out)) ∈ ['array', 'record'],
8071
8072 ('universal_fixed'):  false
8073
8074 else
8075   -- name equivalence of base types holds for everything else.
8076   base_type(t_in) = base_type(t_out)
8077
8078 end ;
8079
8080 end proc compatible_types;
```

type_error

```
8082 proc type_error(op_names,typ,num_types) ;
8083
8084
8085 -- Emit error message after a type error was detected during
8086 -- type resolution.
8087 -- if num_types > 1, the expression is ambiguous : the operator of
8088 -- op_names is overloaded, and the argument list is not sufficient to
8089 -- disambiguate.
8090 -- If num_types=0, the argument list is incompatible with the op.
```

```
8091
8092 op_name := arb op_names ;
8093 op_n := original_name(op_name) ;
8094 if num_types > 1 then
8095    case nature(op_name) of
8096
8097    ('procedure','function','procedure_spec','function_spec'):
8098       names := +/[original_name(scope_of(x)) + '.' +
8099                   original_name(x) + ' ' : x ∈ op_names] ;
8100       errmsg('Ambiguous call to one of ', names,'6.6, 8.3') ;
8101
8102    ('op'):
8103       errmsg('Ambiguous operands for ', op_n,'6.7, 8.3') ;
8104
8105    ('literal'):
8106       errmsg('Ambiguous literal: ', op_n,'3.5.1, 4.7, 8.3') ;
8107    else errmsg('ambiguous expression','','8.2, 8.3') ;
8108    end case  ;
8109
8110    -- If the type is ambiguous the expression is of couse invalid.
8111
8112    noop_error := true ;
8113
8114 else              -- Num_types is zero.
8115    if noop_error then
8116       -- Current expression contained previous error. Do not emit
8117       -- an aditional one.
8118       return ;
8119    end if ;
8120
8121    noop_error := true ;        -- For sure.
8122
8123    if typ = om then   -- Operator or subprogram .
8124
8125       if op_n ∈ {'GET', 'PUT'} then
8126          errmsg('Text_io not instantiated nor defined for type',
8127                      '','8.4, 14.4') ;
8128        else
8129          if nature(op_name) ∈ {'entry', 'entry_family'} then
8130             op_n := 'entry call' ;
8131          end ;
8132          msg1 := if nature(op_name) = 'op' then
8133                        'invalid types for '
8134                  else    'invalid argument list for '   end;
8135          errmsg(msg1, op_n,'none') ;
8136       end if ;
8137
8138    elseif nature(op_name) = 'literal' then
8139       errmsg('no instance of ' + op_n,
```

```
8140                        ' has type ' + full_type_name(typ),'3.5.1') ;
8141    else
8142            errmsg('Expect expression to yield type ',
8143                        full_type_name(typ),'none') ;
8144    end if ;
8145 end if ;
8146
8147
8148 end proc type_error ;
8149 --+S premature_access
8150 proc premature_access(type_mark) ;
8151
8152 -- Called when trying to use ( an access to ) a fully private type.
8153 return
8154 pass1_error('Premature usage of access, private or incomplete type ',
8155                                 original_name(type_mark), '7.4.2') ;
8156 end proc premature_access ;
```

pass1_error

```
8158 proc pass1_error(msg1, msg2, lrm_sec) ;
8159 -- This procedure is invoked when a type error which requires a special
8160 -- message is encountered in resolve1.
8161
8162 if not noop_error then
8163       errmsg(msg1, msg2, lrm_sec) ;
8164 end if ;
8165 noop_error := true ;          -- To avoid cascaded errors.
8166
8167 return [ [ ], {} ] ;         -- Error template for resolve1.
8168
8169 end proc pass1_error ;
```

full_type_name

```
8171 proc full_type_name(typ) ;
8172
8173
8174 -- Error message procedure. Restore source name of type, or if anonymous
8175 -- build some approximate description of its ancestry.
8176
8177
8178 type_name := original_name(typ) ;
8179 if type_name = '' then                    -- Anonymous type.
8180    type_name :=
8181      case nature(typ) of
8182
8183         ('subtype'):  full_type_name(type_of(typ)) ,
8184
```

```
8185          ('array'):    'array(' + full_type_name(index_type(typ))+'...' ,
8186
8187          ('type'):                          -- derived type
8188                        'new ' + full_type_name(type_of(typ))
8189      else '--anonymous--' end ;
8190  end if ;
8191
8192  return type_name ;
8193
8194  end proc full_type_name ;
```

5. Statements.

assign_statement

```
8197 proc assign_statement(name, expn) ;
8198
8199
8200 noop_error := false ;                    -- To clear previous type errors
8201 e1 := resolve1(find_old(name)) ;
8202 [-, t_l] := e1 ;
8203 t_left := {t ∈ t_l | not is_limited_type(t)} ;                    --
8204 e2 := resolve1(expn) ;
8205 [-, t_right] := e2 ;
8206
8207 if noop_error then return [ ] ; end if ;          -- previous error.
8208
8209 ok_types := {[ t1, t2] : t1 ∈ t_left, t2 ∈ t_right |
8210                                   compatible_types(t1, t2)} ;      --
8211 -- For the assignment to be unambiguous, the left-hand and right_hand
8212 -- sides must have a single compatible interpretation.
8213
8214 if #ok_types = 0 then
8215    if #t_l = 1 and #t_left = 0 then
8216       errmsg('assignment not available on a limited type','','7.4.2') ;
8217       return [ ] ;
8218    else
8219       errmsg('incompatible types for assignment','','5.2') ;
8220       return [ ] ;
8221    end if ;
8222
8223 elseif
8224  (#ok_types > 1          -- ambiguous expression.
8225    -- Ambiguity may be between op and function yielding the same
8226    -- base type. Ambiguity will be lifted in resolve2. Otherwise:
8227   and #{base_type(t): [t,-] ∈ ok_types} > 1
8228   and  #t_left > 1) then       -- ambiguous left-hand side.
8229    errmsg('ambigous types for assigment','','5.2') ;
8230    return [ ] ;
8231 else
8232    [t1,t2] := arb ok_types ;
8233 --
8234    out_context := true ;
8235    e1 := resolve2(e1, t1) ;
8236    out_context := false ;
```

```
8237
8238    if not is_variable(e1) then
8239     errmsg('left-hand side in assignment ',' is not a variable','5.2');
8240     return [ ] ;
8241    end if ;
8242
8243    -- If the left-hand side  is a slice,  its bounds must be imposed on
8244    -- the right-hand side. We create a subtype of the index type of the
8245    -- lhs, whose bounds are given by the slice bounds. A special slice-
8246    -- assignment instruction is emitted, with the following structure:
8247    --              [ '[..]:=', index, index_definition, lhs, rhs]
8248
8249    if is_tuple(e1) and e1(1) = '[..]' then
8250        [-, obj, discr_range] := e1 ;
8251
8252        if is_identifier(discr_range) then
8253           ix := discr_range ;
8254        else                                -- Explicit range. Create anon.
8255           ix := str newat ;                -- entry for it.
8256           SYMBTAB(ix) := discr_range ;
8257           root_type(ix) := root_type(index_type(t1)) ;
8258        end if ;
8259
8260        a_type := str newat ;               -- Build constrained array type.
8261        SYMBTAB(a_type) := ['subtype', base_type(t1),
8262                                        [[ix], component_type(t1), 1]] ;
8263        root_type(a_type) := root_type(t1) ;
8264
8265        -- Now resolve right-hand side with it.
8266       e2 := resolve2(e2, a_type) ;
8267
8268        -- Label right-hand side with array type.
8269        if is_tuple(e2) then top(e2) := t1; end if;
8270
8271       return ['[..]:=', ix, subtype_expr(ix), obj, e2] ;
8272
8273    else          -- Impose the unique type of the lhs on the rhs.
8274        e2 := resolve2(e2, t1);
8275    end if ;
8276
8277 end if;
8278
8279 --
8280 if is_identifier(e2) and is_type(e2) then
8281     errmsg('Invalid right-hand side in assignment', '', '5.2') ;
8282 end if ;
8283
8284 -- constraints on assignment are special-cased for the interpreter.
8285 if is_array(t1) then
```

```
8286    -- Array assignments are length_checked in the interpreter and
8287    -- do not need an explicit qualification.
8288    rhs := e2 ;
8289 else
8290    -- Apply constraint and try to constant fold.
8291    rhs := eval_static(apply_constraint(e2, t1)) ;
8292
8293    --- Label right-hand side with its type (temporary).
8294    if is_value(rhs) and #rhs=2 then rhs with:= t1; end if ;
8295
8296 end if ;
8297
8298 return  [':=', e1, rhs] ;
8299
8300 end proc assign_statement ;
8301
```

is_variable

```
8303 proc is_variable(xpr) ;
8304
8305 -- Verify that an expression is a variable name. This is called for
8306 -- assignment statements, when validating  -out- and -inout-
8307 -- parameters in a procedure or entry call, and for generic inout parms.
8308
8309  return if is_identifier(xpr)
8310              then nature(xpr) ∈ {'obj', 'inout', 'out'}
8311          else
8312            case xpr(1) of
8313
8314                ('[ ]', '[..]') : is_variable(xpr(2)),
8315
8316                ('.') : is_variable(xpr(2))
8317                        and nature(xpr(3)) = 'obj',
8318
8319                ('@') : is_identifier(access := xpr(2)) or
8320                        is_variable(access) or
8321                --- More detailed check needed here.
8322                   is_tuple(access) and access(1) = 'call'
8323                   and is_access(type_of(access(2))) ,
8324
8325                ('convert'): is_variable(xpr(4))
8326
8327              else false
8328              end
8329
8330    end ;
8331
8332  end  proc is_variable ;
```

8333

case_statement

```
8335 proc case_statement(expn, case_nodes) ;
8336
8337
8338
8339 cexp := check_type('discrete_type', expn) ;
8340 if cexp = [ ] then                              -- type error
8341     return [ ] ;
8342 else
8343     exptype := expr_type(cexp) ;
8344
8345     if exptype = 'universal_integer' then
8346         cexp := specialize(cexp, 'INTEGER') ;
8347     end if;
8348
8349     case_list := [[ [adasem(c): c ∈ choices], adasem(stats)] :
8350                                   [choices, stats] ∈ case_nodes] ;
8351
8352     return ['case', cexp] + process_case(exptype, case_list) ;
8353 end if;
8354
8355 end proc case_statement ;
8356
8357
8358
8359 proc process_case(exptype, case_list);
8360
8361
8362 --
8363 -- This procedure is given the type of the case expression and
8364 -- uses this type to resolve the choices appearing in the case_list.
8365 -- It also checks that the choices are static expressions and
8366 -- constructs the case statement intermediate code.
8367 -- It is called both for case statements and for variant parts.
8368 --
8369 -- The case_list has the form
8370 --
8371 --      case_list ::= [ [choice_list, statement_list] ... ]
8372 --
8373 -- where a choice_list is a sequence of choices,
8374 --
8375 --          choice_list ::= [choice ...]
8376 --
8377 -- each of the form
8378 --
8379 --          choice ::= ['simple_choice', simp_expr,   node]
```

```
8380 --                 |['range_coice',   discr_range, node]
8381 --                 |['others_choice', '',          node]
8382 --
8383 -- where -node- is carried along for error reporting purposes.
8384 -- The case statement intermediate code has the form
8385 --
8386 --      case_statement ::= ['case', expr, altlist, others]
8387 --
8388 -- where
8389 --      altlist  ::=  { {choice} →g statement_list}
8390 -- and
8391 --      choice ::=  choiceval | ['range', choiceval,choiceval]
8392 --
8393 --
8394
8395
8396 --
8397 -- Check that the case expression is of a discrete type
8398 -- and that its range is static, and find the length of
8399 -- the range.
8400 --
8401    invalid_case_type := false;
8402    save_node := current_node ;       -- Use node of the choices
8403
8404    if exptype = 'universal_integer' then              -- literal expr
8405        exp_base_type := 'INTEGER';                        -- force INTEGER
8406    else
8407       exp_base_type := base_type(exptype);
8408    end if;
8409
8410    if not is_discrete_type(exp_base_type) then
8411        errmsg('Case expression not of discrete type','','3.7.3, 5.4');
8412        invalid_case_type := true;     -- Still check the alternatives
8413
8414    elseif is_generic_type(exp_base_type) then
8415        errmsg('Case expression cannot be of a generic type','','5.4') ;
8416        invalid_case_type := true ;
8417    end if;
8418
8419    [-, exp_lo, exp_hi] := signature(exptype) ? [ ];
8420
8421    is_static := is_static_subtype(exptype) ;                    --
8422
8423    if not is_static then
8424
8425        [-, exp_lo, exp_hi] := signature(exp_base_type) ? [ ];
8426        if not is_value(exp_lo) or not is_value(exp_hi) then
8427           -- This alternative can arise only if the type of the
8428           -- case expression does not have static bounds.  This
```

```
8429            -- has alreay been caught, so we give no error message here,
8430            -- but only the choices are type checked and no code put out.
8431            invalid_case_type := true;
8432          end if;
8433      end if;
8434
8435      if not invalid_case_type then
8436         exp_lov := exp_lo(2);
8437         exp_hiv := exp_hi(2);
8438         range_size := (exp_hiv - exp_lov + 1) max 0 ;                --
8439      end if;
8440
8441 --
8442 -- Now check each of the case choices against exp_base_type, ensure
8443 -- that each is static, and construct the case statement code.
8444 --
8445      resolved_case_list := {};
8446      others_part := [ ];
8447      is_others_part := false;
8448      valset := {};
8449      numval := 0;
8450      if case_list /= [ ] then                -- empty case list is allowed
8451           last_clist := top(case_list)(1);
8452           if #last_clist = 1 and last_clist(1)(1) = 'others_choice' then
8453              is_others_part := true;
8454              [-, stmt_list] frome case_list;
8455              others_part +:= stmt_list; -- pick up others part
8456           end if;
8457           (for [choice_list, stmt_list] ∈ case_list)
8458              choice_alt := {};
8459              (for [choice_kind, choice, current_node] ∈ choice_list)
8460
8461            -- Type check the choice and ensure that it is static,
8462            -- that it is in the range for the expression subtype, and
8463            -- that it appears no more than once as a value possibility.
8464
8465                  case choice_kind of
8466                    ('others_choice'):
8467                      errmsg('The choice OTHERS must appear '
8468                             'alone and last','','3.7.3, 5.4');
8469                      is_others_part := true;
8470                      continue;
8471
8472                    ('simple_choice'):
8473
8474                      if choice(1) = 'name' and is_identifier(choice(2))
8475                      and is_type(choicev:= find_old(choice(2)))
8476                      then
8477                        t := choicev ;
```

```
8478                    if not compatible_types(t, exp_base_type)then
8479                        errmsg('choice must have type ',
8480                            original_name(exp_base_type), '5.4') ;
8481                        choicev := [ ] ;
8482                    end if ;
8483                else
8484                    choicev := check_type(exp_base_type, choice);
8485                end if ;
8486
8487                if choicev = [ ] or invalid_case_type then
8488                    continue;
8489                elseif is_value(choicev) then
8490                    lov := hiv := choicev(2);
8491                elseif is_identifier(choicev) and is_type(choicev)
8492                then                    -- discrete range
8493                    [-, lov, hiv] := signature(choicev);
8494                    if is_value(lov) and is_value(hiv) then
8495                        [lov, hiv] := [lov(2), hiv(2)];
8496                    else
8497                        errmsg('Case choice not static', '',
8498                                                    '3.7.3, 5.4');
8499                        continue;
8500                    end if;
8501                else
8502                    errmsg('Case choice not static', '',
8503                                                    '3.7.3, 5.4');
8504                    continue;
8505                end if;
8506
8507              ('range_choice'):
8508                choicev := check_type(exp_base_type, choice);
8509                if choicev = [ ] or  invalid_case_type then
8510                    continue;
8511                else
8512                    [-,-,[-,choicev1, choicev2]] := choicev ;
8513                  if is_value(choicev1) and is_value(choicev2)
8514                  then
8515                      [lov, hiv] := [choicev1(2), choicev2(2)];
8516                  else
8517                      errmsg('Case choice not static', '',
8518                                                    '3.7.3, 5.4');
8519                      continue;
8520                    end if ;
8521                end if;
8522            end case;
8523 --
8524 -- At this point the choice is known to be static and is expressed as
8525 -- a range [lov, hiv].
8526 --
```

```
8527                if is_static and (lov ≤ hiv) and
8528                                    ( lov < exp_lov or hiv > exp_hiv)
8529                then
8530                   errmsg('choice value(s) not in range of static',
8531                        ' subtype of case expression', '5.4') ;
8532                end if ;
8533                lov max:= exp_lov;    -- Remove junk values from below
8534                hiv min:= exp_hiv;    -- Remove junk values from above
8535
8536                if lov > hiv then     -- Null range -- ignore it.
8537                   continue;
8538                end if;
8539
8540 -- Ensure that range is disjoint from all others.
8541
8542                range_choice := hiv > lov;
8543                duplicate_choice := false;
8544
8545                if exists [a, b] ∈ valset | lov ≥ a and lov ≤ b then
8546                   duplicate_choice := true;
8547                   lov := b + 1;
8548                end if;
8549
8550                if range_choice
8551                and
8552                  (exists [a, b] ∈ valset | hiv ≥ a and hiv ≤ b)
8553                then
8554                   duplicate_choice := true;
8555                   hiv := a - 1;
8556                end if;
8557
8558                if range_choice
8559                and
8560                  (exists [a, b] ∈ valset | lov < a and hiv > b)
8561                then
8562                   duplicate_choice := true;
8563                end if;
8564
8565                if duplicate_choice then
8566                   errmsg('Duplicate choice value(s)', '',
8567                                                       '3.7.3, 5.4');
8568                end if;
8569
8570                if lov > hiv then     -- Again check for null range
8571                   continue;
8572                end if;
8573
8574 --
8575 -- Add interval to set of values seen so far, add the number of choices
```

```
8576 -- to the count, and add to the case alternative.
8577 --
8578                 valset with:= [lov, hiv];
8579                 numval +:= (hiv - lov + 1);
8580                 choice_alt with:= if lov = hiv then lov
8581                                   else ['range', lov, hiv] end;
8582
8583             end for;
8584             resolved_case_list(choice_alt) := stmt_list ;
8585         end for;
8586     end if;
8587     current_node := save_node ;
8588 --
8589 -- Check that all of the possibilities in the range of the
8590 -- case expression have been used.
8591 --
8592     if not invalid_case_type
8593     and
8594         not is_others_part
8595     and
8596         (numval /= range_size or exptype = 'universal_integer')
8597     then
8598         errmsg('Missing OTHERS choice', '','3.7.3, 5.4');
8599         others_part +:= [ ];            -- add null others_part
8600     end if;
8601 --
8602     return [resolved_case_list, others_part];
8603
8604 end proc process_case;
8605 --
```

is_static_subtype

```
8607 proc is_static_subtype(subtype) ;
8608
8609 bt := type_of(subtype) ;
8610 if is_generic_type(bt)
8611    or not is_scalar_type(bt)  -- RM 4.9 (11)
8612 then
8613    return false ;
8614 elseif bt = subtype then
8615    return true ;
8616 else
8617    [-, lo, hi] := signature(subtype) ;
8618    return is_static_subtype(bt) and
8619            is_value(lo) and is_value(hi) ;
8620 end if ;
8621
8622 end proc is_static_subtype ;
```

```
8623 --
```

new_block

```
8625 proc new_block(id) ;
8626
8627
8628
8629 --
8630 -- block names are declared when procedure containing them is entered.
8631 block_id := find_old(id) ;
8632
8633 NATURE(block_id) := 'block' ;
8634
8635 newscope(block_id) ;
8636
8637 return block_id ;
8638
8639 end ;
8640
```

block

```
8642 proc block(decls,stats, handler) ;
8643
8644 check_incomplete_decls(scope_name,decls) ; --
8645
8646 --
8647 -- Constraints on allocators appearing in the statement part may generate
8648 -- anonymous types which need not be emitted, because they are replaced
8649 -- in the aix by the corresponding subtype expressions.
8650 top (NEWTYPES) := [ ] ;
8651
8652 -- Insert the proper end statement at each exit from the block. Exits
8653 -- include the statement list and each one of the exception handlers.
8654 -- For procedures, the end statement is a RETURN.
8655
8656 if is_proc(scope_name) then
8657    -- Insert return statement at end.
8658    if nature(scope_name) ∈ {'function', 'generic_function'} then
8659       return end_function(decls, stats, handler) ;
8660    else
8661       exit_stat := return_statement(om);
8662    end if ;
8663
8664 elseif NATURE(scope_name) ∈ {'task_obj', 'task_type'} then
8665
8666    exit_stat := terminate_statement() ;                    --
8667
```

```
8668 else                                   -- scope is a block.
8669    exit_stat := ['end'] ;
8670 end if ;
8671
8672 stats with:= exit_stat ;
8673
8674 if not is_empty(handler) then
8675    [[-, hlist]] := handler ;
8676    new_hlist := [ ] ;
8677    (for [e_list, s] ∈ hlist)
8678       new_hlist with:= [e_list, s with exit_stat] ;
8679    end for ;
8680    handler := [['exception', new_hlist]];
8681 end if ;
8682
8683 return [decls,  stats, handler] ;
8684
8685 end proc block;
```

end_function

```
8687 proc end_function(decls, stats, handler) ;
8688
8689 -- For functions, the exception PROGRAM_ERROR is raised if an explicit
8690 -- RETURN statement is not found. This is implemented by appending the
8691 -- raise statement to each possible exit path.
8692
8693 if not has_return(stats) then
8694    errmsg('function body must contain a return statement',
8695                                                    '','6.5');
8696 end if ;
8697 exit_stat := ['raise', 'PROGRAM_ERROR', 'missing RETURN'] ;
8698
8699 if top(stats)(1) /= 'return' then
8700    stats with:= exit_stat ;
8701 end if ;
8702
8703 if not is_empty(handler) then
8704    [[-, hlist]] := handler ;
8705    new_hlist := [ ] ;
8706    (for [e_list, s] ∈ hlist)
8707       if top(s)(1) /= 'return' then s with:= exit_stat; end if ;
8708       new_hlist with:= [e_list, s] ;
8709    end for ;
8710    handler := [['exception', new_hlist]];
8711 end if ;
8712
8713 return [decls,  stats, handler] ;
8714
```

```
8715 end proc end_function ;
```

has_return

```
8717 proc has_return(stats) ;
8718
8719 -- Ascertain that the statement part of a function body contains a return
8720 -- statement (simple recursive tree traversal).
8721 return if not is_tuple(stats) then false
8722       elseif is_string(stats(1)) and stats(1) = 'return' then true
8723       else (exists stat ∈ stats | has_return(stat) )
8724 end ;
8725 end proc has_return ;
```

loop_statement

```
8727 proc loop_statement(iterat, stmts) ;
8728
8729 -- Close loop statement. If label of loop is used by EXIT statement
8730 -- then emit label definition.
8731 --
8732 [used, loop_type, t_def] := signature(scope_name) ;
8733 loop_name := if used then scope_name else om end ;
8734
8735 -- For loops whose iterators are given by a discrete range, an anonymous
8736 -- type  is generated. Its  type definition is emitted with the loop.
8737
8738 if loop_type /= om then
8739     iterat +:= [loop_type, t_def] ;
8740 end if;
8741
8742 -- Exit from loop scope.
8743 popscope() ;
8744 return  iterat + [stmts, loop_name] ;
8745
8746 --
8747
8748 end proc loop_statement;
```

iter_var

```
8750 proc iter_var(id, disc_range) ;
8751
8752 -- Insert loop variable in scope of loop.
8753 loop_var := find_new(id) ;
8754
8755 -- If the iteration is given by a discrete range, construct an anonymous
8756 -- type for it, and save the defining expression. It is  emitted as part
8757 -- of the loop header.
```

```
8758
8759 iter_type := make_index(disc_range) ;
8760
8761 if top(top(NEWTYPES)) = iter_type then
8762     -- Remove from anonymous types, and save subtype definition.
8763     it frome top(NEWTYPES) ;
8764     type_def := subtype_expr(iter_type) ;
8765 else
8766     type_def := [ ] ;
8767 end if ;
8768
8769 SYMBTAB(loop_var) := ['constant', iter_type] ;
8770
8771 signature(scope_name) +:= [iter_type, type_def] ;
8772 return loop_var ;
8773
8774 end proc iter_var;
8775
```

exit_statement

```
8777 proc exit_statement(id,cond);
8778
8779 -- An unqualified exit refers to the innermost enclosing scope.
8780
8781 if id = [ ] then
8782     if exists scope ∈ open_scopes | type_of(scope) = 'loop' then
8783         -- Indicate that loop label must be emitted.
8784         signature(scope)(1) := true ;
8785     else
8786         errmsg('EXIT statement not in loop','','5.7') ;
8787         return [ ] ;
8788     end if;
8789 else
8790     -- Verify that loop label exists.
8791     if exists scope ∈ open_scopes | type_of(scope) = 'loop' and
8792       original_name(scope) = id then
8793         signature(scope)(1) := true ;
8794     else
8795         errmsg('Invalid loop label in EXIT: ', id, '5.5, 5.7') ;
8796         return [ ] ;
8797     end if ;
8798 end if ;
8799
8800 -- Now verify that the exit statement does not try to exit from
8801 -- a procedure, task, package or accept statement. This amounts
8802 -- to requiring that the scope stack contain only blocks up to the
8803 -- scope being exited.
8804
```

```
8805 (forall sc ∈ open_scopes)
8806     if sc = scope then quit ;
8807     elseif nature(sc) /= 'block' then
8808         errmsg('attempt to exit from ', nature(sc), '5.7') ;
8809         quit ;
8810     end if ;
8811 end forall ;
8812
8813 return if cond = [ ] then ['exit', scope]
8814                      else ['exit', scope, cond]  end ;
8815
8816 end proc exit_statement;
8817
```

return_statement

```
8819 proc return_statement(exp1);
8820
8821 -- Find subprogram or accept statement which is enclosing scope, and keep
8822 -- track of the  number of blocks that have to be exited.
8823
8824     certain := exists proc_name = open_scopes(i) |
8825                                   (nat:=nature(proc_name)) /= 'block' ;
8826
8827     out_depth := i - 1 ;
8828
8829     -- Exception handlers are blocks for syntactic purposes, but not at
8830     -- run-time. They must be excluded from this count.
8831     -- The same is true for loops.
8832     (forall j ∈ [1..i] | type_of(open_scopes(j)) ∈ {'handler','loop'})
8833         out_depth -:= 1 ;
8834     end forall ;
8835
8836     if nat ∈ {'function', 'procedure', 'generic_function',
8837       'generic_procedure'} then
8838         instr_name := 'return' ;
8839     elseif nat ∈ {'entry', 'entry_family'}   then
8840         instr_name := 'entry_return' ;
8841     else
8842         errmsg('invalid context for RETURN statement','','5.8') ;
8843         return [ ] ;
8844     end if ;
8845     r_type := if nat = 'entry_family' then 'none'
8846                                       else TYPE_OF(proc_name) end ;
8847     if exp1 = om then
8848         ret_exp := om ;
8849
8850     elseif exp1 /= [ ] then
8851         if r_type = 'none' then
```

```
8852            errmsg('Procedure cannot return value', '', '5.8') ;
8853            ret_exp := om ;
8854        else
8855            -- If the value returned is an aggregate, there is no sliding
8856            -- for it, and named associations can appear together with
8857            -- "others" (see 4.3.2(6)).
8858
8859            full_others := true ;
8860            ret_exp := check_type(r_type, exp1) ;
8861          full_others := false ;
8862        end if ;
8863     elseif r_type /= 'none' then
8864          errmsg('Function must return value','','5.8') ;
8865          ret_exp := [ ] ;
8866     end if ;
8867
8868    return [instr_name, proc_name, out_depth, ret_exp] ;
8869
8870 end proc return_statement;
8871 --
```

label_decl

```
8873 proc label_decl(id_list) ;
8874
8875
8876 (forall id ∈ id_list)
8877
8878    if id ∈ top(LAB_SEEN) then
8879        errmsg('Duplicate label in program unit: ', id ,'5.1') ;
8880     else
8881          top(LAB_SEEN) with:= id ;
8882        label := find_new(id) ;
8883        nature(label) := 'label' ;
8884        label_status(label) := 'unreachable' ;
8885     end if ;
8886
8887 end forall ;
8888
8889 end proc label_decl;
```

lab_init

```
8891 proc lab_init ;
8892 LAB_SEEN with:= {} ;
8893 end proc lab_init ;
8894 --
8895
```

lab_end

```
8897 proc lab_end ;
8898 old_labels frome LAB_SEEN ;
8899 end proc lab_end ;
8900 proc goto_statement(id) ;
8901
8902
8903 label := find_old(id) ;
8904 if nature(label) /= 'label' then
8905     errmsg(id, ' is not a label', '5.9') ;
8906
8907 elseif label_status(label) = 'unreachable'  then
8908     errmsg(id, ' is not a reachable label', '5.9') ;
8909
8910 else
8911     (forall s ∈ open_scopes)
8912
8913         if s = scope_of(label) then quit ;
8914       elseif nature(s) /= 'block' then
8915           errmsg('attempt to jump out of ', nature(s), '5.9') ;
8916         end if ;
8917
8918     end forall ;
8919 end if ;
8920
8921 return ['goto', label] ;
8922
8923 end proc goto_statement ;
8924 --
```

6. Subprograms

subprog_decl

```
8927 proc subprog_decl(specs) ;
8928
8929
8930
8931 --
8932 [nat, id, formals, ret] := specs ;                -- Unpack specs.
8933
8934 new_compunit('subprog spec', id) ;
8935 -- Process formals and check for overloading.
8936 name := subprog_spec([nat + '_spec', id, formals, ret]) ;    --
8937
8938 save_subprog_info ;                                -- If compilation unit
8939 return [[nat, name, build_signature(name)]] ;
8940 --
8941
8942 end proc subprog_decl ;
8943
```

subprog_spec

```
8945 proc subprog_spec(specs) ;
8946
8947
8948
8949 -- Process subprogram specification. If the subprogram name is an
8950 -- operator designator, verify that it has the proper type and number
8951 -- of arguments. Return the unique name of the subprogram, after chaining
8952 -- possible overloads.
8953
8954     [proc_nat, proc_id, formals, ret ] := specs ;
8955     prog_info := [proc_nat, ret, formals] ;
8956
8957     if proc_id ∈ op_designators then
8958         if  proc_id ∈ {'+', '-'} and #formals = 1 then
8959             pass ;                                 -- Unary operators.
8960         elseif
8961             (if proc_id ∈ {'not', 'abs'} then #formals = 1      --
8962                                          else #formals = 2 end)
8963           then pass ;
8964         else
```

```
8965        errmsg('Incorrect no. of arguments for operator ',
8966                                        proc_id, '6.7') ;
8967        end if ;
8968
8969        if exists [-,-,-,initv] ∈ formals | initv /= [ ] then
8970            errmsg('Initializations not allowed ', ' for operators',
8971                                                '6.7') ;
8972         elseif proc_id = '/=' then
8973            errmsg(' "/=" cannot be given an explicit definition',
8974                                                '', '6.7') ;
8975         end if ;
8976    end if ;
8977
8978    return chain_overloads(proc_id, prog_info) ;                --
8979
8980 end proc subprog_spec;
```

subprog_body

```
8982 proc subprog_body(node) ;
8983
8984
8985
8986 [-, specs_node, decl_node, stats_node, except_node] := AST(node) ;
8987
8988    [-, id_node, -, -] := AST(specs_node) ;
8989    id := adasem(id_node) ;
8990
8991    if is_comp_unit then
8992
8993       new_compunit('subprog', id) ;
8994    -- If the specification of the unit was itself a compilation unit, we
8995    -- will verify that the two specs are conforming. If this is the
8996    -- body to a generic comp. unit, will have to access and update the
8997    -- spec. In both cases see if the spec. is available.
8998
8999       spec_name := ['subprog spec', id];    -- Already retrieved
9000       if UNIT_DECL(spec_name) /= om then
9001
9002          -- Unpack declarations and install symbol table of unit.
9003          [unname, s_info, decmap] := UNIT_DECL(spec_name); --
9004 --
9005          declared('STANDARD#0')(id) := unname;  -- Install unit name
9006          declared(unname) := decmap ;
9007
9008          (for [unam, entry] ∈ s_info)
9009             SYMBTABF(unam) := entry;                --
9010          end for;
9011 --
```

```
9012        end if;
9013    end if;
9014    LAB_INIT() ;
9015
9016    if (prog_name := check_old(id)) /= '?' and
9017    is_identifier(prog_name) and nature(prog_name) ∈
9018        {'generic_procedure_spec','generic_function_spec'}
9019    then
9020        return generic_subprog_body(prog_name, node) ;
9021    else
9022        -- (Re)process subprogram specification.
9023        prog_name := subprog_spec(adasem(specs_node)) ;          --
9024        if is_comp_unit and
9025       # [n ∈ (overloads(prog_name) ? {})| nature(n) /= 'literal'] > 1
9026       then
9027            -- Spec. does not match its previous occurrence, or several
9028            -- subprograms with same name are present.
9029            errmsg('library subprograms cannot be overloaded', '',
9030                                                    '10.1(10)') ;
9031            return [ ] ;
9032        end if ;
9033    end if ;
9034
9035    newscope(prog_name) ;
9036    proc_stack with:= prog_name ;
9037    prefix +:= id + '.' ;              -- For names of local objects.
9038    body :=
9039      block(adasem(decl_node), adasem(stats_node), adasem(except_node));
9040
9041    lab_end() ;                  -- Validate goto statements in subprogam.
9042    popscope() ;                 -- And exit.
9043    save_subprog_info ;
9044
9045    aix := [[nature(prog_name), prog_name,
9046                                  build_signature(prog_name), body]];
9047
9048 --
9049    -- If the subprogram is named '=', we emit as well the body for the
9050    -- corresponding definition of /= . The specification for it has
9051    -- been introduced in the same scope.
9052
9053    if original_name(prog_name) = '=' and
9054    #(sig := signature(prog_name))= 2
9055    then
9056        assert exists nam ∈ overloads(declared(scope_name)('/=')) |
9057                                  same_signature(signature(nam), sig) ;
9058
9059        [[x1,-], [x2,-]] := sig ;
9060        aix with:= ['function', nam, sig,
```

```
9061          [ [ ],
9062            [['return', nam, 0, ['not',['call',prog_name, [x1, x2]]]]],
9063          [ ] ]] ;
9064
9065     end if ;
9066 --
9067     return  aix ;
9068
9069 end proc subprog_body;
```

build_signature

```
9071 proc build_signature(prog_name) ;
9072
9073 -- The default values for in parameters are not needed in the interpreter
9074 -- because they are inserted where needed in calls. We only keep names,
9075 -- modes and types in the signature.
9076
9077 sig := signature(prog_name) ;
9078 if not is_tuple(sig) or not is_tuple(sig(1)) then
9079     return [ ] ;                          -- error in previous declaration
9080 else
9081     return [[n,m,t] : [n,m,t,-] ∈ sig] ;
9082 end if;
9083
9084 end proc build_signature;
```

get_formals

```
9086 proc get_formals(form_list) ;
9087
9088 formals := [ ] ;
9089 decl_tree := [ ] ;
9090
9091 (for f_spec ∈ form_list)
9092     [id_list, fmode, type_id, opt_init] := f_spec ;
9093
9094     -- A missing mode indication corresponding to an 'in' parameter.
9095     if fmode = '' then fmode := 'in' ; end if ;
9096
9097     type_mark := find_old(type_id) ;
9098     if not is_type(type_mark) then
9099         errmsg('expect type mark for formal parameter','','6.1') ;
9100       type_mark := 'any' ;
9101     end if ;
9102
9103     init_val  := if opt_init = [ ] then [ ]
9104                  else check_type(type_mark, opt_init) end ;
9105
```

```
9106    formals +:= [[id, fmode, type_mark, init_val]: id ∈ id_list] ;
9107
9108    init_norm := normalize(type_mark, opt_init) ;    -- sans folding
9109    decl_tree with:= f_spec + [type_mark, init_norm] ;
9110 end for ;
9111
9112 -- Save original tree to verify conformance in case of redeclaration.
9113 if #formals > 0 then formal_decl_tree(formals) := decl_tree; end if ;
9114
9115 return formals ;
9116 end proc get_formals ;
```

process_formals

```
9118 proc process_formals(scope, form_list) ;
9119
9120
9121
9122 -- This  is called to process  formal parameters of a procedure spec. or
9123 -- entry spec.
9124
9125 new_form_list := [ ] ;
9126 -- Initialize -declared- map for new scope.
9127  declared(scope) ?:= {} ;
9128
9129 (forall t ∈ form_list)
9130    [form_id, in_out, type_mark, opt_init] := t ;
9131
9132    if (form_name := declared(scope)(form_id)) = om then
9133         form_name := prefix + form_id + suffix + str newat ;
9134         declared(scope)(form_id) := form_name ;
9135         scope_of(form_name) := scope ;
9136 --
9137         -- formal parameters cannot have a incomplete type. They can
9138         -- have an incomplete private type however.
9139        if type_of(type_mark) = 'incomplete' then
9140            errmsg('invalid use of incomplete_type ',
9141                                  original_name(type_mark), '3.8.1') ;
9142         end if ;
9143
9144    else
9145         -- This id was already seen in the formals list.
9146         errmsg('Duplicate formal parameter: ', form_id, '6.1') ;
9147    end if ;
9148
9149 --
9150    -- Assignable parameters must not appear in functions.
9151    if in_out /= 'in' and nature(scope) = 'function' then
9152         errmsg(in_out, ' parameter not allowed for functions','6.5') ;
```

```
9153        return [ ] ;
9154    end if ;
9155
9156    if in_out = 'out' then
9157        if is_access(type_mark) then pass ;
9158
9159        elseif type_of(type_mark) = 'limited private'
9160            and scope_name = scope_of(type_mark) then
9161              -- We are in the visible part of the package that declares
9162              -- the type. Its full decl. will have to be given with an
9163            -- assignable type.
9164              misc_type_attributes(type_mark) :=
9165                      (misc_type_attributes(type_mark) ? {}) with 'out' ;
9166 --
9167          elseif is_limited_type(type_mark) then
9168              errmsg('Invalid use of limited type for out parameter',",
9169                                                          '7.4.4') ;
9170          end if ;
9171    end if ;
9172
9173    NATURE(form_name)     := in_out ;
9174    TYPE_OF(form_name)    := type_mark ;
9175 --
9176
9177    if opt_init /= [ ] and in_out /= 'in' then
9178            errmsg('default initialization only allowed for ',
9179                                    'IN parameters', '6.1') ;
9180        opt_init := [ ] ;
9181    end if ;
9182
9183    TO_XREF(form_name) ;
9184    new_form_list with:= [form_name, in_out, type_mark, opt_init] ;
9185 end forall ;
9186
9187 if #form_list > 0 then
9188    -- Preserve the declaration tree for conformance checks later on.
9189    formal_decl_tree(new_form_list) := formal_decl_tree(form_list);
9190 end if ;
9191
9192 return new_form_list ;
9193
9194 end proc process_formals ;
```

reprocess_formals

```
9196 proc reprocess_formals(scope, sig) ;
9197
9198
9199 if #sig = 0 then return ; end if ;
```

```
9200 old_sig  := signature(scope) ;
9201 old_decl := formal_decl_tree(old_sig) ;
9202 new_decl := formal_decl_tree(sig) ;
9203
9204
9205 if #old_decl /= #new_decl then
9206     conformance_error ;
9207     return ;
9208 end if  ;
9209
9210 (forall i ∈ [1..#old_decl])
9211     [id_list, fmode, type_id, opt_init, type_mark, init_val] :=
9212                                                     new_decl(i) ;
9213     [oid_list, ofmode, otype_id, oopt_init, otype_mark, oinit_val] :=
9214                                                     old_decl(i) ;
9215
9216     if id_list /= oid_list       or fmode /= ofmode
9217       or type_mark /= otype_mark or not conform(type_id, otype_id)
9218       or init_val /= oinit_val   or not conform(opt_init, oopt_init)
9219     then
9220         conformance_error ;
9221       return ;                        -- No need for more than one
9222     end if ;
9223
9224 end forall ;
9225
9226 end proc reprocess_formals ;
```

normalize

```
9228 proc normalize(typ, opt_init) ;
9229
9230 -- transform the default expression of  a discriminant or  a formal para-
9231 -- meter, into a standard  form for conformance checking. Name resolution
9232 -- by itself is not sufficient, because of the possible presence of over-
9233 -- loaded operators. Full type-checking with constant-folding is too much
9234 -- because two  identical-looking expressions may be non-conforming. (see
9235 -- ACV test  c38105ab for examples).  We therefore resolve the expression
9236 -- but inhibit constant folding by means of the global flag fold_context.
9237 -- Finally,  the expression  may have  a constraint imposed  on it in one
9238 -- place and  not in the other,  because of  additional information  on a
9239 -- private type, e.g. The constraint is removed for conformance checking.
9240
9241 if opt_init = [ ] then return [ ] ;
9242 else
9243     fold_context := false ;
9244     v := check_type(typ, opt_init) ;
9245     fold_context := true ;
9246     return if is_tuple(v) and v(1) ∈ qualifiers then v(3) else v end ;
```

```
9247 end if ;
9248
9249 end proc normalize ;
```

conform

```
9251 proc conform(t1, t2) ;
9252
9253 -- Check  conformance  of two expressions.  The expressions have been
9254 -- resolved but not  constant folded. In some cases involving private
9255 -- types, one of the expressions may include a constraint check while
9256 -- the other doesn't; in that case remove the constraint before veri-
9257 -- fying conformance.
9258
9259 if is_identifier(t1) then
9260  return
9261    if is_identifier(t2) then t1=t2
9262    elseif is_tuple(t2) then
9263       (t2(1) = '.'                              -- expanded name
9264       and top(t2) = t1                          -- with the same selector
9265         and t1   op_designators                 -- which is a simple name
9266         and (#t1 /= 3                                   -- and not an operator
9267          or t1(1) /= "'" or t1(3) /= "'"))  -- or character literal
9268
9269       or                                        -- constrained expression
9270         (t2(1) ∈ qualifiers and conform(t1, t2(3)))
9271    else false
9272  end ;
9273
9274 elseif is_tuple(t1) then
9275  return
9276    if is_identifier(t2) then conform(t2, t1)
9277
9278    elseif is_tuple(t2) then
9279
9280         if t1(1) = 'int_literal' then
9281             (t2(1) = 'int_literal' and
9282              ADAVAL('universal_integer', t1(2)) =
9283                                  ADAVAL('universal_integer', t2(2)))
9284
9285       . elseif t1(1) = 'real_literal' then
9286             (t2(1) = 'real_literal' and
9287              ADAVAL('universal_real', t1(2)) =
9288                                  ADAVAL('universal_real', t2(2)))
9289
9290       elseif t1(1) ∈ qualifiers and t2(1)   qualifiers then
9291            conform(t1(3), t2)
9292
9293       else
```

```
9294          #t1 = #t2 and (forall i ∈ [1..#t1] | conform (t1(i), t2(i)))
9295        end
9296
9297     else false
9298   end ;
9299
9300 elseif is_atom(t1) then
9301     -- a node in the tree appears only to place an error message. Clearly
9302     -- different nodes will appear at congruent places.
9303     return is_atom(t2) ;
9304
9305 elseif is_overloaded(t1) then
9306     -- overloaded name: ok if both are overloaded. The visibility may be
9307     -- different, and the overload sets don't have to match.
9308     return is_overloaded(t2) ;
9309
9310 else                                              -- numeric quantity
9311     return t1=t2 ;
9312
9313 end if ;
9314
9315 end proc conform ;
```

statement_name

```
9317 proc statement_name(name) ;
9318
9319 -- This procedure resolves statements which consist of a single name.
9320 -- These statements can have one of the following meanings :
9321 -- a) Procedure call.
9322 -- b) entry call .
9323 -- c) Code insertions.
9324
9325 -- This last case is ignored in the current implementation.
9326 --
9327 -- Procedure and entry calls are handled by first resolving the name,
9328 -- and then type-checking the argument list. Complications arise
9329 -- for parameterless procedures and entries, and for parameterless
9330 -- entries in entry families. In those cases, this procedure reformats
9331 -- the name by appending an empty argument list.
9332
9333 if is_identifier(name)          -- Parameterless procedure or entry.
9334 or name(1) = '.' then           -- Qualified name of entry or proc.
9335     return proc_or_entry(['call?', name, [ ]] ) ;
9336 elseif name(1) = 'call?' then
9337     return proc_or_entry(name) ;
9338 else
9339     errmsg('Unsuported feature: not procedure or entry call','','5.1') ;
9340     return [ ] ;
```

```
9341 end if ;
9342
9343 end proc statement_name ;
9344
```

proc_or_entry

```
9346 proc proc_or_entry(name) ;
9347
9348
9349
9350 -- Process procedure calls, entry calls, and calls to members of
9351 -- entry families.
9352 -- The statement :          name(args) ;
9353 -- can have 3 meanings :
9354 -- a) It can be a procedure call.
9355 -- b) It can be an entry call.
9356 -- c) -name- can be the name of an entry family, and -args- an index
9357 -- into that family. This is recognized by the fact that the type of
9358 -- -name- is an array type.
9359 -- In the first two cases, we must type-check and format the argument
9360 -- list. In the last one, we must emit a parameterless entry call.
9361 --
9362 -- If the statement has the format :    name(arg)(args) ;
9363 --
9364 -- then it can only be a call  with parameters to an element of an
9365 -- entry family.
9366
9367 [-, obj, arg_list] := name ;
9368
9369 if is_identifier(obj) or obj(1) = '.' then
9370     obj_name := find_old(obj) ;
9371
9372     if is_tuple obj_name and obj_name(1) = '.' then
9373         return entry_call(obj_name, arg_list) ;
9374
9375     elseif is_overloaded(obj_name) then
9376         -- Type-checking of the argument list must be done as usual,
9377         -- but there is no required type on the result of the call.
9378         call := check_type('none', [obj_name] + arg_list) ;
9379
9380         if call = [ ] then return [ ] ;  -- Error emitted already.
9381         elseif not is_tuple(call)
9382         or call(1)   {'call', 'ecall'}
9383           or call(1) ='call' and nature(call(2))
9384                                 {'procedure','procedure_spec'}
9385           then errmsg('Invalid procedure or entry call',
9386                                       '','6.5, 9.5') ;
9387           return [ ] ;
```

```
9388          else return call ;
9389          end if ;
9390
9391
9392      else
9393          -- If the name was undeclared, an error message was emitted
9394          -- already. We can detect this case by the fact that the identifier
9395          -- has type -any-.
9396          if type_of(obj_name) /= 'any' then
9397              errmsg('Invalid statement',",'5.1') ;
9398          else
9399              -- Make up a dummy symbol table entry,so that subsequent uses
9400              -- of it have a chance of looking plausible.
9401              nature(obj_name) := 'procedure' ;
9402              signature(obj_name) := #arg_list * [[", 'in', 'any', [ ]]] ;
9403              type_of(obj_name) := 'none';
9404            overloads(obj_name) := {obj_name} ;
9405          end if ;
9406          return [ ] ;
9407      end if;
9408  else
9409      -- Case of an entry family call with parameters.
9410      obj_name := find_old(obj) ;
9411      if obj_name = [ ] or obj_name(1) /= '[ ]' then
9412          errmsg('Invalid call', ",'9.5') ;
9413          return [ ] ;
9414      else
9415          return entry_call(obj_name, arg_list) ;
9416      end if;
9417  end if ;
9418
9419  end proc proc_or_entry ;
9420
```

desig_to_name

```
9422  proc desig_to_name(desig) ;
9423
9424  -- This procedure is invoked at places where an operator designator
9425  -- can appear. If desig is a name, it is returned as is. If it is a
9426  -- string, it can only be one of the predefined operators in the
9427  -- language, excluding those that cannot be overloaded.
9428
9429
9430  const to_lower = { ['A', 'a'], ['D', 'd'], ['E', 'e'], ['M', 'm'],
9431          ['N', 'n'], ['O', 'o'], ['R', 'r'], ['T', 't'], ['X', 'x']} ;
9432
9433
9434
```

```
9435
9436 if desig(1) /= '"' then return desig ;
9437 else
9438     op_name := +/[to_lower(c) ? c : c ∈ desig(2..#desig-1)] ;
9439     if op_name ∈ op_designators then
9440         return op_name ;
9441     else
9442         errmsg(desig, ' is not an operator designator', '4.5') ;
9443         return desig ;
9444     end if ;
9445 end if ;
9446
9447 end proc desig_to_name ;
9448
```

chain_overloads

```
9450 proc chain_overloads(id, new_specs) ;
9451
9452
9453
9454 -- Insert procedure, function, or enumeration literal into the current
9455 -- symbol table. Because these names can be overloaded, each set of
9456 -- overloaded names visible in the current scope is held in the
9457 -- -overload- attribute of the corresponding identifier.
9458 -- If there is no actual overload, the unique name is generated as for
9459 -- any other identifier. Otherwise, successive overloads in the same
9460 -- scope are given an additional arbitrary suffix to distinguish them
9461 -- one from the other.
9462
9463 -- The overloaded name in inserted in the current scope, except in
9464 -- the case procedures in private parts of packages, which are just part
9465 -- of the package declarations (but of course not of the visible part).
9466
9467
9468 certain := exists scope ∈ open_scopes | not is_record(scope) and
9469                                 nature(scope) /= 'private_part' ;
9470
9471 new_name := prefix + id + suffix ;
9472 [new_nat, new_typ, new_sig, parent_subp] := new_specs ;          --
9473
9474
9475 if (seen := declared(scope)(id)) = om then
9476     -- First occurrence in this scope. Define therein.
9477
9478     declared(scope)(id) := new_name ;
9479     new_over_spec(new_name, new_specs, scope) ;
9480     overloads(new_name) := {new_name} ;
9481 else
```

```
9482    -- Name already appears in current scope. One of the following
9483    -- may be the case :
9484    -- a) It is a redeclaration, either because a non-overloaded
9485    -- instance of that id exists, or because an object with the
9486    -- same signature has already been declared : indicate error.
9487    -- b) It is the body of a function or procedure, whose specs
9488    -- have already been seen. Update the corresponding entry.
9489    -- c) It is a new object. Generate a new name, and make entry
9490    -- for it.
9491    -- d) It is a redeclaration of a derived subprogram. in that case
9492    -- the derived subprogram becomes inaccessible.
9493    -- e) If it is a derived subprogram, and there is an explicit user
9494    -- defined one already, the derived one is discarded.
9495
9496    if not can_overload(seen) then
9497        errmsg('Redeclaration of identifier: ',id,'8.3, 8.4') ;
9498        return seen ;
9499    else
9500       current_overload :=  overloads(seen) ;
9501    end if ;
9502
9503    (forall name ∈ current_overload)
9504        if  same_signature(signature(name), new_sig) and
9505            same_type(type_of(name), new_typ)
9506          then
9507            if nature(name) = new_nat and new_nat /= 'aggregate'
9508            or new_nat ∈ {'procedure_spec', 'function_spec'}
9509            or parent_subp /= om                    -- derived subprogram
9510                then
9511 --
9512                if is_derived_subprogram(name) then
9513                    -- An explicit declaration redefines an implicitly
9514                    -- derived subprogram. Make the later unreachable.
9515                    overloads(seen) less:= name ;
9516                    ALIAS(name) := om ;
9517                elseif parent_subp /= om then
9518                    if new_nat = 'literal' then
9519                        -- A derived literal is hidden by any other decl.
9520                        -- but still exists as a literal of the type.
9521                        -- Insert in symbol table but not in declared.
9522
9523                        new_name := named_atom(id) ;
9524                        new_over_spec(new_name, new_specs, scope) ;
9525                        overloads(new_name) := {new_name} ;
9526                        return new_name ;
9527                    else
9528                      -- Implicitly derived subprogram, in the presence
9529                      -- of an explicit one, which hides it forever.
9530                      return '' ;
```

```
9531                    end if ;
9532                 else
9533                    errmsg('Redeclaration of subprogram: ', id,'6.6');
9534                    return name ;
9535                 end if ;
9536 --
9537              elseif
9538                 nature(name) ∈ {'procedure_spec', 'function_spec'} then
9539                    -- case of a body whose spec was already seen.
9540                    nature(name) := new_nat ;
9541                    -- Refresh the symbol table for the body, but do
9542                    -- not emit again the constant initializations.
9543                    reprocess_formals(name, new_sig);
9544                    return name ;
9545              end if ;
9546
9547           end if;
9548     end forall  ;
9549
9550     -- If we fall through, this is a new entry.
9551
9552     new_name := named_atom(id) ;
9553     declared(scope)(id) := new_name ;
9554     new_over_spec(new_name, new_specs,scope) ;
9555     overloads(new_name) := overloads(seen) with new_name ;
9556
9557     -- At this point, the unique name  -seen- cannot be retrieved from
9558     -- SYMBTAB by way of the declared map, because declared(scope)(id)
9559     -- points to  the new name.  In order to insure that  the range of
9560     -- declared coincides with the domain of SYMBTAB,we add the following
9561     -- entry to declared:
9562
9563     declared(scope)(new_name) := seen;
9564
9565     -- This simplifies the saving of scopes in separate compilations.
9566
9567 end if ;
9568
9569 return new_name ;
9570
9571 end proc chain_overloads;
9572
9573 proc can_overload(name) ;
9574
9575 return
9576     NATURE(name)?" ∈ {'procedure_spec', 'function_spec', 'op',
9577                              'function', 'procedure', 'entry', 'literal'} ;
9578 end proc can_overload ;
9579
```

new_over_spec

```
9581 proc new_over_spec(name, specs, scope) ;
9582
9583 -- Place in symbol table maps the specification of a new overloadable
9584 -- object .
9585
9586 [nat, typ, sig, parent_subp] := specs ;
9587
9588 -- Apply the special checks on redefinitions of equality.
9589 if original_name(name) = '=' then
9590    [-,-,typ1] := sig(1) ;                -- type of formal
9591    if # sig /= 2
9592    or typ1 /= sig(2)(3)
9593    or typ /= 'BOOLEAN' then
9594       errmsg('Invalid argument profile for "=" ',",'6.7') ;
9595
9596    -- '=' can only be redefined for limited types, except if it
9597    -- is a renaming or a derived subprogram.
9598
9599    elseif not is_limited_type(typ1) and parent_subp = om then
9600       errmsg('= can only be defined for limited types',",'6.7');
9601 --
9602    elseif parent_subp /= om and original_name(parent_subp) /= '=' then
9603       errmsg('renaming with = can only rename an equality operator',
9604                                                  ", '6.7') ;
9605    end if ;
9606 --
9607    -- If it is not a derivation or renaming, introduce the implicit
9608    -- "/=" as well.
9609    if parent_subp = om then
9610       chain_overloads('/=', ['function', typ, sig]) ;
9611    end if ;
9612 end if ;
9613
9614 NATURE(name)    := nat ;
9615 TYPE_OF(name)   := typ ;
9616 SIGNATURE(name) :=
9617    if nat = 'literal' then [ ] else process_formals(name,sig) end ;
9618 SCOPE_OF(name)  := scope ;
9619 TO_XREF(name) ;
9620
9621 end proc new_over_spec;
9622
```

same_signature

```
9624 proc same_signature(old, new) ;
9625
9626 -- Compare the signatures of two procedures to determine whether
9627 -- they hide each other. Two signatures are considered identical if
9628 -- they have the same length, and the formals match in name and type.
9629
9630 if old = new then return true ;
9631 elseif old = om or new = om
9632     or not is_tuple(old) or not is_tuple(new) then return false ;
9633 elseif #old /= #new then return false ;
9634 end if ;
9635
9636 (forall i ∈ [1..#old])
9637
9638     [-, -, type1, -] := old(i) ;
9639     [-, -, type2, -] := new(i) ;
9640
9641     if not same_type(type1,type2) then return false ;
9642     else pass ;
9643     end if ;
9644 end forall ;
9645
9646 return true ;
9647
9648 end proc same_signature ;
9649
```

same_type

```
9651 proc same_type(type1,type2) ;
9652
9653
9654 return base_type(type1) = base_type(type2) ;
9655
9656 end proc same_type ;
```

7. Packages

package_head

```
9659 proc package_head(id);
9660
9661 -- Process a non-generic package identifier: install scope, initialize
9662 -- mappings.
9663
9664 new_compunit('spec', id);
9665
9666 newmod(id) ;
9667
9668 nature(scope_name)  := 'package_spec' ;
9669 type_of(scope_name) := 'none' ;
9670 -- Create dummy entry to hold use clauses, which are declarative items.
9671 find_new('$used') ;
9672 use_declarations(scope_name) := [ ] ;
9673 private_decls(scope_name) := {} ;
9674
9675 return [kind,scope_name];
9676
9677 end proc package_head;
9678
```

module_body_id

```
9680 proc module_body_id(mod_nature, name) ;
9681
9682
9683
9684 -- This procedure is invoked when the name of a module body has been
9685 -- seen. It opens the new scope, and if necessary retrieves from the
9686 -- library the specifications for the module.
9687
9688 if is_comp_unit and not separate then
9689     get_specs(name) ;
9690 -- When compiling a module body, the contents of the
9691 -- module specs are part of the DECLARED map for the body itself.
9692 -- Initialize it here.
9693     declared(scope_name)(name) := name;          -- Insert in STANDARD
9694     declared(name) := visible(name);
9695 end if;
9696
```

```
9697 new_compunit('body', name);
9698
9699 mod_name := find_old(name);
9700 if not is_comp_unit and scope_of(mod_name) /= scope_name then
9701     errmsg('specification and body are in different scopes' ,
9702                                   '' , '7.1, 9.1') ;
9703 end if ;
9704
9705 -- Nature of specification must match that of current body
9706
9707 new_nature := old_nature := nature(mod_name);
9708 if rmatch(new_nature, '_spec') /= om and   -- Strip off '_spec'
9709     (match(old_nature, mod_nature) /= om
9710     or match(old_nature, 'generic_' + mod_nature) /= om)
9711 then
9712     nature(mod_name) := new_nature;
9713 else
9714    errmsg('Matching specification not found for body ',name,'7.1, 9.1');
9715     private_decls(mod_name) := {} ;  -- For safe processing of body.
9716     declared(mod_name) ?:= {} ;
9717 end if;
9718
9719 if mod_nature = 'task' then
9720     -- The list of entries in a task spec is saved in the signature.
9721     declared(mod_name) := signature(mod_name);
9722     -- if this is a task object, remove -spec marker from its anonymous
9723     -- task type.
9724     if new_nature = 'task_obj' then
9725         nature(type_of(mod_name)) := 'task_type' ;
9726     end if ;
9727
9728 --
9729     -- Initialize information to construct relay set for task.
9730     proc_stack with:= mod_name ;
9731
9732 elseif new_nature ∈ {'package', 'generic_package'} then
9733
9734     -- Within a package body, declarations from the private part of the
9735     -- specification are  visible. Swap  visible and  private versions.
9736
9737     (forall [item, pdecl] ∈ private_decls(mod_name))
9738        [SYMBTABF(item), private_decls(mod_name)(item)] :=
9739                                    [pdecl, SYMBTABF(item)];   --
9740     end forall;
9741     -- Furthermore, composite types that depend on (outer) private types
9742     -- may now be fully useable if the latter received full declarations,
9743     -- (as long as they do not depend on external private types...)
9744
9745     (forall [-, t] ∈ declared(mod_name) | is_type(t)
```

```
9746        and exists mark ∈ incp_types
9747                                | mark ∈  (misc_type_attributes(t) ? {}))
9748
9749          real_t := if is_access(t) then designated_type(t) else t end ;
9750
9751        if not is_private(real_t) then                    --
9752           -- full declaration  of private ancestor(s) has been seen.
9753             -- save visible declaration before updating.
9754             private_decls(mod_name)(t) := SYMBTABF(t) ;
9755             misc_type_attributes(t) less:= mark ;
9756        end if ;
9757
9758    end forall ;
9759 --
9760    -- and install the use clauses that were encountered in the specifi-
9761    -- cation.
9762    used_mods +:= use_declarations(mod_name);
9763 --
9764 -- Else the body was not found. Error was emitted already.
9765 end if;
9766
9767 newscope(mod_name) ;
9768 prefix := prefix + name + '.';
9769 -- Within the body of a task type, the name of the task can be used to
9770 -- designate the task currently executing the body. We create an alias
9771 -- to be elaborated at run-time, under the name 'current_task'.
9772 if nature(mod_name) = 'task_type' then
9773    c := find_new('current_task') ;
9774    type_of(c) := mod_name ;
9775    nature(c) := 'obj' ;
9776 end if ;
9777 -- Initialize the stacks used for label processing.
9778
9779 LAB_INIT() ;
9780
9781 return [mod_nature, mod_name] ;
9782
9783 end proc  ;
9784
```

module_body

```
9786 proc module_body(nat, blk) ;
9787
9788
9789
9790     mod_name := scope_name ;
9791    [decls, stats, except] := blk ;
9792    if nature(mod_name) = 'task_type' then
```

```
9793        decls := [['current_task', find_old('current_task')] ] + decls ;
9794    end if ;
9795
9796    lab_end() ;
9797    popscope() ;
9798    blk := [decls, stats, except];                          --
9799
9800    -- Having finished the module body, we now restore the visible
9801    -- declarations saved in module_body_id (If it is a package).
9802
9803    if nat(1) ∈ {'package', 'generic_package'} then
9804       (forall [item, pdecl] ∈ (private_decls(mod_name)) )
9805          [SYMBTABF(item), private_decls(mod_name)(item)] :=
9806                   [pdecl, SYMBTABF(item)];
9807       end forall;
9808    end if ;
9809
9810    if nature(mod_name) = 'generic_package' then
9811
9812
9813       -- We must update the declarations for the current unit, to
9814       -- include the generic body. This can be done omly if the
9815       -- generic specification appears in the current compilation,
9816       -- which is a restriction on the current implementation that
9817       -- will be lifted some day.
9818       -- For purposes of generic instantiation, we must save not only
9819       -- the visible part of the package, but all declarations in the
9820       -- body as well, including declarations for nested non-generic
9821       -- objects. This parallels what is done at the point of instan-
9822       -- tiation. We also save a list of all local identifiers, which
9823       -- will have to be renamed during the instantiation process.
9824       locals := {} ;
9825
9826       if unit_name(1) ∈ {'body','subprog'} and #unit_name = 2 then
9827          spec_name := ['spec', unit_name(2)] ;
9828          if LIB_UNIT(spec_name) = AISFILE and
9829                      #(UNIT_DECL(spec_name) ? [ ]) > 0 then
9830             -- Unpack unit specification.
9831             [unit_unam, specs, decmap, old_vis, notvis]
9832                                        := UNIT_DECL(spec_name) ;
9833
9834             scopes := {mod_name} ;
9835
9836             -- Update the specs of generic types, that may carry the
9837             -- marker "-constrain', because of usage in body.
9838             (forall [-, t_name] ∈ declared(mod_name)
9839                                   | is_generic_type(t_name))
9840                specs(t_name) := SYMBTABF(t_name) ;
9841             end ;
```

```
9842                (while scopes /= {})
9843                   scope from scopes ;
9844
9845                   specs(scope)  := SYMBTABF(scope) ;
9846                   decmap(scope) := declared(scope) ;
9847                  body_decls    := declared(scope) -
9848                                              (visible(scope) ? {});
9849                   notvis(scope) := body_decls ;
9850
9851                   (forall [-,u_name] ∈ body_decls)
9852                       specs(u_name) := SYMBTABF(u_name) ;
9853
9854                       if scope_of(u_name) = scope then
9855                          -- Local declaration, and not a renaming.
9856                          locals with:= u_name ;
9857                       end if ;
9858
9859                       if declared(u_name) /= om
9860                       and not can_overload(u_name)
9861                       and nature(u_name) /= 'generic_package'
9862                       then    -- Contains further collectible decls.
9863                          scopes with:= u_name ;
9864                       end if ;
9865                   end forall ;
9866
9867                end while ;
9868                -- The body and locals are part of the generic signature.
9869          signature(mod_name) +:= [blk, locals] ;
9870                specs(mod_name) := SYMBTABF(mod_name) ;
9871
9872                -- Repackage the unit's information.
9873
9874                UNIT_DECL(spec_name) :=
9875                 [unit_unam, specs, decmap, old_vis, notvis];  --
9876 --
9877             elseif is_comp_unit then
9878                -- Repackage as a specification.
9879          signature(mod_name) +:= [blk, locals] ;
9880                newscope(mod_name) ;                          -- For end_specs
9881                end_specs(['generic_package', mod_name], {}) ;
9882 --
9883           end if ;
9884          end if;
9885        signature(mod_name) +:= [blk, locals] ; -- In any case.
9886
9887          return [ ] ;
9888
9889     end if;
9890
```

```
9891    if nat(1) = 'task' then
9892        return [['task_body', TYPE_OF(mod_name), blk ]];
9893    else
9894        save_body_info(mod_name) ;
9895        return [['package_body', mod_name, blk]] ;          --
9896    end if;
9897
9898 end proc module_body;
9899
```

private_decl

```
9901 proc private_decl(id, opt_discr, priv_kind) ;
9902
9903
9904
9905 if declared(scope_name)(id) = om then
9906    name := find_new(id) ;
9907    type_of(name) := priv_kind ;
9908    root_type(name) := name ;                                --
9909    process_discr(name, opt_discr) ;
9910    Nature(name) := 'type' ;
9911    misc_type_attributes(name) := {} ;                       --
9912    popscope() ;
9913
9914    if nature(scope_name)
9915      {'package_spec', 'generic_package_spec', 'generic_part' } then
9916        errmsg('Invalid context for private declaration', '',
9917                                                    '7.4, 12.1.2') ;
9918    end if ;
9919 else
9920    errmsg('Invalid redeclaration of ', id, '8.2');
9921    name := '' ;
9922 end if ;
9923 return name ;
9924
9925 end proc private_decl ;
9926
```

fully_declared

```
9928 proc check_fully_declared(type_mark) ;
9929
9930 -- Called from object and constant declarations, to ensure that a
9931 -- private or incomplete type is not used in a declaration before its
9932 -- full declaration has been seen.
9933
9934 if is_incomplete_type(type_mark) then
9935    errmsg('invalid use of type ' + original_name(type_mark),
```

```
9936          ' before its full declaration','3.8.1, 7.4.1') ;
9937 end if ;
9938
9939 -- If the type is a generic private type, and is used as an unconstrained
9940 -- subtype indication, note that its instantiations will have to be with
9941 -- a constrained type.
9942
9943 check_generic_usage(type_mark) ;
9944
9945 end proc check_fully_declared ;
```

check_fully_declared2

```
9947 proc check_fully_declared2(type_mark) ;
9948
9949 -- Called from array element and component declarations, to ensure that
9950 -- an incomplete type is not used before its full declaration is seen.
9951
9952 check_incomplete(type_mark) ;
9953 check_generic_usage(type_mark) ;
9954
9955 end proc check_fully_declared2 ;
9956
```

is_private

```
9958 proc is_private(type_mark) ;
9959
9960 -- Determine whether a type has a private subcomponent. This differs
9961 -- from what is done in private_ancestor, where only incomplete priv.
9962 -- subcomponents are of interest.
9963
9964 return type_of(base_type(type_mark)) ∈ priv_types
9965 or     type_of(root_type(type_mark)) ∈ priv_types
9966
9967 or     is_array(type_mark)
9968 and is_private(component_type(type_mark))
9969
9970 or     is_record(type_mark)
9971 and exists [-, comp] ∈ declared_components(base_type(type_mark))
9972                        | is_private(type_of(comp)) ;
9973 end proc is_private ;
```

is_limited_type

```
9975 proc is_limited_type(type_mark) ;
9976
9977 -- A type is limited if its root type is a limited private type or a task
9978 -- type, or if it is a composite type some of whose components are limit-
```

```
9979 -- ted. The attributes 'limited' and 'limited private' are attached to
9980 -- such composite types when they are created by a definition, derivation
9981 -- or subtype declaration.
9982 -- These attributes are also attached to an access type, to enforce the
9983 -- restrictions of 7.4.2(8) but access types are not limited in any case.
9984 --
9985 return
9986     type_of(base_type(type_mark)) = 'limited private'
9987     or
9988     type_of(root_type(type_mark)) = 'limited private'
9989     or is_task_type(type_mark)
9990
9991     or 'limited' ∈ (mt := (misc_type_attributes(type_mark) ? {}))
9992         and not is_access(type_mark)                          --
9993     or 'limited private' ∈ mt
9994 --
9995       and ( (scope_of(type_mark)   open_scopes
9996                                   and not is_access(type_mark))        --
9997             or
9998             is_array(type_mark)
9999                   and  is_limited_type(component_type(type_mark))
10000            or
10001            is_record(type_mark) and exists [-,comp]
10002                         ∈ declared_components(base_type(type_mark))
10003                         | is_limited_type(type_of(comp)) ) ;
10004--
10005end proc is_limited_type ;
```

private_kind

```
10007proc private_kind(type_mark) ;
10008-- We must distinguish between fully limited types, such as task types,
10009-- and limited private types, which are not limited in the defining
10010-- package body. Limited private types become limited when used outside
10011-- of their scope of definition, and so do composite types with such
10012-- components, or derived types of them. This procedure is used to set
10013-- the corresponding attribute in a type definition.
10014--  Generic limited types and composites of them are always limited.
10015
10016r := root_type(type_mark) ;
10017kind :=
10018    if is_scalar_type(type_mark) then om
10019
10020    elseif (t := type_of(r)) ∈ priv_types
10021    then t
10022
10023    elseif exists attr ∈ incp_types |
10024                        attr ∈ (misc_type_attributes(type_mark) ?{})
10025    then attr
```

```
10026
10027    elseif is_task_type(type_mark) then 'limited'
10028
10029    elseif is_access(type_mark)  and
10030     (t := type_of(base_type(designated_type(type_mark)))) ∈ incp_types
10031        then t
10032
10033    else om
10034end ;
10035
10036if kind = 'limited private' and
10037          (is_generic_type(type_mark) or scope_of(r)   open_scopes)
10038then kind := 'limited' ;
10039end if ;
10040
10041return kind ;
10042
10043end proc private_kind ;
10044
```

is_fully_private

```
10046proc is_fully_private(type_mark) ;
10047
10048-- Check whether a composite type has an 'incomplete' private component.
10049
10050const f_types = ['private', 'limited private', 'incomplete'] ;
10051
10052return  is_set (a := misc_type_attributes(type_mark))
10053                                  and exists kind ∈ f_types | kind ∈ a ;
10054end proc is_fully_private;
10055--
```

check_priv_decl

```
10057proc check_priv_decl(kind, type_name) ;
10058
10059-- Verify that the full declaration of a private type satisfies the
10060-- restrictions stated in 7.4.1., 7.4.4.
10061
10062package_name := scope_of(type_name) ;
10063if kind = 'private' and is_limited_type(type_of(type_name)) then
10064    errmsg('Private type requires full declaration ',
10065                                  ' with non limited type','7.4.1') ;
10066
10067elseif nature(type_name) = 'array' then
10068    errmsg('Private type cannot be fully declared ',
10069                ' as an unconstrained array type','7.4.1') ;
10070--
```

```
10071else
10072    -- If the private type is not declared with discriminants, it cannot
10073    -- be instantiated with a type with discriminants. Retrieve the pri-
10074    -- vate declaration to find if discriminant list was present.
10075    [-,-,-[-, disc_list], attributes ] :=
10076                                    private_decls(package_name)(type_name) ;
10077
10078    if can_constrain(type_name) and disc_list = [ ] then
10079        errmsg('Private type without discriminants cannot be given ',
10080        'full declaration with discriminants','7.4.1') ;
10081    -- and viceversa.
10082    elseif disc_list /= [ ] and
10083     (nature(type_name) /= 'record' or not has_discriminants(type_name))
10084    then
10085        errmsg('A private type with discriminants must be given full ',
10086        'declaration with a discriminated type', '7.4.1') ;
10087--
10088    elseif 'out' ∈ attributes ? {}
10089    and is_limited_type(type_name) then
10090       errmsg('Use of type for an OUT parameter requires ',
10091            ' full declaration  with non limited type', '7.4.4') ;
10092--
10093    end if ;
10094--
10095end if ;
10096-- Composite types defined in the package and which include a component
10097-- whose type is type_name are now usable in full (if type_name itself is
10098-- not limited). They  may be defined in the visible part of the package,
10099-- or in the (current) private part.
10100-- The private dependents are part of the attributes of the private type.
10101
10102if not is_limited_type(type_name) then
10103    (forall t ∈ attributes ? {} |
10104            (scope_of(t) = package_name or scope_of(t) = scope_name))
10105       -- Save visible definition before updating.
10106       private_decls(package_name)(t) := SYMBTABF(t) ;
10107         if is_set(misc_type_attributes(t)) then          -- safety code
10108           misc_type_attributes(t) less:= kind ;
10109         end if ;
10110    end forall ;
10111end if ;
10112check_generic_usage(type_name) ;
10113
10114end proc check_priv_decl ;
```

private_ancestor

```
10116proc private_ancestor(t_name) ;
10117
10118-- A type name has  a private ancestor  if it  is a subtype of, or has a
10119-- component which is a subtype of, a private type whose full definition
10120-- has not been seen yet. If the private ancestor of  t is defined, then
10121-- t cannot  appear in  a type derivation,  and its  elaboration must be
10122-- performed after that of the ancestor.
10123
10124const relevant_scopes = {'package_spec', 'generic_package_spec',
10125                                                    'private_part'} ;
10126
10127--
10128if  nature(scope_name) ∈ relevant_scopes
10129or (nature(scope_name) ∈ ['record', 'void']
10130    and nature(scope_of(scope_name)) ∈ relevant_scopes)
10131then pass ;                                  -- will have to check further
10132    else                              -- No private ancestor.
10133return om ;
10134end if ;
10135
10136if type_of(t_name) ∈ priv_types
10137    and scope_of(t_name) ∈ open_scopes then
10138--
10139    if not is_generic_type(t_name) then
10140        return t_name ;
10141    else       -- A generic type is never seen by the interpreter
10142        return om ;
10143    end if ;
10144--
10145else
10146    case nature(t_name) of
10147    ('subtype'):
10148       return private_ancestor(base_type(t_name)) ;
10149
10150    ('array'):
10151       return private_ancestor(component_type(t_name)) ;
10152
10153    ('record'):
10154      if exists [id, comp] ∈ declared_components(base_type(t_name)) |
10155                              (pr := private_ancestor(type_of(comp))) /= om
10156      then
10157        return pr ;
10158      end if ;
10159
10160    ('access'):
10161        -- Access types need not be deferred.
```

```
10162         return om ;
10163
10164     end case ;
10165
10166end if ;
10167
10168return om ;              -- If none of the above.
10169
10170end proc private_ancestor ;
```

end private

```
10172proc end_private(priv_part) ;
10173
10174-- Check that private types and deferred constants were redeclared, and
10175-- other identifiers were not redeclared. (This cannot be done  sooner
10176-- because the private part is implemented as a separate scope which is
10177-- here merged with the declarations from the visible part.)
10178
10179    -- Save private declarations before exiting scope od private part.
10180
10181    decls := declared(scope_name);
10182    priv_scope_name := scope_name ;
10183    old_vis := vis_mods ;                -- may include inner packages.
10184    popscope();
10185
10186    vis_mods := old_vis ;                -- Which are visible in the body.
10187
10188    -- For all those private objects that are not renaming of outer
10189    -- entities, mark their scope correctly.
10190
10191    (forall [id, u_name] ∈ decls | scope_of(u_name) = priv_scope_name)
10192         scope_of(u_name) := scope_name ;
10193    end ;
10194
10195    (forall [nam, u_name] ∈ visible(scope_name) )
10196        -- Check that private types and deferred constants were fully
10197          -- declared in private part.
10198
10199        if type_of(u_name) ∈ priv_types
10200          and scope_of(u_name) = scope_name                    --
10201        and not is_generic_type(u_name)
10202           or nature(u_name) = 'constant' and signature(u_name) = [ ]
10203        then
10204            -- Private object did not get private description.
10205          errmsg('Missing full declaration in private part for ',
10206                                        nam, '7.4.1');
10207          end if;
10208
```

```
10209       -- check for possible redeclaration.
10210
10211       priv_name := decls(nam) ;
10212       if priv_name /= om then
10213           errmsg('Redeclaration of ' + nam, ' in private part',
10214                                   '7.2 , 7.4.1') ;
10215           decls(nam) := om ;        -- lest declared be multivalued.
10216       end if ;
10217
10218   end forall;
10219
10220   -- Now exchange contents of private_decls and symbol table. In this
10221   -- fashion the declarations that were visible in the private part of
10222   -- the package, and that will be visible in the package body, become
10223   -- inaccessible outside of the package specification.
10224
10225   (for [u_name, vis_decl] ∈ private_decls(scope_name))
10226      priv_decl := SYMBTABF(u_name) ;
10227      SYMBTABF(u_name) := vis_decl ;
10228      TO_XREF(u_name) ;
10229      private_decls(scope_name)(u_name) := priv_decl ;
10230   end for ;
10231
10232   -- Finally, merge fully private objects into declared map for package
10233   -- spec. The visible map is unaffected.
10234
10235   declared(scope_name) +:= decls;
10236
10237   return priv_part ;
10238
10239end proc end_private;
10240
```

end_specs

```
10242proc end_specs(module_head, visible_part) ;
10243
10244-- This procedure is invoked at the end of a module specification.
10245-- If this spec. is a compilation unit, then we save in UNIT_DECL
10246-- all the declarations processed for the module. These declarations
10247-- are retrieved (by procedure get_specs) when the separate compilation
10248-- facility is used.
10249-- In the case of generic modules, we must we must save the
10250-- specs of the generic object in its signature, to simplify its instan-
10251-- tiation. In order to insure that a separately compiled generic object
10252-- is properly saved, we make the object name accessible within its own
10253-- scope. This insures that its symbol table entry is correctly saved.
10254
10255[kind, nam] := module_head ;
```

```
10256
10257-- save visible mods for this scope.
10258--
10259old_vis := [v ∈ vis_mods | v /= 'ASCII'];
10260
10261popscope() ;
10262
10263if is_comp_unit then
10264    save_spec_info(nam, old_vis) ;
10265else
10266    -- If the module is a sub-unit, make sure that it is visible in
10267    -- its enclosing scope (except if it is a generic package).
10268
10269    vis_mods +:= [v ∈ old_vis | v   vis_mods] ;
10270    if kind /= 'generic_package' then
10271        vis_mods +:=  [nam];
10272    end if ;
10273
10274end if ;
10275
10276return [['package', nam, visible_part]] ;                    --
10277
10278end proc end_specs ;
10279--
```

check_incomplete_decls

```
10281proc check_incomplete_decls(scope, rw decls) ;
10282
10283-- At the end of a block, verify that entities that need a body received
10284-- one, and add dummy package bodies for package specs that do not have
10285-- an explicit one.
10286
10287(forall [id, name] ∈ declared(scope) )
10288    if needs_body(name)
10289     and not is_anonymous_task(name)
10290     and not exists stub ∈ domain LIB_STUB
10291                                        | stub(2) = original_name(id)
10292    then
10293        errmsg('Missing body for ' + nature(name) + ' ',
10294                        original_name(scope) + '.' + id, '7.3') ;
10295
10296    elseif type_of(name) = 'incomplete' then
10297        errmsg('Missing full type declaration for incomplete type ',
10298                id, '3.8.1') ;
10299
10300    elseif nature(name) = 'package_spec' then
10301        decls with:= ['package_body', name, [ ]] ;
10302    end if ;
```

```
10303end ;
10304
10305end proc check_incomplete_decls ;
10306--
```

needs_body

```
10308proc needs_body(name) ;
10309
10310
10311
10312-- Procedures and function specs need bodies of course. So do package
10313-- specs that contain objects which need bodies.
10314
10315nat := nature(name) ;
10316return
10317    if nat ∈ ['package_spec', 'generic_package_spec']
10318        then (exists [-,obj] ∈ visible(name) | needs_body(obj))
10319        or
10320        (exists [-,obj] ∈ declared(name) | type_of(obj) = 'incomplete')
10321    elseif rmatch(nat, '_spec') /= om then true
10322    else false
10323    end ;
10324
10325end proc needs_body ;
```

get_specs

```
10327proc get_specs(name) ;
10328
10329-- Obtain the specification for a module name. This is done in
10330-- two cases :
10331-- a) When we process the WITH clause of a new compilation
10332-- unit.
10333-- b) When we compile the body of a module. The corresponding
10334-- module specification must have been compiled already, an must be
10335-- available. The specs for the module are used to initialize the
10336-- declarations for the body. Note that this is not equivalent to
10337-- placing the module name in the list of used modules for this
10338-- compilation.
10339
10340
10341
10342if RETRIEVE(['spec', name]) then
10343    spec_name := ['spec', name] ;
10344elseif RETRIEVE(['spec instance', name]) then
10345    spec_name := ['spec instance', name] ;
10346else
10347      errmsg('Compilation unit not found: ', name,'10.1');
```

```
10348    return;
10349end if;
10350
10351versions_used(spec_name) := if COMP_DATE(spec_name) = om then om
10352                        else COMP_DATE(spec_name)(spec_name) end;
10353
10354-- Read in the unique names and the declared types of all visible
10355-- names in the module specification.
10356[unit_unam, specs, decmap, old_vis, notvis] := UNIT_DECL(spec_name) ;
10357specs   ?:= {};
10358decmap  ?:= {};
10359old_vis ?:= {};
10360notvis  ?:= {};
10361
10362(forall info = specs(u_nam))
10363    SYMBTABF(u_nam) := info ;
10364end forall ;
10365
10366(for dec = decmap(sn))
10367    declared(sn) := dec;
10368    visible(sn) :=  dec - (notvis(sn) ? {});
10369end for;
10370--
10371-- Predefined unit that are mentioned in a WITH clauses are not saved in
10372-- UNIT_LIB, for storage reasons. Their contents must be brought in ex-
10373-- plicitly, but their direct visibility must not be modified.
10374
10375(for u ∈ old_vis | ['spec',u] ∈ PREDEF_UNITS and u   vis_mods)
10376    maybe_decl := declared('STANDARD#0')(u);
10377    get_specs(u) ;
10378    declared('STANDARD#0')(u) := maybe_decl ;
10379end ;
10380--
10381vis_mods +:= [v ∈ old_vis | v   vis_mods];
10382declared('STANDARD#0')(name) := unit_unam;
10383
10384
10385end proc get_specs ;
```

8. Visibility rules

find_old

```
10388proc find_old(id) ;
10389
10390-- return unique name of identifier.
10391-- Yield error if identifier is undefined.
10392-- In the case of long and short integers, indicate that they are
10393-- unimplemented rather than 'undefined'.
10394
10395const unimp = {'SHORT_INTEGER','LONG_INTEGER','SHORT_FLOAT',
10396              'LONG_FLOAT'} ;
10397
10398
10399
10400
10401u_name := check_old(id) ;
10402if u_name = '?' then
10403    if is_tuple(id) and id(1) = 'name' then id := id(2) ; end if ;
10404    if id   unimp then
10405    -- The identifier is undefined, or not visible. This is an error.
10406    name_error(id) ;
10407    else
10408       errmsg(id, ' is not supported in current implementation','none');
10409    end if ;
10410    -- insert in current scope,and give it default type.
10411    if declared(scope_name)(id) = om then
10412         newn := if is_identifier(id) then id else str newat end ;
10413         u_name := find_new(newn) ;
10414         nature(u_name)  := 'obj' ;              -- Could be more precise.
10415         type_of(u_name) := 'any' ;
10416    end if ;
10417end if ;
10418
10419return u_name ;
10420end proc find_old ;
```

name_error

```
10422proc name_error(id) ;
10423
10424
10425
```

```
10426-- Name was not found in environment. This may be because it is undeclared,
10427-- or because several imported instances of the name hide each other.
10428-- The marker '?' is also returned when a type name is mentioned in
10429-- the middle of its own elaboration.
10430
10431if all_imported_names = {} then
10432    if declared(scope_name)(id) = om then
10433        errmsg('identifier undeclared or not visible ', id,'3.1') ;
10434    else
10435        errmsg('Invalid reference to type ', id,'3.3') ;
10436    end if ;
10437else
10438    names := +/[ original_name(scope_of(x)) + '.' + original_name(x)
10439                        + ' ':  x ∈ all_imported_names ] ;
10440    errmsg('Ambiguous identifier. Could be one of: ',names,'8.3, 8.4') ;
10441 end if ;
10442
10443end proc name_error ;
10444
10445
```

check_old

```
10447 proc check_old(name) ;
10448
10449
10450
10451-- This procedure performs name resolution for several syntactic
10452-- instances of -name-. These include identifiers, selected components,
10453-- array indexing and slicing, function calls and attribute expressions.
10454-- If -name- is an identifier and is undeclared, this proc yields
10455-- the special marker '?' which is used by error routines.
10456-- If -name- is overloaded, the procedure returns the set of overloaded
10457-- names which correspond to -name-. This set is constructed by
10458-- scanning first the open scopes, and then examining visible modules.
10459-- To facilitate the collection of overloaded names, the procedure
10460-- chain_overload, which is called when a procedure specification, or
10461-- and enumeration type are processed, collects successive overloads of the
10462-- same id together, using the -overloads- field of the symbol table.
10463
10464
10465if is_tuple(name) then
10466    case name(1) of
10467
10468        ('call?'): found := array_or_call(name) ;
10469
10470        ('.'):     found := find_qual(name) ;
10471
10472        ('name', 'string'): found := check_old(name(2)) ;
```

```
10473
10474        ('"'):    found := name ; found(3) := find_old(found(3)) ;
10475
10476        ('@'):     found := name ; found(2) := find_old(found(2)) ;
10477
10478    else
10479        errmsg('Invalid object for name: ', name,'3.2') ;
10480        found := '?' ;
10481    end case ;
10482
10483elseif exists sc = open_scopes(sc_num) st
10484        (u_name := declared(sc)( name)) /= om then
10485        if not can_overload(u_name) then
10486                found := u_name ;
10487            TO_XREF(u_name) ;
10488        else
10489            names := overloads(u_name) ;
10490
10491            -- Scan open scopes for further overloadings.
10492            (for i ∈ [sc_num+1..#open_scopes])
10493
10494              if (u_n := declared(open_scopes(i))(name)) = om then
10495                    continue for i ;
10496
10497              elseif not can_overload(u_n) then
10498                    return names ;
10499
10500              else names +:= {o ∈ overloads(u_n) |
10501                  not exists n ∈ names |
10502                        (same_type(type_of(n), type_of(o)) and
10503                         same_signature(signature(n), signature(o)))} ;
10504              end if ;
10505            end for ;
10506            imported := collect_imported_names(name) ;
10507
10508            -- Keep only the imported names which are not hidden
10509            -- by visible names with the same signature.
10510            if is_overloaded(imported) then
10511                    names +:= { foreign : foreign ∈ imported |
10512                                not exists n ∈ names |
10513                     same_type(type_of(n),type_of(foreign)) and
10514                    same_signature(signature(n),signature(foreign))} ;
10515            end if ;
10516
10517            found := names ;
10518        end if ;
10519
10520elseif (imported := collect_imported_names(name)) /= {} then
10521    found :=  imported ;
```

```
10522
10523    -- the error recovery routine sends a " when it can
10524    -- recover by token insertion. return it as is, to avoid
10525    -- subsequent spurious messages.
10526elseif name = " then found := name ;
10527else
10528        found := '?' ;
10529end if ;
10530
10531if is_set(found) and #found = 1 then
10532     TO_XREF(arb found) ;
10533end if ;
10534
10535return found ;
10536end proc ;
10537
```

collect_inported_names

```
10539proc collect_imported_names(name) ;
10540
10541
10542-- This procedure collects the set of all imported names corresponding
10543-- to identifier -name-, which appear in currently visible modules.
10544-- An imported identifier is visible if :
10545-- a) It is not an overloadable identifier, and it appears in only
10546-- one used package.
10547-- b) Or, all of its appearances in used packages are overloadable.
10548
10549imported := {} ;
10550(forall used ∈ used_mods | (f:= visible(used)) /= om
10551                                  and (foreign := f(name)) /= om )
10552        if can_overload(foreign) then
10553           imported +:= overloads(foreign) ;
10554        else
10555           if imported /= {} then
10556                return {} ;                    -- mutual hiding
10557           else
10558              imported := {foreign} ;
10559           end if ;
10560        end if;
10561end forall ;
10562
10563
10564-- Save imported names in global variable, for possible error message.
10565all_imported_names := imported ;
10566
10567if exists fgn ∈ imported | not can_overload(fgn) then
10568    if #imported = 1 then return fgn ;
```

```
10569    else                    -- various visible names hide each other.
10570        return {} ;
10571    end if ;
10572else                                 -- All occurrences are overloadable.
10573    return imported ;                                          --
10574end if ;
10575end proc ;
```

array_or_call

```
10577 proc array_or_call(name) ;
10578
10579-- This procedure resolves the construct
10580--        name aggregate
10581-- The meaning of this construct is one of the following :
10582-- a) Array reference.
10583-- b) Procedure call.
10584-- c) Entry invocation.
10585-- d) Conversion expression.
10586
10587
10588
10589
10590
10591[-,obj_name,arg_list] := name ;
10592
10593-- Find unique name of object (procedure, array, etc).
10594obj_name := find_old(obj_name) ;
10595if obj_name = '?' then
10596    -- error message emitted already by find_old.
10597    return name ;
10598end if ;
10599
10600if is_identifier(obj_name) and   is_type(obj_name) then
10601    -- Case of a conversion.
10602
10603    if # arg_list = 1 then
10604        arg := arg_list(1) ;
10605    else                    -- Conversion of an aggregate.
10606        arg := ['aggregate'] + arg_list ;
10607    end if ;
10608
10609    return ['convert', obj_name, arg] ;
10610
10611elseif is_overloaded(obj_name) then
10612    -- Must be a procedure or function call.
10613    call_name := [obj_name] + arg_list;
10614--
10615    -- The  nature of at least one  of the  overloaded instances  must be
```

```
10616    -- callable. This is checked by the type resolution routines. An
10617    -- unpleasant syntactic ambiguity appears if parameterless functions
10618    -- that return an array type appear in obj_name. In this case the
10619    -- expression must be reformatted as an indexing on the result of a
10620    -- function call. If both parameterless and parametered functions
10621    -- are present, then the tree itself is ambiguous, and both parsings
10622    -- must be carried, to be resolved by the type resolution routines.
10623
10624    npfs := {f ∈ obj_name | parameterless_callable(f)
10625                  and (is_array(t := type_of(f))
10626                    or is_access(t) and is_array(designated_type(t))) } ;
10627    if npfs /= {} then
10628       np_call := index_or_slice([obj_name], arg_list) ;
10629
10630       if np_call(1) = '[..]' then
10631            -- no ambiguity: it must be a slice.
10632            call_name := np_call ;
10633         else
10634           call_name := ['call_or_index', obj_name, arg_list] ;
10635         end if ;
10636
10637    end if ;
10638--
10639    return call_name ;
10640else
10641    -- Object or expression. Either an array expression or an attribute.
10642
10643    oprator := obj_name(1) ;
10644
10645    if oprator = '''' then                          -- Predefined attribute.
10646         -- Right argument of attribute is first element of
10647         -- argument list.
10648         obj_name(4) := arg_list(1) ;
10649         return obj_name ;
10650
10651    else                                        -- Indexed name or slice.
10652       return index_or_slice(obj_name, arg_list) ;
10653    end if ;
10654end if ;
10655
10656end proc array_or_call ;
10657--
```

parameterless_callable

```
10659proc parameterless_callable(f) ;
10660
10661-- Assert that f is a parameterless function, or that default values
10662-- exist for all its parameters and it can be called without arguments.
```

```
10663
10664return nature(f) ∈ {'function', 'function_spec'} and
10665        (forall [-,-,-,init_val] ∈ signature(f) | init_val /= [ ] );
10666
10667end proc parameterless_callable ;
10668--
```

index_or_slice

```
10670proc index_or_slice(obj_name, arg_list) ;
10671
10672-- A slice is not always recognizable sintactically from an
10673-- indexing expression. v(arg) is a slice in 3 cases:
10674-- a) arg is a range : L..R
10675-- b) arg is of the form V'RANGE
10676-- c) arg is a type mark, possibly with a range constraint.
10677
10678if #arg_list = 1 and is_tuple(arg := arg_list(1))
10679then
10680    if arg(1) = 'subtype' then
10681        [-, type_mark, constraint] := arg ;
10682        s := ['[..]', obj_name,                                    --
10683                ['subtype', type_mark, constraint]] ;
10684
10685    elseif arg(1) = 'range' then
10686        s := [ '[..]', obj_name, ['subtype',[ ], arg]] ;
10687
10688    elseif arg(1) = 'name' and is_tuple(e := find_old(arg))
10689      and is_range_attribute(e) then
10690        s := [ '[..]', obj_name, e] ;
10691
10692    elseif arg(1)='name' and is_identifier(mark:=find_old(arg))
10693      and is_type(mark) then
10694        s := ['[..]', obj_name, mark] ;
10695    else                              -- single index operation.
10696        s := ['[ ]', obj_name, [arg] ] ;
10697    end if ;
10698
10699else                                          -- multiple indexing operation.
10700    s := [ '[ ]', obj_name, arg_list] ;
10701
10702end if ;
10703
10704return s ;
10705
10706end proc index_or_slice ;
10707
```

find_qual

```
10709 proc find_qual(name) ;
10710
10711 -- Resolve  a qualified name. The prefix must be one of the following:
10712 --
10713 -- a) A variable whose type is a record or a task.
10714 -- b) An open scope.
10715 -- c) A visible module .
10716 --
10717 -- If the prefix of a qualified name is an expression, then it
10718 -- must yield a record or task object( or an access to such).
10719
10720
10721
10722
10723 [-, name1, selector ] := name ;
10724
10725 obj := find_old(name1) ;
10726 if obj = '?' then        -- Undeclared name. Error message already emitted
10727    return [ ] ;
10728 end if ;
10729
10730 if is_identifier(obj) then
10731    obj_type := type_of(obj) ;
10732
10733    if is_record(obj_type)  or
10734       is_access(obj_type) then
10735          return [ '.', obj, selector] ;
10736
10737    elseif nature(obj_type) ∈ {'task_type', 'task_type_spec'} then
10738
10739          -- if the selector is an entry name, return a selected component.
10740          -- Otherwise it is a variable named within the body of the task:
10741          -- return its qualified name.
10742
10743          if obj ∈ open_scopes and (u_n := declared(obj)(selector)) /= om
10744          and nature(u_n)   {'entry', 'entry_family'} then
10745             TO_XREF(u_n) ; return u_n ;
10746          else
10747             return ['.', obj, selector] ;
10748          end if;
10749
10750    elseif obj ∈ open_scopes then
10751          if (var_name := declared(obj)(selector)) /= om then
10752             if can_overload(var_name) then
10753                return overloads(var_name) ;
10754             else TO_XREF(var_name) ; return var_name ;
```

```
10755           end if ;
10756       elseif nature(obj) = 'void' then
10757           errmsg('premature usage of ',original_name(obj), '8.3(16)');
10758       else errmsg(selector + ' not declared in ',
10759                       name1, '4.1.3, 8.3') ;
10760           return 'any_id' ;
10761       end if ;
10762   elseif obj ∈ vis_mods and
10763       (imported_name := visible(obj)(selector)) /= om then
10764           -- If the object is overloaded, collect its local occurences.
10765
10766           if can_overload(imported_name) then
10767               nams := overloads(imported_name) ;
10768               if selector ∈ op_designators then
10769                   -- Include implicitly defined operators, if they are
10770                   -- not hidden by an explicit declaration in the scope
10771                   predef_op := declared('STANDARD#0')(selector) ;
10772                   if forall subp ∈ nams |
10773                       (op_matches_spec(predef_op, signature(subp),
10774                                       type_of(subp)) = om)
10775                   then
10776                       nams with:= obj + '.' + selector ;
10777                   end if ;
10778               end if ;
10779               return nams ;
10780           else
10781               TO_XREF(imported_name) ; return imported_name ;
10782           end if ;
10783
10784    elseif obj_type = 'any' then
10785       -- Object was undeclared, and error message emitted already.
10786       -- return as is, and forget about the selector.
10787       return obj ;
10788    elseif obj ∈ vis_mods and selector ∈ op_designators then
10789       -- Case of P.'+' and the like. Return it as an overloaded
10790       -- name. routine result-types will special-case this marker
10791       -- for type resolution.
10792       return {obj + '.' + selector} ;
10793    else errmsg('Invalid name for selected component: ',
10794                       name1,'4.1.3') ;
10795           return [ ] ;
10796   end if ;
10797
10798elseif is_overloaded(obj) then
10799   -- It can be an enclosing procedure, to be resolved by the name of
10800   -- the selected variable; it can also be a call to a parameterless
10801   -- procedure that yields a record type.
10802
10803   if exists proc_name ∈ obj |
```

```
10804         proc_name ∈ open_scopes and
10805         (var_name := declared(proc_name)(selector)) /= om then
10806         if can_overload(var_name) then
10807           return overloads(var_name) ;
10808         else
10809           TO_XREF(var_name) ; return var_name ;
10810         end ;
10811     elseif selector ∈ op_designators and exists proc_name ∈ obj |
10812         proc_name ∈  open_scopes and
10813           ( exists [-,typ] ∈ declared(proc_name) | root_type(typ) ∈
10814               {'universal_integer', 'universal_real', '$FIXED'} )
10815     then
10816       return {proc_name + '.' + selector} ;
10817
10818     elseif exists proc_name ∈ obj |                              --
10819         parameterless_callable(proc_name) then                   --
10820         -- Selector applied to call to parameterless function.
10821         return ['.', [obj], selector] ;
10822
10823     else errmsg('Ambiguous name in selected component: ',
10824                                 name1,'4.1.3') ;
10825         return [ ] ;
10826   end if ;
10827else
10828   -- If it is an expression, it must yield a record .
10829   -- Type checking will insure that this is the case.
10830   return ['.', obj, selector] ;
10831end if ;
10832end proc find_qual ;
```

find_agg_types

```
10834proc find_agg_types ;
10835-- The possible types of an aggregate  are all the structured types  that
10836-- are  visible, even if  not directly  visible.
10837
10838return
10839{}  +/[overloads(agg): s ∈ open_scopes
10840                          |(agg := declared(s)('aggregate')) /= om]
10841    +/[overloads(fgn) : p ∈ vis_mods
10842                          |(fgn :=  visible(p)('aggregate')) /= om] ;
10843
10844end proc find_agg_types ;
```

find_access_types

```
10846proc find_access_types ;
10847
10848-- Similarly, the possible types of NULL, and of any allocator, are all
```

```
10849-- visible access types. To simplify their retrieval, they are treated
10850-- like aggregates, and attached to the marker 'access', whenever an
10851-- access type definition is processed.
10852
10853a_types :=
10854{} +/[overloads(a): s ∈ open_scopes
10855                    |(a := declared(s)('access')) /= om]
10856  +/[overloads(fgn) : p ∈ vis_mods
10857                    |(fgn :=  visible(p)('access')) /= om] ;
10858if a_types = {} then
10859    noop_error := true ;
10860    errmsg('No available access types for allocator', '', '3.8,4.8') ;
10861end if ;
10862return a_types ;
10863
10864end proc find_access_types ;
10865
```

find_new

```
10867proc find_new(name) ;
10868
10869
10870
10871-- insert new name in symbol table of current scope. Check
10872-- against duplications.
10873
10874-- IF error token was seen ('') , return undeclared marker.
10875         if name = '' then return  'any_id' ; end ;
10876
10877         -- add new name to current scope declarations.
10878         -- generate a unique identifier for it.
10879
10880         unique_nam := prefix + name + str newat ;
10881
10882       -- Insert new name in DECLARED table for current scope
10883
10884         if (old := DECLARED(scope_name)( name)) /= om then
10885            -- The name has been seen already. This is acceptable
10886            -- if it was inserted after  some previous  error, in
10887            -- which case its type is 'any'.
10888
10889            if  type_of(old) = 'any' then
10890                 return old ;
10891            else
10892                 errmsg('duplicate identifier: ', name,'8.3') ;
10893            end if ;
10894         else
10895            DECLARED(scope_name)( name) := unique_nam ;
```

```
10896          end ;
10897          -- Initialize symbol table entry. The nature of the new entity
10898          -- is 'void' until the end of its declaration.
10899
10900          SYMBTAB(unique_nam)  := ['void', 'none'] ;
10901          SCOPE_OF(unique_nam) := scope_name ;
10902          TO_XREF(unique_nam) ;
10903          return unique_nam ;
10904 end proc  ;
10905proc check_void(id) ;
10906
10907-- Verify that within a procedure specification no use is made of the
10908-- procedure identifier under any guise. This cannot be automatically
10909-- caught by the name resolution routines.
10910
10911if original_name(scope_name) = id and nature(scope_name) = 'void' then
10912    errmsg('premature usage of ', id, '8.3(16)');
10913end if ;
10914end proc check_void ;
```

new_agg_or_access

```
10916proc new_agg_or_access(marker, type_mark) ;
10917
10918-- The possible types of an aggregate are all composite types that are
10919-- currently visible. To simplify their use, an entry  with the marker
10920-- 'aggregate' is created for each such type definition. Its overloads
10921-- set carries all such types  defined in  the current  scope. This is
10922-- similar to what is done for other overloadable constructs.
10923-- The same is done for access types, using the marker 'access'.
10924
10925scope := if nature(scope_name) = 'private_part' then open_scopes(2)
10926                                                  else scope_name end ;
10927
10928old_def := declared(scope)(marker) ;        -- save if present
10929new_def := marker + str newat ;
10930declared(scope)(marker) := new_def ;
10931scope_of(new_def) := scope ;
10932nature(new_def)   := marker ;
10933type_of(new_def)  := type_mark ;
10934overloads(new_def):= {type_mark} ;
10935
10936if old_def /= om then                        -- Save its overloads
10937    overloads(new_def) +:= overloads(old_def) ;
10938    declared(scope)(str newat) := old_def ;   -- and keep accessible
10939end if ;
10940
10941-- If the type has an incomplete private component, list it in the set
10942-- of private dependents of that ancestor.
```

```
10943maybe_priv := if marker = 'access' then  designated_type(type_mark)
10944                                  else type_mark end ;
10945if (pr := private_ancestor(maybe_priv)) /= om
10946  and scope_of(pr) ∈ open_scopes              -- i.e. still incomplete.
10947then
10948    private_dependents(pr) with:= type_mark;
10949end if ;
10950end proc new_agg_or_access ;
10951
```

original_name

```
10953proc original_name(unique_nam) ;
10954
10955
10956
10957-- This procedure strips the prefix and suffix of a generated name,to
10958-- recover the original source name. Is is used when looking for a
10959-- compilation stub, and for error messages.
10960
10961if is_overloaded(unique_nam) then
10962    unique_nam := arb unique_nam ;
10963end if ;
10964if not is_string(unique_nam) then return '?'; end if;
10965
10966old_name := unique_nam ;
10967loop doing old := break(old_name,'.') ; while old /= om do
10968    old_name := old_name(2..) ;
10969    if old(#old) = "'" then                    -- Special case literal
10970        return "'.'";                          -- '.'
10971    end if ;
10972end loop ;
10973old := break(old_name,'#') ;
10974return if old = om then old_name
10975    elseif old_name(2) = "'" then "'#'"        -- and also '#'
10976else old end ;
10977
10978end proc original_name ;
10979
```

renames

```
10981proc renames(new_id, kind, old_obj) ;
10982
10983
10984
10985-- Process RENAMES clauses. If old_obj is an identifier, then the
10986-- renames clause simply creates a synonym : new_id shares the symbol
10987-- table entry for old_obj. If old_obj is an expression, then the
```

```
10988-- interpreter will have to elaborate it, and a 'renames' statement
10989-- is emitted. In addition, a new symbol table entry is created for
10990-- new_id, with the the appropriate type.
10991
10992case kind of
10993
10994('exception','package'):
10995
10996    old := find_old(old_obj) ;
10997    if not is_identifier(old) then
10998        errmsg('Expect identifier in renaming of ', kind,'8.5') ;
10999    elseif nature(old) /= kind and nature(old) /= kind + '_spec'
11000    then
11001        errmsg(original_name(old), ' is not ' + kind,'8.5') ;
11002    else
11003
11004        declared(scope_name)(new_id) := old ;
11005    end if ;
11006    return [ ] ;
11007
11008('procedure'):
11009
11010    [name, formals, -] := new_id ;
11011
11012    old_procs := find_old(old_obj) ;
11013
11014    if is_overloaded(old_procs) then
11015        old1 := renamed(old_procs, formals, 'none') ;
11016
11017    elseif is_tuple(old_procs) and old_procs(1) ∈ {'.','[ ]'} then
11018        old1 := renamed_entry(old_procs, formals) ;
11019
11020    else
11021        errmsg('Object cannot be renamed as procedure.','','8.5') ;
11022        return [ ] ;
11023    end if ;
11024
11025    if old1 = '' then    -- Ambiguous or invalid signatures.
11026        return [ ] ;
11027    else
11028        -- A renaming declaration provides the subprogram specification
11029        -- and the body as well.
11030        new_obj := chain_overloads(name, ['procedure_spec',
11031                                          'none', formals]);
11032        nature(new_obj) := 'procedure' ;    -- no further body needed .
11033
11034        if is_identifier(old1) then        -- Procedure renames procedure
11035            ALIAS(new_obj) := ALIAS(old1) ? old1 ;                --
11036        return [ ] ;
```

```
11037        else                              -- Procedure renames task entry.
11038           return
11039             [entry_as_procedure(new_obj, signature(new_obj), old1)] ;
11040        end if ;
11041
11042     end if ;
11043
11044('function'):
11045
11046     [name, formals, ret] := new_id ;
11047     old_fun := find_old(old_obj) ;
11048     if is_overloaded(old_fun) then
11049        old1 := renamed(old_fun, formals, ret) ;
11050
11051        if old1 = '' then return [ ] ;
11052        else
11053           new_fun := chain_overloads(name, ['function_spec', ret,
11054                                                    formals, old1]) ;
11055           nature(new_fun) := 'function' ;
11056
11057           if name = '=' and original_name(old1) = '=' then
11058              -- Introduce corresponding inequality operator.
11059              assert
11060                 exists [id, ne] ∈ declared(scope_of(old1)) |
11061                   (id = '/=' and
11062                    same_signature(signature(old1), signature(ne))) ;
11063              new_ne := chain_overloads('/=',
11064                                   ['function', ret, formals, ne]) ;
11065              if nature(ne) /= 'op' then
11066                    ALIAS(new_ne) := ALIAS(ne) ? ne ;
11067              end if;
11068           end if ;
11069           if nature(old1) /= 'op' then
11070              -- Function renaming function(spec). No code is emitted,
11071              -- but check for renaming of renaming.
11072              ALIAS(new_fun) := ALIAS(old1) ? old1 ;
11073              return [ ] ;
11074           else
11075             return [['renames', new_fun, old1]];
11076           end if ;
11077        end if ;
11078     else
11079        -- If old_fun is an expression, it can be an attribute.
11080        -- Verify that the signature match.
11081        if old_fun(1) = '"' then        -- renaming an attribute.
11082           [-, attr, typ] := old_fun ;
11083           old1 := old_fun ;
11084
11085        else
```

```
11086           errmsg('Object cannot be renamed as function.','','8.5') ;
11087           return [ ] ;
11088         end if ;
11089         if not is_type(typ) then
11090           errmsg('Expect a type for the attribute','','4.1.4, 8.5') ;
11091           return [ ] ;
11092
11093         end if ;
11094         [new_return,-] := attribute_type(attr, typ, '', 'I:1') ;
11095       if #formals /= 1 or base_type(ret) /= base_type(new_return)
11096       or base_type(typ) /= base_type(formals(1)(3)) then
11097           errmsg('Function spec. does not match ',
11098                       ' renamed attribute','8.5') ;
11099           return [ ] ;
11100       end if ;
11101         new_fun := chain_overloads(name,
11102                       ['function', ret, formals]) ;
11103   end if ;
11104   return [renamed_attribute(new_fun, formals, old1)] ;
11105
11106else
11107   -- Object renaming.
11108   typ := find_old(kind) ;
11109   if not is_type(typ) then
11110       errmsg('invalid type_mark in RENAMES clause: ','','8.5') ;
11111       return [ ] ;
11112   end if ;
11113--
11114   out_context := true ;              -- Subcomponents of out parameters
11115   old_expr := check_type(typ, find_old(old_obj)) ;
11116   out_context := false ;             -- are certainly renamable.
11117
11118   if is_tuple(old_expr) and old_expr(1) ∈ qualifiers then
11119       -- Constraints implied by the type mark of the clause are ignored
11120       old_expr := old_expr(3) ;
11121   end if ;
11122   -- It is tempting to say that if a simple object is being renamed,
11123   -- the new one has the same unique name. This simple optimization
11124   -- must however be delayed to allow conformance checks, as a renamed
11125   -- entity does not conform to the old name.
11126
11127   if is_discriminant_dependent( old_expr ) then
11128      errmsg('existence of object '+ str new_id ,
11129           ' depends on a discriminant ',
11130           '8.5');
11131      return [ ];
11132
11133   else
11134       new_obj := find_new(new_id) ;
```

```
11135
11136          [nat, -] := check_nat_type(old_expr);
11137          nature(new_obj)  := nat ;
11138          type_of(new_obj) := typ ;
11139          signature(new_obj) := old_expr ;                    --
11140          return [['renames', new_obj, old_expr]] ;
11141      end if ;
11142
11143end  case ;
11144
11145end proc renames ;
11146
```

renamed

```
11148proc renamed(subprogs, formals, ret) ;
11149
11150
11151
11152-- Find the subprogram in the overloaded set -subprog- which matches
11153-- the specification given in a renames clause or in a generic instantia-
11154-- tion.
11155
11156found := {} ;
11157
11158(for subp ∈ subprogs)
11159
11160    if nature(subp) = 'op' or
11161     nature(subp) = om and '.' ∈ subp then        -- Case of P."+"
11162     if not exists [-,-,-,opt_init] ∈ formals | opt_init /= [ ] then
11163        -- Renamings cannot introduce default values for operators.
11164           o := op_matches_spec(subp, formals, ret);
11165            if o /= om then found with:= o ; end if ;
11166    end if ;
11167    else
11168        old_sig := signature(subp) ;
11169         if #old_sig /= # formals or
11170            base_type(ret) /= base_type(type_of(subp))
11171        then
11172            continue for subp ;
11173        else
11174            (forall [-, -, form_type,-] = old_sig(i))
11175
11176               [-, -, new_t,-] := formals(i) ;
11177              if not same_type(form_type, new_t)
11178                    then continue for subp ;
11179               end if ;
11180            end forall ;
11181
```

```
11182              found with:= subp ;
11183          end if ;
11184
11185      end if ;
11186end for ;
11187
11188if # found > 1 then
11189      if exists subp ∈ found | nature(subp) /= 'op'
11190        and scope_of(subp) ∈ open_scopes then
11191          found := {subp ∈ found | nature(subp) /= 'op'} ;
11192
11193      elseif exists subp ∈ found | nature(subp) = 'op' then
11194          found := {subp} ;
11195      end if ;
11196end if ;
11197
11198if #found = 1 then
11199
11200      found := arb found ;
11201      check_modes(formals, signature(found));          --
11202
11203      if is_identifier(found) and nature(found) = 'entry' then
11204          -- renaming of entry within a task. Give full name of
11205          -- entry as qualified expression.
11206          found := ['.', scope_of(found), found] ;
11207      end if ;
11208
11209
11210      return found ;
11211
11212elseif #found >1 then
11213      errmsg('ambiguous subprogram name: ',
11214                              original_name(arb subprogs), '8.5,12.3.6') ;
11215else
11216      errmsg('No match for subprogam specification ','','8.5,12.3.6') ;
11217end if ;
11218return '' ;
11219
11220end proc renamed;
```

check_modes

```
11222proc check_modes(formals, sig) ;
11223
11224(forall [-, md, -] = sig(i))
11225      if md /= formals(i)(2) then
11226          errmsg('parameter modes do not match','','8.5(8)') ;
11227      end if ;
11228end forall ;
```

```
11229end proc check_modes ;
```

op_matches_spec

```
11231proc op_matches_spec(op_nam, formals, ret) ;
11232
11233-- Determine whether a predefined operator matches a given subprogram
11234-- specification. Called for renamings and for name resolution of
11235-- selected components whose selector is an operator designator.
11236-- The matching is analogous to the type-checking of an expression. We
11237-- construct a skeletal argument list out of the formals, and use
11238-- result-types(q.v) to find the specific operator being renamed.
11239
11240
11241if #(formals?[ ]) < 1 or #formals > 2 then return om ;
11242
11243else
11244    arg_list := [ [om, {t}] : [-,-,t] ∈ formals] ;
11245    [[ops,-], types] := result_types({op_nam}, arg_list) ;
11246
11247    if #(ops?[ ]) = 1 and exists t ∈ types |
11248    compatible_types(t,ret) then                              --
11249        return arb ops ;
11250    else
11251        return om ;
11252    end if ;
11253end if ;
11254end proc op_matches_spec ;
```

renamed_attribute

```
11256proc renamed_attribute(new_fun, formals, attr_exp) ;
11257
11258-- Called when renaming an attribute as a function,or instantiating a
11259-- generic subprogram parameter as a function. We construct a function
11260-- body whose single statement returns the specified attribute. Ideally
11261-- the pragma INLINE should be attached to these renamed functions.
11262-- The return type of the function is already known to match the first
11263-- argument of the attribute.
11264--
11265[[name, md, ft, o_i]] := formals ;
11266ft := base_type(ft) ;                          -- Only base type matters
11267formals := [[name, md, ft, o_i]] ;
11268
11269[-, attr, typ] := attr_exp ;
11270if attr ∈ base_attributes then
11271    -- the type returned is the base type of the indicated one.
11272    typ := base_type(typ) ;
11273end if ;
```

```
11274-- Second parameter of attribute is the formal of the function.
11275attr_exp := ["", attr, typ, name] ;
11276--
11277
11278return ['function', new_fun, formals,
11279                [ [ ], [['return', new_fun, 0, attr_exp]], [ ] ] ];--
11280
11281end proc renamed_attribute ;
11282
```

renamed_entry

```
11284proc renamed_entry(entry_expr, formals) ;
11285
11286
11287-- A procedure is being renamed with an expression. This can only be the
11288-- renaming of an entry or a member of an entry family.
11289
11290full_name := find_entry_name(entry_expr) ;
11291
11292if full_name = [ ] then              -- Invalid entry name or expression
11293    return [ ] ;
11294else
11295    [task1, entries, indx] := full_name ;
11296end if ;
11297
11298found := {} ;
11299(forall e ∈ entries)
11300    sig := signature(e) ;
11301    if # sig /= # formals then continue forall e ; end if ;
11302
11303    (forall [-, -, typ] = sig(i))
11304        [-, -, new_typ] := formals(i) ;
11305        if not same_type(typ, new_typ) then
11306            continue forall e ;
11307        end if ;
11308    end forall ;
11309
11310    found with := e ;
11311end forall ;
11312
11313if #found /= 1 then
11314    errmsg('ambiguous or invalid entry name in renaming','','8.5') ;
11315    return '' ;
11316else
11317    -- Use entry name to complete resolution of task name.
11318    e_name := arb found ;
11319    task_name := complete_task_name(task1, type_of(scope_of(e_name))) ;
11320    check_modes(formals, signature(e_name)) ;
```

```
11321end if ;
11322
11323if indx /= om then
11324    if nature(e_name) = 'entry' then
11325        errmsg('Invalid index on entry name (not entry family)',
11326                                                    '','9.5') ;
11327    else
11328        -- entry name is indexed expression. Check type of index.
11329        i_type := index_type(type_of(e_name)) ;
11330        e_name := ['[ ]', e_name, [check_type(i_type, indx)]] ;
11331    end if ;
11332
11333elseif nature(e_name) = 'entry_family' then
11334    errmsg('Cannot rename entry family as a whole','','9.5') ;
11335end if ;
11336
11337return ['.', task_name, e_name] ;
11338
11339end proc renamed_entry ;
```

entry_as_procedure

```
11341proc entry_as_procedure(proc_name, formals, entry_expr) ;
11342
11343-- For a procedure that renames an entry, we emit a procedure definition
11344-- whose body is an entry call. Eventually this should be an inline proc.
11345
11346-- The calling parameters on the entry are the names of the formals of
11347-- the procedure being constructed.
11348
11349form_names := [f(1) : f ∈ formals] ;
11350
11351[-, task, entry_name] := entry_expr ;
11352ecall_stat  := ['ecall', task, entry_name, form_names] ;
11353
11354body := [ [ ], [ecall_stat, ['return', proc_name, 0]], [ ] ] ;
11355
11356return ['procedure', proc_name, formals, body] ;--
11357
11358end proc entry_as_procedure ;
```

check_nat_type

```
11360proc check_nat_type(expn) ;
11361
11362-- Obtain the nature and the actual type of of a renamed  expression,
11363-- and verify that it designates an object.
11364
11365const object_natures = {'constant', 'in', 'inout', 'out', 'obj'} ;
```

```
11366
11367if is_identifier(expn) then
11368    [nat, t, -] := SYMBTAB(expn) ;
11369    if nat   object_natures then
11370        errmsg('Renamed entity must be an object',", '8.5') ;
11371    end if ;
11372    return [nat,t] ;
11373
11374else
11375    [opn, exp1, exp2] := expn ;
11376
11377    case opn of
11378
11379    ('[ ]'):
11380            -- The nature of an indexed component is the same as the
11381            -- nature of the array object itself.
11382          [nat, t] := check_nat_type(exp1) ;
11383            return [nat, component_type(t)] ;
11384
11385    ('[..]'):
11386            -- The nature of the slice is that of the array object.
11387            return check_nat_type(exp1);
11388
11389    ('.'):
11390            [nrec, -]     := check_nat_type(exp1) ;
11391            [nfield, t,-] := SYMBTAB(exp2) ;      -- attributes of selector
11392
11393            -- IF selector is a discriminant, the new entity must be
11394            -- treated as such.  Otherwise the  nature of the record
11395            -- object (constant, formal, etc.) determines that of the
11396            -- new entity.
11397
11398            nat := if  nfield = 'discriminant' then nfield else nrec end;
11399            return [nat, t] ;
11400
11401    ('@'):
11402            -- A dereferenced pointer always yields an object.
11403          [nat, t] := check_nat_type(exp1) ;
11404            return ['obj', designated_type(t)] ;
11405
11406    ('call'):
11407            -- The function being called must yield an access type.
11408          t := type_of(exp1) ;
11409            if not is_access(t) then
11410                errmsg('Renamed entity must be an object',", '8.5') ;
11411            end if ;
11412
11413            return ['obj', t] ;
11414
```

```
11415     ('ivalue'):
11416          return ['constant', 'any'] ;
11417     else                                        -- error somewhere.
11418          return ['obj', 'any'] ;
11419     end case ;
11420end if ;
11421
11422end proc check_nat_type ;
```

newscope

```
11424proc newscope(new_name) ;
11425
11426-- This procedure is invoked when a new lexical scope is entered.
11427-- Lexical scopes include package specifications, package bodies ,
11428-- subprogram bodies and entry bodies (ACCEPT statements) . In addition
11429-- record and task declarations and private parts are treated as scopes.
11430-- In each case, the environment of the previous scope is stacked
11431-- and the symbol table for the new scope is initialized.
11432
11433
11434scope_st with:= scope_info ;
11435scope_name := new_name ;
11436
11437declared(scope_name) ?:= {} ;
11438
11439visible(scope_name) ?:= {} ;
11440
11441open_scopes := [scope_name] + open_scopes ;
11442suffix := str newat ;              -- For the formation of unique names
11443
11444end proc newscope;
11445
```

popscope

```
11447proc popscope() ;
11448
11449-- Ths procedure is called on exit from a completed lexical scope.
11450-- Eventually , it should contain various housekeeping functions
11451-- relating to symbol table statistics and space recovery. For now
11452-- it simply restores the environment of the enclosing scope.
11453
11454
11455scope_info frome scope_st ;
11456
11457end proc popscope ;
11458
```

newmod

```
11460proc newmod(name) ;
11461    -- As a concession to readability, the unique names of compilation
11462    -- units are identical to their identifiers.
11463    if is_comp_unit then
11464        new_name := name ;
11465        -- Enter module name in STANDARD
11466         if declared(scope_name)(name) = om then
11467             declared(scope_name)(name) := new_name ;
11468           scope_of(new_name) := scope_name ;
11469           TO_XREF(new_name) ;
11470         else
11471        errmsg('Duplicate unit name ', name , '8.3');
11472         end if;
11473    else
11474         new_name := find_new(name) ;
11475    end if ;
11476    -- Initialize its symbol table, and enter new scope.
11477    declared(new_name) := visible(new_name) := {} ;
11478    newscope(new_name) ;
11479
11480    -- Construct prefix for names declared within.
11481    prefix := prefix + name + '.' ;
11482
11483 end proc newmod ;
11484
```

use_clause

```
11486proc use_clause(id_list) ;
11487
11488used := [ ] ;
11489
11490(forall id ∈ id_list)
11491    if (rnam:=check_old(id)) = '?' then
11492        errmsg('undeclared name in USE list ', id, '8.4, 10.1') ;
11493    elseif nature(rnam)   {'package', 'package_spec'}
11494    then
11495        errmsg(id,' is not the name of a USEable package','8.4') ;
11496    else  used with:= rnam ;
11497    end if ;
11498end forall ;
11499
11500used_mods +:= [u ∈ used | u   used_mods] ;
11501
11502-- The use clauses appearing in a package specification are declarative
11503-- items, visible in the corresponding body. They are saved here.
```

```
11504
11505if nature(scope_name) ∈ {'package_spec', 'generic_package_spec'} then
11506    use_declarations(scope_name) +:= used ;
11507
11508elseif nature(scope_name) = 'private_part' then
11509    pack_name := open_scopes(2) ;
11510    use_declarations(pack_name) +:= used ;
11511end if ;
11512
11513
11514end proc use_clause ;
11515
```

9. Tasks.

task_spec

```
11518proc task_spec(task_node) ;
11519
11520
11521
11522[kind, id_node, entries_node, opt_specs_node] := AST(task_node) ;
11523
11524-- If this is a task declaration, an anonymous task type is introduced
11525-- for it. Entry declarations are always attached to the task type.
11526-- TBSL : processing of specifications.
11527
11528anon := (kind = 'task_spec') ;
11529id := adasem(id_node) ;
11530t_id := if anon then ( 'task_type:' + id + str newat )
11531                  else id end ;
11532
11533task_type_name := find_type_name(t_id) ;
11534if task_type_name = 'any' then
11535   return [ ] ;                          -- Illegal redeclaration.
11536end if ;
11537old_kind := type_of(task_type_name) ;       -- May be a private type.
11538
11539if anon then XREF lessf:= task_type_name ; end if;
11540
11541SYMBTAB(task_type_name) := ['task_type_spec', task_type_name, [ ]] ;
11542root_type(task_type_name) := task_type_name ;                    --
11543--
11544if old_kind ∈ priv_types then
11545   check_priv_decl(kind, task_type_name) ;
11546end if ;
11547if anon then
11548   t_name := find_new(id) ;
11549   SYMBTAB(t_name) := ['task_obj_spec', task_type_name, [ ]];
11550else
11551   t_name := task_type_name ;
11552end if ;
11553
11554newscope(task_type_name);                  -- introduce new scope
11555prefix := prefix + id + '.' ;              -- For unique names.
11556
11557entries := [ ] ;
```

```
11558priority := om ;
11559
11560(for e ∈ entries_node)
11561    entry := adasem(e) ;
11562    if is_tuple(entry(1)) then
11563        -- priority declaration. Extract it from task specification.
11564        [-, priority] := entry(1) ;
11565    else
11566        entries with:= entry ;
11567    end if ;
11568end for ;
11569
11570entry_list := declared(scope_name) ;
11571popscope() ;
11572signature(task_type_name) := entry_list ;
11573
11574result := [['type', task_type_name,
11575                        ['task_former', entries, [priority]]  ]] ;
11576
11577if anon then
11578    -- Attach entry declarations for task object as well, and emit a
11579    -- declaration for the task object itself.
11580
11581    declared(t_name) := SIGNATURE(t_name) := entry_list ;
11582    (for entry = entry_list(id))
11583        scope_of(entry) := t_name ;
11584    end for ;
11585
11586    result with:= ['object', [t_name], task_type_name] ;
11587end if ;
11588
11589return result ;
11590
11591end proc task_spec ;
11592
```

accept_statement

```
11594proc accept_statement(id, indx, formals, body_node) ;
11595
11596
11597
11598-- This procedure opens a new scope when an ACCEPT statement is seen.
11599-- In the case of an overloaded entry name, it selects the one with
11600-- the matching signature.
11601
11602
11603-- Find the task  in which the accept statement occurs. The accept
11604-- may of course appear within a block or another accept statement.
```

```
11605
11606certain := exists task_name ∈ open_scopes st
11607          NATURE(task_name)  {'block', 'entry', 'entry_family'} ;
11608
11609if NATURE(task_name)  {'task_type', 'task_obj'} then
11610       errmsg('Accept statements can only appear in tasks', '','9.5') ;
11611      entry_name := id ;
11612      return [ ] ;
11613end if ;
11614
11615task_type := if NATURE(task_name) = 'task_obj' then
11616          type_of(task_name) else task_name end ;
11617
11618
11619
11620real_name := entry_name := declared(task_name)(id) ;
11621
11622
11623if entry_name = om then
11624   errmsg('Undefined entry name in ACCEPT: ', id, '9.5') ;
11625   entry_name := id ;                    -- For dummy scope.
11626   return [ ] ;                  -- to Initialize it .
11627
11628elseif nature(entry_name) = 'entry' then
11629   -- Collect all its overloadings and select the one with the
11630   -- correct signature.
11631   entries := overloads(entry_name) ;
11632
11633   if indx /= [ ] then
11634       errmsg('invalid index on entry (not entry family)','','9.5') ;
11635   end if ;
11636
11637   if exists entry_name ∈ entries |
11638                same_signature(signature(entry_name),formals) then
11639      real_name := entry_name ;
11640   else
11641       errmsg('ACCEPT statement does not match any entry','','9.5') ;
11642      return [ ] ;                                          --
11643   end if ;
11644
11645elseif nature(entry_name) = 'entry_family' then
11646
11647       ix_t := index_type(type_of(entry_name)) ;
11648
11649      if indx = [ ] then
11650          errmsg('Missing index for entry family.','','9.5') ;
11651
11652       else
11653          indx := check_type(ix_t, indx) ;
```

```
11654          end if ;
11655
11656        real_name := ['[ ]', entry_name, [indx]] ;
11657else
11658     errmsg('invalid entry name in accept: ', id, '9.5') ;
11659     return [ ] ;
11660end if ;
11661TO_XREF(entry_name) ;
11662
11663reprocess_formals(entry_name, formals) ;                    --
11664--
11665if entry_name ∈ open_scopes then
11666     errmsg('An accept statement cannot appear within an accept ',
11667                              ' for the same entry','9.5') ;
11668end if ;
11669--
11670newscope(entry_name) ;
11671
11672body := adasem(body_node) ;
11673popscope() ;
11674
11675return ['accept', real_name, body] ;
11676
11677
11678end proc accept_statement ;
11679
```

entry_decl

```
11681proc entry_decl(id, formals) ;
11682
11683
11684-- An entry declaration is treated like a procedure specification.
11685-- An anonymous type is created for the entry object. This type is
11686-- used by the interpreter to build the environment of an entry.
11687
11688
11689entry := chain_overloads(id, ['entry', 'none', formals]) ;
11690
11691entry_type := anonymous_type() ;
11692SYMBTAB(entry_type) := ['entry_former', scope_name, signature(entry)] ;
11693
11694return ['object', [entry], entry_type] ;
11695
11696end proc entry_decl ;
11697
```

entry_family_decl

```
11699proc entry_family_decl(id, opt_range, formals) ;
11700
11701
11702
11703-- An entry family is not an overloadable object. It is constructed
11704-- as an array of entries. An anonymous type is introduced for the entry
11705-- former, just as for an entry declaration, and another is introduced
11706-- for the array representing the family.
11707
11708entry := find_new(id) ;
11709
11710f := process_formals(entry,formals) ;
11711
11712entry_type := anonymous_type() ;
11713
11714SYMBTAB(entry_type) := ['entry_former', scope_name, f] ;
11715
11716opt_range := make_index(opt_range) ;
11717family_type := anonymous_type() ;
11718SYMBTAB(family_type) :=
11719                    ['array', family_type, [[opt_range], entry_type]] ;
11720root_type(family_type) := family_type ;
11721
11722SYMBTAB(entry) := ['entry_family', family_type, f] ;
11723
11724return ['object', [entry], family_type] ;
11725
11726
11727end proc entry_family_decl ;
11728
```

entry_call

```
11730proc entry_call(obj, arg_list) ;
11731
11732
11733-- process an entry call. -obj- is a selected component of an expression,
11734-- whose type is a task type, or else an indexed component of an array
11735-- expression, in the case of a call to a member of an entry family.
11736
11737full_name := find_entry_name(obj) ;
11738
11739if full_name = [ ] then                    -- previous error.
11740    return [ ] ;
11741else
11742    [task1, entry_names, indx] := full_name ;
```

```
11743end if ;
11744
11745if is_array(family := type_of(e := arb entry_names)) then
11746
11747    entry_type := component_type(family) ;
11748    range_typ  := index_type(family) ;
11749
11750    if indx = om then
11751        -- Case of a call to a parameterless family. The arg-list
11752        -- is actually the index expression.
11753        if #arg_list /= 1 then
11754            errmsg('Call to member of entry family requires one',
11755                                        ' index','9.5, 3.6.1') ;
11756            return [ ] ;
11757        else
11758            indx := arg_list(1) ;
11759            arg_list := [ ] ;
11760        end if ;
11761    end if ;
11762    -- The actual entry name is given by an indexed expression.
11763    e_name := [ '[ ]', e, [check_type(range_typ , indx)]] ;
11764
11765    -- but resolution of the call needs only the family name.
11766    ecall := check_type(family, [entry_names] + arg_list);
11767
11768else
11769    if indx /= om then
11770        errmsg('invalid index on ' + original_name(arb entry_names) ,
11771                ' (not an entry family)', '9.5') ;
11772    end if ;
11773
11774    ecall := check_type('none', [entry_names] + arg_list) ;
11775    e_name := ecall(3) ;                -- The resolved simple entry
11776end if ;
11777
11778-- having resolved the call, use the unique entry name to complete the
11779-- resolution of the task name itself.
11780[-, task, -, arg_list] := ecall ;
11781
11782return ['ecall', complete_task_name(task1, type_of(task)),
11783                                                e_name, arg_list] ;
11784
11785end proc entry_call ;
```

check_entry_call

```
11787--
11788proc check_entry_call(stat_node) ;
11789
```

```
11790-- Verify that the call in a timed entry call or a conditional entry call
11791-- is indeed a call to an entry,  and not to  a procedure, even if it re-
11792-- names and entry.
11793
11794stat := adasem(stat_node) ;
11795if stat=[ ] then return stat ;                    -- previous error.
11796elseif stat(1) = 'ecall' then return stat ;       -- simple case
11797elseif stat(1) = 'call' then
11798    errmsg('context requires entry call', '', '9.7.2, 9.7.3') ;
11799    return [ ] ;
11800else                                             -- Previous error.
11801    return stat ;
11802end if ;
11803end proc check_entry_call ;
11804--
11805
```

find_entry_name

```
11807proc find_entry_name(obj) ;
11808
11809
11810
11811-- Find the name of an entry or entry family, given by a qualified and
11812-- ( in the case of a family) indexed expression. This differs from other
11813-- cases of name resolution in that the name of the task containing the
11814-- entry can be given by a complex expression that must be resolved.
11815-- This procedure is also called to validate the argument of the COUNT
11816-- attribute; this attribute can only be used within the task body, in
11817-- which the entry need not be named as a selected component. An entry
11818-- name will then be seen as an overloaded identifier. The task name is
11819-- the scope of the entry.
11820
11821if is_overloaded(obj) then
11822    entry_name := arb obj ;
11823    task_name:= scope_of(entry_name) ;
11824    entry := original_name(entry_name) ;
11825    if not is_task_type(type_of(task_name)) then          --
11826      errmsg('invalid entry name','','9.5') ;
11827      return [ ];
11828    end if ;
11829
11830elseif is_identifier(obj) then
11831    -- Member of entry family, with index missing.
11832    errmsg('Missing index in name of member of entry family','','9.5') ;
11833    return [ ] ;
11834
11835elseif obj(1) = '.' then
11836    [-, task_name, entry] := obj ;
```

```
11837    indx := om ;
11838else                                        -- case of entry family.
11839    [-, family_name, indx_list] := obj ;
11840
11841    if is_identifier(family_name) then
11842        -- Entry family named in task body. Get enclosing task name.
11843        task_name := scope_of(family_name) ;
11844        entry := original_name(family_name) ;
11845
11846    else                              -- Name is selected component.
11847      [-,task_name, entry] := family_name ;
11848    end if;
11849
11850    if # indx_list /= 1 then
11851      errmsg('Member of entry family requires a single index ',
11852                              '',9.5, 3.6.1') ;
11853       return [ ] ;
11854    end if ;
11855    indx := indx_list(1) ;                    -- In any case.
11856
11857end if ;
11858task1 := valid_task_name(task_name) ;
11859task_types := task1(2) ;
11860if task_types = {} then return [ ] ; end if ;          -- Not a task.
11861
11862entry_names := {} ;
11863(forall t ∈ task_types)
11864  if is_access(t) then t := designated_type(t) ; end if ;
11865
11866  if (e := signature(t)(entry)) /= om then
11867    if nature(e) = 'entry' then
11868        entry_names +:= overloads(e) ;
11869    else                              -- entry family name
11870        entry_names with:= e ;
11871    end if ;
11872  end if ;
11873end forall ;
11874
11875if entry_names = {} then
11876    errmsg('Undefined entry name in task : ', entry,'9.5') ;
11877    return [ ] ;
11878elseif #entry_names > 1 and exists e ∈ entry_names |
11879                                        nature(e) = 'entry_family' then
11880    errmsg('ambiguous entry family name: ', entry, '9.5') ;
11881    return [ ] ;
11882else
11883    return [ task1, entry_names, indx] ;
11884end if ;
11885
```

```
11886end proc find_entry_name ;
11887
```

select_statement

```
11889proc select_statement(node) ;
11890
11891
11892
11893[-, alternative_list, else_stat_node] := AST(node) ;
11894
11895has_delay := exists delay_node ∈ alternative_list
11896                        | node_kind(delay_node) = 'delay_alt' ; --
11897
11898has_term  := exists  term_node ∈ alternative_list
11899                     | node_kind(term_node) = 'terminate_alt' ; --
11900
11901has_else  := not is_empty(else_stat_node) ;
11902
11903if has_delay and has_term then
11904    current_node := term_node ;
11905    errmsg('Delay and terminate alternatives cannot appear ',
11906                'in the same SELECT statement', '9.7.1') ;
11907
11908elseif (has_delay or has_term) and has_else then
11909    current_node := else_stat_node ;
11910    errmsg('ELSE part cannot appear in SELECT statement if ',
11911                'delay or terminate alternatives are present','9.7.1');
11912
11913elseif #[n ∈ alternative_list | node_kind(n) = 'terminate_alt'] > 1
11914then
11915    errmsg('Only one terminate alternative can appear ',
11916                                'in a SELECT statement','9.7.1') ;
11917
11918end if ;
11919
11920alternatives := [adasem(alt_node): alt_node ∈ alternative_list] ;
11921else_stat := if is_empty(else_stat_node) then om
11922                        else adasem(else_stat_node) end ;
11923
11924if not exists alt ∈ alternatives | alt(1) = 'accept'
11925  or alt(1) = 'guard' and alt(3)(1) = 'accept' then
11926        errmsg('SELECT statement must have at least one ACCEPT ',
11927                'alternative', '9.7.1') ;
11928end if ;
11929
11930return ['selective_wait', alternatives, else_stat] ;
11931
11932end proc select_statement ;
```

terminate_statement

```
11934proc terminate_statement ;
11935
11936-- A terminate alternative is annotated with the nesting level of the
11937-- statement, to simplify the retrieval of the task environment.
11938
11939certain := exists scope = open_scopes(i) |
11940                          nature(scope) ∈ ['task_obj', 'task_type'] ;
11941if not certain then
11942    errmsg('Invalid context for TERMINATE alternative', '', '9.7.1') ;
11943    i := 1 ;
11944end if ;
11945-- Loops and exception handlers are scopes for syntactic purposes, but
11946-- not at run-time: exclude them from depth count.
11947out_depth := i - 1 - #[scope ∈ open_scopes(1..i) |
11948                          TYPE_OF(scope) ∈ ['loop', 'handler'] ] ;
11949return ['terminate', out_depth] ;
11950end proc terminate_statement;
```

abort_statement

```
11952proc abort_statement(node_list) ;
11953
11954
11955tasks := [ ] ;
11956
11957(for name_node ∈ node_list)
11958    task_name := find_old(adasem(name_node)) ;
11959    task1 := valid_task_name(task_name) ;
11960    types := task1(2) ;
11961    if types = {} then                          -- previous error
11962        continue ;
11963    else
11964        task_type := arb types ;
11965        if not is_task_type(task_type) then     -- access type maybe.
11966            errmsg('expect task name in ABORT', '', '9.10') ;
11967            continue ;
11968        else
11969            task_obj := resolve2(task1, task_type) ;
11970        end if ;
11971    end if ;
11972
11973    if is_identifier(task_obj) and nature(task_obj) = 'task_type'
11974    then
11975        -- This is a reference to the task currently executing the body.
11976        -- replace the name of the task type by its run-time identity.
11977
```

```
11978          if task_obj ∈ open_scopes then
11979              task_obj := declared(task_obj)('current_task') ;
11980
11981          else
11982              errmsg('Invalid task type in ABORT statement','','9.10') ;
11983          end if ;
11984--
11985      end if ;
11986
11987      tasks with:= task_obj ;
11988end for ;
11989
11990return ['abort', tasks] ;
11991
11992end proc abort_statement ;
11993
```

10. Program structure and Compilation issues.

11995

init_compunit

```
11997 proc init_compunit;
11998     -- Initialize global maps for a new compilation unit.
11999
12000     comp_units := 0;
12001     NEWTYPES with:= [ ];
12002     all_vis := {} ;
12003     return;
12004
12005 end proc init_compunit;
12006
```

new_compunit

```
12008 proc new_compunit(typ, name);
12009
12010  -- Establish global name and library name for new compilation unit.
12011  -- If this (sub)unit was previously compiled, all stubs of it must
12012  -- be dropped. For these stubs, the current unit is either the top
12013  -- ancestor, or an intermediate one (if the current unit is itself
12014  -- a subunit).
12015  -- A stub is uniquely named by the tuple of all his ancestors.
12016
12017  loop
12018     for stub ∈ domain LIB_STUB | name ∈ stub(3..)
12019  do
12020     LIB_STUB(stub) := om;
12021  end loop;
12022
12023
12024  if is_comp_unit then
12025
12026     unit_type := typ;
12027     unit_name := [unit_type, name];
12028
12029     if unit_name ∈ PREDEF_UNITS then          -- Overrides predefined .
12030         PREDEF_UNITS less:= unit_name;
12031     end if;
12032
12033     establish_context(name) ;
```

```
12034
12035   end if ;
12036
12037   return unit_name;
12038
12039end proc new_compunit;
```

compunit

```
12041proc compunit(u);
12042
12043-- Collect global maps and library information after completion of
12044-- a compilation unit. If no errors were detected in the unit, add it
12045-- to program library.
12046
12047comp_units +:= 1;
12048t_def := new_type_list() ;
12049
12050if (ERRCOUNT = PRE_ERRCOUNT) then          -- No errors detected.
12051    vis_units := {v ∈ all_vis | v(2..) /= unit_name(3..)} ;
12052    PRE_COMP(unit_name) := vis_units ;
12053
12054    (forall v ∈ vis_units)
12055        versions_used(v) :=  if COMP_DATE(v) = om then om
12056                                    else COMP_DATE(v)(v) end;
12057    end forall;
12058
12059    CODE_UNITS(unit_name) :=  u ;
12060    UPDATE_LIB_MAPS(unit_name, 'unit');
12061    time_now := date;
12062    COMP_DATE(unit_name) := {};
12063    (forall vers_date = versions_used(vers))
12064        COMP_DATE(unit_name)(vers) := vers_date;
12065    end forall;
12066    COMP_DATE(unit_name)(unit_name) :=
12067        [time_now(9..17), time_now(20..27), comp_units];
12068    if UNIT_DECL(unit_name) = om then
12069      UNIT_DECL(unit_name) := [ ] ;
12070    end if ;
12071    UNIT_DECL(unit_name) with:= CONTEXT;
12072
12073else
12074    TO_LIST('   ');
12075    [kind, nam] := unit_name ;
12076    TO_LIST(' *** WARNING: error(s) detected in ' + kind + ' ' + nam +
12077                    '. Unit not inserted in library.');
12078end if ;
12079
12080-- Reinitialize compilation environment.
```

```
12081unit_name := ['','];
12082VISIBLE := {};
12083all_vis := {} ;
12084versions_used := {};
12085NEWTYPES with:= [ ];
12086DECLARED := BASE_DECLARED;
12087VISIBLE('ASCII') := DECLARED('ASCII');
12088--
12089scope_info := init_scope_info ;
12090scope_st  := [ ] ;
12091
12092if REF_OPTION then
12093    PRINT_XREF() ;
12094end if ;
12095
12096end proc compunit;
```

unit_symbtab

```
12098proc unit_symbtab(unit_unam) ;
12099
12100-- Collect symbol table entries for all entities declared in a compila-
12101-- tion unit, including inner units and blocks. We iterate over the
12102-- symbol table, and save all objects that are declared in the unit and
12103-- in inner scopes. For non-generic package bodies, we omit the decl-
12104-- arations that appear in the visible part, and are already saved with
12105-- the package spec.
12106
12107symb_map := {} ;
12108ignore := if nature(unit_unam) = 'package'
12109          then range visible(unit_unam)
12110        else {} end ;
12111
12112seen := scopes := {unit_unam} ;
12113
12114(while scopes /= {})
12115    sc from scopes ;
12116    (forall [-, u_name] ∈ declared(sc))
12117        if u_name  ignore then                    -- save its info.
12118          symb_map(u_name) := SYMBTABF(u_name) ;
12119        end if ;
12120
12121        if declared(u_name) /= om and u_name  seen then
12122            -- collect local declarations of inner scope.
12123            scopes with:= u_name ;
12124            seen  with:= u_name ;
12125        end if ;
12126    end forall ;
12127end while ;
```

```
12128
12129-- We include in symb_map the information for the unit itself, which is
12130-- declared in STANDARD.
12131symb_map(unit_unam) := SYMBTABF(unit_unam) ;
12132
12133return symb_map ;
12134
12135end proc unit_symbtab;
```

predef_scopes

```
12137proc predef_scopes ;
12138
12139-- Return the set of names of predefined scopes, whose declarations do
12140-- not need to be saved in library files. These are STANDARD, ASCII, and
12141-- those predefined packages that have not been redefined.
12142
12143p_s := {'STANDARD#0', 'UNMENTIONABLE#0', 'ASCII'}
12144        + {u(2): u ∈ PREDEF_UNITS | u(1) ∈ ['spec', 'subprog spec']} ;
12145return p_s ;
12146
12147end proc predef_scopes ;
```

save_subprog_info

```
12149proc save_subprog_info ;
12150
12151-- Save declarations for a subprogram specification or body which is a
12152-- compilation unit.
12153
12154if is_comp_unit then
12155
12156   unit := unit_name(2);
12157   unit_unam := declared('STANDARD#0')(unit);
12158   if unit_unam = om then
12159        errmsg('Invalid compilation unit', '', 'none') ;
12160       return ;
12161   end if ;
12162   -- For subprograms, UNIT_DECL has 4 fields:
12163   --    1. unique name of subprogram
12164   --    2. symbol table entries
12165   --    3. declared map for subprogram's scope
12166   --    4. context (supplied in compunit)
12167
12168   UNIT_DECL(unit_name) :=
12169        [unit_unam, unit_symbtab(unit_unam), declared(unit_unam) ? {}];
12170end if ;
12171
12172end proc save_subprog_info ;
```

save_spec_info

```
12174proc save_spec_info(unit_unam, old_vis) ;
12175
12176-- Build UNIT_DECL for a package spec. that is a compilation unit.
12177
12178if nature(unit_unam) = 'generic_package_spec' then
12179    -- save name within its own declarations, to simplify retrieval at
12180    -- instantiation time.
12181    declared(unit_unam)(original_name(unit_unam)) := unit_unam ;
12182end if ;
12183
12184-- For package specifications, UNIT_DECL has 6 fields
12185--        1. unique name of compilation unit
12186--        2. symbol table entries
12187--        3. declared maps for program defined scopes
12188--        4. vis_mods
12189--        5. difference between declared and visible
12190--        6. context (supplied in comp_unit)
12191
12192decmap := {[sn, dsn] : dsn = declared(sn) | sn   predef_scopes()} ;
12193
12194-- Notvis keeps track of things declared but not visible
12195notvis := {[sn, {dec : dec ∈ dsn | dec   (visible(sn) ?{}) }] :
12196                dsn = decmap(sn) };
12197
12198UNIT_DECL(unit_name) :=
12199        [unit_unam, unit_symbtab(unit_unam), decmap, old_vis, notvis];
12200
12201end proc save_spec_info ;
```

save_body_info

```
12203proc save_body_info(nam) ;
12204
12205-- For a package body, only the  symbol table  information  needs to be
12206-- saved, for purposes of generic instantiation. Visibility information
12207-- is not kept.
12208
12209if is_comp_unit then
12210    UNIT_DECL(unit_name) := [nam, unit_symbtab(nam)] ;
12211end if ;
12212
12213end proc save_body_info ;
```

establish_context

```
12215proc establish_context(name);
12216
12217-- Called after seeing the name of a compilation unit, this procedure
12218-- processes the unit's context specification (and if the unit is a body
12219-- or a subunit, any inherited context specification).
12220
12221if not is_comp_unit then return; end if;
12222
12223-- Get any inherited context specification.
12224--
12225save_node := current_node ;
12226current_node frome CONTEXT ;                    -- For error reporting
12227
12228if separate then
12229    check_separate(name);
12230else
12231    if unit_type = 'body' then -- package body, retrieve spec
12232        get_specs(name);
12233        spec_decl := UNIT_DECL(['spec', name]) ? [ ];
12234        CONTEXT +:= (spec_decl(6) ? [ ]);
12235    elseif unit_type = 'subprog' then -- see if there was a spec
12236
12237        spec_name := ['subprog spec', name];
12238        if RETRIEVE(spec_name) then
12239            versions_used(spec_name) :=
12240                if COMP_DATE(spec_name) = om then om
12241                    else COMP_DATE(spec_name)(spec_name) end;
12242        end if;
12243
12244        spec_decl := UNIT_DECL(spec_name) ? [ ];
12245        CONTEXT +:= (spec_decl(4) ? [ ]);
12246    end if;
12247
12248end if;
12249
12250-- Option to add with TEXT_IO; use TEXT_IO; to context
12251
12252if USETIO then
12253        CONTEXT(1) := [['with', ['TEXT_IO']], ['use', ['TEXT_IO']]]
12254                + (CONTEXT(1) ? [ ]);
12255end if;
12256
12257
12258-- Process the complete context specification (including inherited specs)
12259
12260lib_units := (domain LIB_UNIT) + {unit ∈ PREDEF_UNITS |
```

```
12261  unit(1) /= 'subprog' or
12262        (unit(1) = 'subprog' and not exists other ∈ PREDEF_UNITS |
12263           other = (['subprog spec'] + unit(2..)) ) };
12264
12265
12266-- The flag -current- distinguishes between the context clause attached
12267-- the current (sub)unit, and the clauses that appeared in its spec. or
12268-- in its parent unit.
12269current := true;
12270
12271(forall clause_list ∈ CONTEXT)
12272
12273   (forall [clause, nam_list] ∈ clause_list | clause = 'with')
12274        (forall nam ∈ nam_list)
12275           if is_tuple(nam) then        -- Qualified name.
12276              nam := find_old(nam) ;
12277           end if ;
12278
12279           if (exists unit ∈ lib_units |
12280              unit = ['subprog spec', nam] or
12281              unit = ['subprog instance', nam] or
12282              unit = ['spec instance', nam] or
12283              unit = ['spec', nam])
12284             or exists unit ∈ lib_units |
12285                    unit = ['subprog', nam] -- prefer 'subprog spec'
12286           then
12287              if current then
12288                 all_vis with:= unit;
12289              end if;
12290
12291              if unit(1) ∈ {'spec', 'spec instance'}
12292              then
12293                 -- avoid multiple retrievals.
12294                 if nam ∈ vis_mods and not current then  --
12295                    continue forall ;
12296                 end if ;
12297                 get_specs(nam);
12298                 if nature(nam) /= 'generic_package' then
12299                    vis_mods with:= nam;
12300                 end if ;
12301
12302              else     -- unit(1) = subprog... or instance
12303                 if RETRIEVE(unit) then
12304                    [unit_unam, s_info, decls] := UNIT_DECL(unit) ;
12305
12306                    (for [unam, entry] ∈ s_info)
12307                          SYMBTABF(unam) := entry;--
12308                    end for;
12309                    DECLARED(unit_unam) := decls;
```

```
12310                    end if;
12311
12312                    DECLARED('STANDARD#0')(nam) := unit_unam ; --
12313                end if;
12314            else
12315                if current then
12316                    all_vis with:= ['spec', nam];
12317                end if;
12318
12319                errmsg('Unknown unit in with clause: ',nam ,'10.1.1');
12320
12321                if check_old(nam) = '?' then -- safe to add it.
12322                    n := find_new(nam) ;          -- To avoid error
12323                    visible(n) := {} ;            -- in subsequent USE
12324                end if ;
12325
12326            end if;
12327        end forall;
12328    end forall;
12329--
12330-- Process USE clauses of current context.
12331    if current then
12332        (forall [clause, nam_list] ∈ clause_list )              --
12333            if clause = 'use' then
12334              use_clause(nam_list) ;
12335            elseif clause = 'elaborate' then
12336                elaborate_pragma(nam_list) ;
12337            end if ;                                            --
12338        end forall;
12339    end if ;
12340--
12341    current := false;
12342end forall;
12343
12344current_node := save_node ;                          --
12345end proc establish_context;
```

elaborate_pragma

```
12347proc elaborate_pragma(nam_list) ;
12348
12349-- The names of the units in the elaborate list must have appeared in the
12350-- current context clause. the unit names of the corresponding bodies are
12351-- added to the set all_vis.
12352
12353(for nam ∈ nam_list)
12354    if ['spec', nam] ∈ all_vis then
12355        all_vis with:= ['body', nam] ;              -- package body needed.
12356    elseif ['subprog spec', nam] ∈ all_vis then
```

```
12357         all_vis with:= ['subprog', nam] ;      -- subprogam body needed.
12358     elseif ['subprog',nam] ∈ all_vis then      -- already listed.
12359         pass ;
12360     else
12361         warning('unknown unit name in ELABORATE pragma: '+ nam) ;
12362     end if ;
12363end for ;
12364end proc elaborate_pragma ;
```

stub_head

```
12366
12367proc stub_head(nat, id) ;
12368
12369-- Find unique name of package or task stub, and verify that it occurs
12370-- in the proper scope.
12371
12372stub_name := find_old(id) ;
12373
12374if scope_of(stub_name) /= scope_name then
12375    errmsg('specification and stub for '+ id,
12376                        'are in different scopes', '7.1, 9.1') ;
12377end if ;
12378
12379-- Nature of specification must match that of stub.
12380
12381if  nat = 'package' and nature(stub_name)
12382                            {'package_spec','generic_package_spec'}
12383or  nat = 'task'    and nature(stub_name)
12384                            {'task_type_spec', 'task_obj_spec'}
12385then
12386    errmsg('Matching specification not found for stub ', id, '7.1, 9.1');
12387    declared(stub_name) ?:= {} ;
12388end if;
12389
12390return stub_name ;
12391
12392end proc stub_head ;
12393
```

save_stub

```
12395proc save_stub(kind,name) ;
12396
12397-- Save current state of compilation : scope stack and related declared
12398-- maps, for a subprogram or module stub.
12399
12400
12401-- In order to uniquely identify the stub, we create for it a name which
```

```
12402-- includes the names of all surrounding scopes, with the exception of
12403-- the ever-present standard environment and its enclosing scope.
12404
12405defined_scopes := [name] + open_scopes(1..#open_scopes-2) ;
12406stub_name := [kind] + [original_name(d_s) : d_s ∈ defined_scopes] ;
12407
12408-- Ada requires that the identifiers of all subunits of a given library
12409-- unit (as well as the name of the library unit itself) be unique.
12410
12411if exists other_unit ∈ domain LIB_STUB |                    --
12412    other_unit(2) = stub_name(2) and          -- have same identifier
12413    other_unit(#other_unit) = stub_name(#stub_name) -- same eldest ancestor
12414then
12415    errmsg('Subunit identifier not unique: ', original_name(name),
12416                                                '10.2');
12417end if;
12418--
12419-- Verify that the stub appears immediately within a compilation unit.
12420if original_name(scope_name) /= unit_name(2) then
12421    errmsg( 'stubs can only appear in the outermost scope of a ',
12422                'compilation unit', '10.2') ;
12423end if ;
12424--
12425
12426UPDATE_LIB_MAPS(stub_name, 'stub');
12427
12428-- Save stub environment.
12429open_decls := {};
12430(forall decl = declared(os) | os  predef_scopes())
12431    open_decls(os) := {[nam, decl(nam), SYMBTABF(decl(nam))] :
12432                                nam ∈ domain decl};
12433end forall;
12434unit_unam := declared('STANDARD#0')(stub_name(#stub_name));
12435STUB_ENV(stub_name) :=
12436    [ (scope_st + [scope_info]),
12437      defined_scopes,
12438      open_decls,
12439      {[vm, visible(vm)] : vm ∈ vis_mods | vm  predef_scopes()},
12440      unit_unam,
12441      SYMBTABF(unit_unam),
12442      CONTEXT
12443     ] ;
12444
12445return  [['SEPARATE', stub_name]] ;
12446
12447end proc save_stub ;
12448
12449
```

separate_context

```
12451proc separate_context(name) ;
12452-- Process the name of the enclosing unit of a subunit. The result is
12453-- left in global variable ENCLOSING_SCOPE, for processing when the full
12454-- context is established.
12455
12456SEPARATE := true ;
12457ENCLOSING_SCOPE:= [ ] ;
12458(while is_tuple(name))
12459   ENCLOSING_SCOPE with:= name(3) ;
12460   name := name(2) ;
12461end ;
12462ENCLOSING_SCOPE with:= name ;
12463
12464end proc separate_context ;
12465
12466
```

check_separate

```
12468proc check_separate(name);
12469-- This procedure restores the environment saved for a stub,
12470-- including the original scope stack.
12471
12472        if separate then
12473            u_decl, u_symbtab, old_con] := RETRIEVE_ENV(name) ;
12474           if unit_name /= [ ] then
12475               CONTEXT +:= old_con;
12476               if exists parent_unit ∈ domain LIB_UNIT |
12477                  parent_unit(1) ∈ {'subprog', 'body'} and
12478                  parent_unit(2..) = unit_name(3..)
12479               then
12480                  RETRIEVE(parent_unit);
12481                  versions_used(parent_unit) :=
12482                       if COMP_DATE(parent_unit) = om then om
12483                       else COMP_DATE(parent_unit)(parent_unit) end;
12484                  -- put name of outer-most scope in standard
12485                  outer_most := parent_unit(#parent_unit);
12486                  DECLARED('STANDARD#0')(outer_most) := u_decl;
12487                  SYMBTABF(u_decl) := u_symbtab;
12488               end if;
12489               popscope();
12490               -- Initialize visibility info.
12491               (forall vis_vm = vis(vm))
12492                  visible(vm) := vis_vm;
12493                  declared(vm) := vis_vm;
12494               end forall;
```

```
12495                (forall decls = decl(os))
12496                   scope_decl := {};
12497                   (forall [nam, u_nam, info] ∈ decls)
12498                        scope_decl(nam) := u_nam;
12499                        SYMBTABF(u_nam) := info;
12500                   end forall;
12501                   declared(os) := scope_decl;
12502                end forall;
12503             else
12504                unit_name := ['', ''];
12505             end if;
12506          end if ;
12507          SEPARATE := false ;
12508
12509end proc check_separate;
12510
```

retrieve_env

```
12512proc RETRIEVE_ENV(name) ;
12513-- retrieve from library the environment in which a stub was
12514-- first seen.
12515
12516stub_names := {stub_nam : stub_nam ∈ domain LIB_STUB |
12517                stub_nam(2) = original_name(name) and
12518                stub_nam(3..) = ENCLOSING_SCOPE and
12519                stub_nam(1) ∈ {'subprog', 'body'} };
12520
12521if #stub_names /= 1 then
12522        errmsg('cannot find stub for subunit ', name,'10.2') ;
12523        return [ [ ], [ ], [ ], [ ], [ ], [ ], [ ], [ ] ] ;
12524end if ;
12525stub_name := arb stub_names ;
12526if STUB_RETRIEVE(stub_name) = false then
12527        errmsg('cannot find stub for subunit ',name,'10.2') ;
12528        return [ [ ], [ ], [ ], [ ], [ ], [ ], [ ], [ ] ] ;
12529end if;
12530return [stub_name] + STUB_ENV(stub_name) ;
12531
12532end proc RETRIEVE_ENV ;
12533
```

11. Exceptions

except_decl

```
12536proc except_decl(id_list) ;
12537nam_list := [ ] ;
12538(forall id ∈ id_list)
12539
12540    name := find_new(id) ;
12541    SYMBTAB(name) := ['exception', 'exception'] ;
12542    nam_list with:= name ;
12543end forall ;
12544
12545return [['exception_decl', nam_list]] ;
12546
12547end proc except_decl ;
```

exception_part

```
12549
12550proc exception_part(node) ;
12551
12552-- A scope is established for the exception handlers. This scope
12553-- or a block nested within it, are the only valid scopes for the
12554-- occurence of a non-specific RAISE statement.
12555
12556handler := find_new(str newat) ;
12557newscope(handler) ;
12558SYMBTAB(handler) := ['block', 'handler', [ ]] ;
12559
12560-- Process individual handlers.
12561[-, handler_list] := AST(node) ;
12562handlers := [adasem(h) : h ∈ handler_list] ;
12563ex_names := signature(scope_name) ;
12564popscope() ;
12565
12566if 'others' ∈ ex_names and top(ex_names) /= 'others' then
12567    errmsg('OTHERS must appear last', ' in handler', '11.2') ;
12568end if ;
12569
12570return [['exception', handlers]] ;
12571
12572end proc exception_part ;
12573
```

12574

exception_handler

```
12576proc exception_handler(node) ;
12577
12578
12579[-, exception_list, statements_node] := AST(node) ;
12580
12581ex_list := {} ;
12582(forall id_node ∈ exception_list)
12583    current_node := id_node ;
12584    id := adasem(id_node) ;
12585
12586    if id /= 'others' then
12587
12588        except := find_old(id) ;
12589        if nature(except) /= 'exception' then
12590            errmsg(original_name(except), ' is not an exception',
12591                                                    '11.1') ;
12592        elseif except ∈ signature(scope_name) then
12593            errmsg('duplicate exception name in handler','','11.2') ;
12594        else
12595            ex_list with:= except ;
12596            signature(scope_name) with:= except ;
12597        end if ;
12598    else
12599
12600        if 'others' ∈ signature(scope_name) then
12601            errmsg('Duplicate OTHERS in exception part','','11.2') ;
12602        elseif # exception_list /= 1 then
12603            errmsg('OTHERS must appear alone in exception list', '',
12604                                                    '11.2');
12605        else
12606            ex_list with:= 'others' ;
12607            signature(scope_name) with:= 'others' ;
12608        end if ;
12609
12610    end if ;
12611
12612end forall ;
12613
12614stmts := adasem(statements_node) ;
12615
12616return  [ex_list, stmts] ;
12617
12618end proc exception_handler ;
12619
```

raise_statement

```
12621
12622proc raise_statement(opt_id) ;
12623
12624if is_empty(opt_id) then
12625    -- Non-specific raise. This statement form can appear only within
12626    -- an exception handler.
12627
12628    assert exists scope ∈ open_scopes | nature(scope) /= 'block' or
12629                                        type_of(scope) = 'handler' ;
12630    if type_of(scope) /= 'handler' then
12631        errmsg('RAISE statement not directly in exception handler',
12632                                                  '', '11.3') ;
12633    end if ;
12634
12635    return ['raise'] ;
12636
12637else
12638    except := find_old(opt_id) ;
12639    if nature(except) /= 'exception' and type_of(except) /= 'any' then
12640        errmsg(opt_id, 'is not an exception', '11.1') ;
12641    end if ;
12642
12643    return ['raise', except] ;
12644end if ;
12645
12646end proc raise_statement ;
12647
12648
```

12. Generic Program units.

generic_subprog_spec

```
12651proc generic_subprog_spec(node) ;
12652
12653
12654
12655-- Build specifications of a generic subprogram. We create a scope for
12656-- it, and define within the names of generics and formal parameters.
12657-- The signature of the generic subprogram includes the generic parameter
12658-- list and the formals. These two are unpacked during instantiation.
12659
12660[kind, id_node, generic_part_node, formals_list, ret_node] := AST(node);
12661
12662    obj_id := adasem(id_node) ;
12663
12664    new_compunit('subprog spec', obj_id) ;
12665
12666    gen_name := find_new(obj_id) ;
12667    declared(gen_name) := {} ;
12668    nature( gen_name) := 'generic_part' ;
12669    newscope(gen_name) ;
12670
12671    [gen_list, gen_code] := adasem(generic_part_node) ;
12672
12673    -- Now declared(gen_name) contains the generic parameters:
12674    -- types, objects and subprograms.
12675
12676    formals := get_formals([adasem(f): f ∈ formals_list]) ;
12677
12678    -- For the formal parameters, we simply must recognize their names
12679    -- and types. Type checking on initialization is repeated on
12680    -- instantiation.
12681
12682    (forall [id ,in_out,type_mark, opt_init] = formals(i))
12683
12684        form_name := find_new(id) ;
12685          NATURE(form_name)  := in_out ;                    --
12686          TYPE_OF(form_name) := type_mark ;
12687
12688        if opt_init /= [ ] then
12689            opt_init := bind_names(opt_init) ;
12690              formals(i)(4) := opt_init ;
```

```
12691         end if ;
12692
12693     end forall ;
12694
12695     -- Save signature of generic object, in the format which the
12696     -- instantiation procedure requires.
12697
12698     NATURE(gen_name) := kind + '_spec' ;
12699     SIGNATURE(gen_name) := [[gen_list, gen_code], formals] ;
12700     TYPE_OF(gen_name)  := if kind = 'generic_function' then
12701                             find_old(adasem(ret_node)) else 'none' end ;
12702
12703
12704     popscope() ;
12705
12706     -- No code is emitted for a generic specification, but if it is a
12707     -- compilation unit the library must be updated.
12708     save_subprog_info ;
12709
12710     return [ ] ;
12711
12712
12713end proc generic_subprog_spec ;
12714
```

generic_subprog_body

```
12716proc generic_subprog_body(prog_name, node) ;
12717
12718-- Within its body, the generic subprogram name behaves as a regular
12719-- (i.e. non-generic) subprogram. In particular, it can be called (and
12720-- it cannot be instantiated). Its nature must be set accordingly, prior
12721-- to compilation of the body.
12722
12723
12724
12725new_nat :=  if nature(prog_name) = 'generic_procedure_spec'
12726                  then 'procedure'  else 'function'   end ;
12727
12728nature(prog_name) := new_nat ;
12729
12730nat := 'generic_' + new_nat ;          -- Save till end of body.
12731
12732-- The signature of a generic object includes the generic part. During
12733-- compilation of the body, set the signature to contain only the formals
12734
12735sig := signature(prog_name) ;
12736signature(prog_name) := sig(2) ;
12737overloads(prog_name) := {prog_name} ;
```

```
12738
12739-- TBSL :agreement of names, subtypes and default expressions.
12740
12741newscope(prog_name) ;
12742prefix +:= original_name(prog_name) + '.'; -- For names of local objects.
12743
12744[-, -, decl_node, stats_node, except_node] := AST(node) ;
12745body :=
12746     block(adasem(decl_node), adasem(stats_node), adasem(except_node));
12747
12748lab_end() ;                    -- Validate goto statements in subprogam.
12749popscope() ;                   -- And exit body.
12750
12751-- Outside of its body, the object is generic again.
12752
12753nature(prog_name) := nat ;
12754signature(prog_name) := sig with body ;      -- Save for instantiation
12755overloads(prog_name) := om ;                       -- Not a callable object.
12756
12757-- If the spec for  this generic subprogram was defined in another
12758-- compilation unit, it must  be updated  (see module_body).  This
12759-- places the restriction on generic units that spec and body must
12760-- be defined in the same compilation (source file). Ideally the
12761-- librarian will know to look for the updated specs of a generic
12762-- object in the library entry for the body. Currently there is no
12763-- such.
12764
12765if is_comp_unit then
12766    unit_kind := 'subprog spec' ;
12767elseif unit_name(1) = 'body' and #unit_name = 2 then
12768    unit_kind := 'spec' ;
12769else
12770    unit_kind := om ;                              -- Just a sub-unit.
12771end if ;
12772
12773if unit_kind /= om then
12774    spec_name := [unit_kind, unit_name(2)] ;
12775
12776    if LIB_UNIT(spec_name) = AISFILE              -- ie current compilation
12777      and       #(UNIT_DECL(spec_name) ? [ ]) > 0 then
12778        -- update nature, signature, etc. for generic object, and
12779        -- for its generic types.
12780        [-, specs, -] := UNIT_DECL(spec_name) ;  --
12781        specs(prog_name) := SYMBTABF(prog_name) ;
12782
12783         (forall [-, t_name] ∈ declared(prog_name)
12784                                    | is_generic_type(t_name))
12785           specs(t_name) := SYMBTABF(t_name) ;
12786         end ;
```

```
12787
12788          UNIT_DECL(spec_name)(2) := specs ;                --
12789     elseif is_comp_unit then
12790          -- save it as if it were a new subprogram specification. This
12791          -- will replace the previous copy in the library.
12792          unit_name := [unit_kind, original_name(prog_name)] ;
12793          save_subprog_info ;
12794     end if ;
12795end if ;
12796
12797return [ ] ;              -- No code is generated for a generic subprogram.
12798
12799end proc generic_subprog_body ;
12800
```

generic_pack_spec

```
12802proc generic_pack_spec(node) ;
12803
12804
12805
12806
12807[-, id_node, generic_part_node, decls_node, opt_priv_node] := AST(node);
12808
12809id := adasem(id_node) ;
12810
12811new_compunit('spec', id);
12812
12813mod_name := prefix + id ;              -- Standard naming con-
12814                                                -- vention for packages.
12815declared(mod_name) := {} ;             -- To receive generic def
12816if declared(scope_name)(id) /= om then
12817     errmsg('Duplicate declaration of ', id,'8.3') ;
12818end if ;
12819declared(scope_name)(id) := mod_name ;
12820scope_of(mod_name) := scope_name;
12821nature(mod_name)   := 'generic_part' ;
12822TO_XREF(mod_name) ;
12823
12824newscope(mod_name) ;
12825-- Proeess generic parameters. Their definition will appear in
12826-- the scope of the generic package. The list of them is also
12827-- preserved in the signature of the package, for instantiation.
12828-- The signature of the generic package as the format:
12829--
12830--    [ [generic_type_list, anonymous_types], specification,  body]
12831--
12832
12833signature(mod_name) := [adasem(generic_part_node)] ;
```

```
12834prefix := prefix + id + '.' ;  -- For local names
12835
12836nature(scope_name)  := 'generic_package_spec' ;
12837type_of(scope_name) := 'none' ;
12838-- Create marker to save use declarations.
12839find_new('$used') ;
12840use_declarations(scope_name) := [ ] ;
12841private_decls(scope_name) := {} ;
12842
12843-- The rest of the package is processed as in a non-generic case.
12844
12845visible_decls := adasem(decls_node) ;
12846-- The declarations so far constitute the visible part of the package
12847
12848visible(scope_name) := declared(scope_name) ;
12849
12850-- Now process private part of package.
12851
12852newscope(str newat) ;
12853nature(scope_name) := 'private_part' ;
12854priv_decls := end_private(adasem(opt_priv_node)) ;
12855specs := visible_decls + priv_decls ;
12856
12857signature(scope_name) +:= [specs] ;
12858
12859end_specs(['generic_package', scope_name], specs) ;
12860
12861return [ ] ;
12862
12863end proc generic_pack_spec ;
12864
12865
```

generic_obj_decl

```
12867proc generic_obj_decl(node) ;
12868
12869
12870
12871[-, id_list_node, in_out_node, type_node, exp_node] := AST(node) ;
12872
12873id_list  := [adasem(id_node): id_node ∈ id_list_node] ;
12874nam_list := [find_new(id) : id ∈ id_list] ;
12875
12876kind     := adasem(in_out_node) ;
12877if kind = '' then kind := 'in' ; end if ;        -- implicit "in"
12878
12879type_mark := find_old(adasem(type_node)) ;
12880check_incomplete(type_mark) ;
```

```
12881
12882opt_init := adasem(exp_node) ;
12883
12884case kind of
12885
12886   ('in'):
12887      if is_limited_type(type_mark) then
12888          errmsg('Type of a generic formal object of mode IN ',
12889                          'must not be a limited type', '12.1.1') ;
12890      end if ;
12891
12892      if opt_init /= [ ] then
12893
12894          -- Type check  default value, and bind identifiers to
12895          -- visible objects, but do not save resolved
12896          -- expression: it may depend on generic parameters,
12897          -- and will be substituted at instantiation  time.
12898
12899          opt_init := bind_names(opt_init) ;
12900          e := check_type(type_mark, opt_init) ;            --
12901          if is_deferred_constant(e) then
12902              errmsg('Deferred constant cannot be default expression',
12903                      ' for a generic parameter','7.4.3') ;
12904          end if ;
12905      end if;
12906
12907      (forall name ∈ nam_list)
12908          nature(name)  := 'constant' ;
12909          type_of(name) := type_mark ;
12910      end forall ;
12911
12912   ('inout'):
12913        -- No constraints apply to generic inout formals.
12914        type_mark := base_type(type_mark) ;
12915
12916        if opt_init /= [ ] then
12917          errmsg('Initialization not allowed for IN OUT ',
12918                                  'generic parameters','12.1.1') ;
12919        end if ;
12920
12921      (forall name ∈ nam_list)
12922          nature(name)  := 'obj' ;
12923          type_of(name) := type_mark ;
12924      end forall ;
12925
12926   ('out'):
12927      current_node := in_out_node ;
12928      errmsg('OUT generic formals objects not allowed','', '12.1.1') ;
12929
```

```
12930end case ;
12931
12932return [[name, kind, type_mark, opt_init]: name ∈ nam_list] ;
12933end proc generic_obj_decl ;
12934
```

generic_type_decl

```
12936proc generic_type_decl(node) ;
12937
12938
12939
12940[-, id_node, def_node] := AST(node) ;
12941id := adasem(id_node) ;
12942
12943-- In the case of generic array types, anonymous parent array may be
12944-- introduced. They are not generic in themselves, and play no role in
12945-- the instantiated code; they are collected here and  discarded.
12946
12947NEWTYPES with:= [ ] ;
12948if node_kind(def_node) = 'generic' then              --scalar type
12949        type_name := find_new(id) ;
12950        [-, root] := adasem(def_node) ;
12951--
12952        -- A generic signature must be constructed for these types, in
12953        -- order  to verify bounds  in instantiations,  subtypes,  etc.
12954        -- These bounds must expressed by means of attributes.
12955
12956        if root ∈ {'INTEGER', 'discrete_type'} then
12957            lo := ["", 'T_FIRST', type_name, 'I:1'] ;
12958            hi := ["", 'T_LAST' , type_name, 'I:1'] ;
12959            bounds := ['range', lo, hi] ;
12960      else
12961            [kind, lo, hi, precision] := signature(root) ;
12962            attr := if kind = 'digits' then 'DIGITS' else 'DELTA' end ;
12963            precision := ["", attr, type_mark] ;
12964            bounds := [kind, lo, hi, precision] ;
12965        end if ;
12966
12967        -- The base type of a generic type is the base of its actual. In
12968      - -- order to be able to refer to the base type of a generic within
12969        -- the object, we introduce an anonymous type that will be instan
12970        -- tiated with the base type of the actual.
12971
12972        generic_base := anonymous_type() ;
12973        SYMBTAB(generic_base) := ['type', root, bounds] ;
12974      root_type(generic_base) := root_type(root) ;
12975        misc_type_attributes(generic_base) := {'$generic'} ;
12976
```

```
12977         SYMBTAB(type_name) := ['subtype', generic_base, bounds] ;
12978         root_type(type_name) := root_type(root) ;
12979else                                  -- array type or access type.
12980    type_name := type_decl(id, [ ], def_node) ;
12981end if ;
12982
12983misc_type_attributes(type_name) := (misc_type_attributes(type_name) ?{})
12984                                       with '$generic' ;              --
12985
12986anon_typ frome NEWTYPES ;
12987
12988return [[type_name, 'type', [ ]]] ;
12989
12990end proc generic_type_decl ;
```

is_generic_type

```
12992proc is_generic_type(type_mark) ;
12993
12994
12995
12996attr := misc_type_attributes(type_mark) ;
12997return (is_set(attr) and '$generic' ∈ attr) ;
12998
12999end proc is_generic_type ;
13000
```

generic_priv_decl

```
13002proc generic_priv_decl(node) ;
13003
13004
13005
13006
13007[-, id_node, discr_list, kind] := AST(node) ;
13008id := adasem(id_node) ;
13009discriminants := [adasem(d) : d ∈ discr_list] ;
13010type_name := private_decl(id, discriminants, kind) ;
13011if type_of(type_name) /= kind then
13012    -- error in type declaration, or duplicate name.
13013    return [ ] ;
13014end if ;
13015
13016misc_type_attributes(type_name) := {'$generic'} ;
13017
13018if exists discr ∈ discriminant_list(type_name)
13019                                    | default_expr(discr) /= om then
13020    errmsg('generic private type cannot have defaults for discriminants',
13021                                    '','12.1.2') ;
```

```
13022end if ;
13023
13024return [[type_name, 'type', [ ] ]] ;
13025
13026end proc generic_priv_decl ;
```

check_generic_usage

```
13028proc check_generic_usage(type_mark) ;
13029
13030-- if a private generic type, or a subtype or derived type of it, is used
13031-- in an object declaration, component declaration, or allocator,indicate
13032-- that it must be instantiated with a constrained type.
13033
13034
13035
13036t := root_type(type_mark) ;
13037
13038if type_of(t) ∈ priv_types and is_generic_type(t)
13039     and (can_constrain(type_mark) or                              --
13040     not has_discriminants(type_mark))
13041then
13042    misc_type_attributes(t) with:= '$constrain' ;
13043
13044end if ;
13045
13046end proc check_generic_usage ;
13047
```

generic_subp_decl

```
13049proc generic_subp_decl(node) ;
13050
13051
13052
13053        [-, spec_node, opt_is_node] := AST(node)  ;
13054        [proc_nat, id, sig, typ] := adasem(spec_node) ;
13055      if id = '/=' then
13056        errmsg(' explicit overloading of the inequality operator',
13057              ' is not allowed ','6.7,12.3.1');
13058        return [ ];
13059      end if;
13060      opt_default := adasem(opt_is_node)  ;
13061
13062      if opt_default = 'box' then
13063           opt_default := ['box', id] ;
13064
13065       elseif opt_default = [ ] then pass ;
13066
```

```
13067       elseif is_tuple(opt_default) and opt_default(1) = 'string' then
13068           -- default is an operator name.
13069         opt_default :=
13070              find_old(desig_to_name('"' + opt_default(2) + '"')) ;
13071
13072       else
13073          -- Names must be bound at the point of generic definition.
13074           opt_default := find_old(opt_default) ;
13075
13076        end if ;
13077
13078
13079        u_name := chain_overloads(id, [proc_nat, typ, sig]) ;
13080
13081        return [[u_name, proc_nat, [sig, typ], opt_default]] ;
13082
13083
13084end proc generic_subp_decl ;
13085
```

bind_names

```
13087proc bind_names(tree) ;
13088
13089-- Perform name resolution for default initializations for generic IN
13090-- parameters and for discriminant specifications.
13091
13092return
13093    if is_tuple(tree) then
13094       (if #tree = 2 and tree(1) = 'name' then
13095           (if is_identifier(tree(2)) then find_old(tree(2))
13096                else bind_names(find_old(tree(2))) end )
13097
13098       elseif tree(1) = 'call?' then bind_names(find_old(tree))
13099       elseif tree(1) = '.' then find_old(tree)
13100       else [ bind_names(c) : c ∈ tree ] end)
13101    else tree  end ;
13102
13103end proc bind_names ;
13104
```

subprog_instance

```
13106proc subprog_instance(kind, new_id, gen_name, instance) ;
13107
13108-- Create an instantiation of a generic procedure.
13109--
13110-- To construct  the new instance, we  first process the instantiation of
13111-- the  generics. This yields a series  of renames  statements, which map
```

```
13112-- the generic parameters into actual types and subprograms. We then copy
13113-- the body of the generic procedure, and insert within the renames list.
13114-- If the body has not been  seen, we emit an instance stub, to be filled
13115-- by the binder.
13116
13117
13118rmatch(kind, '_instance') ;                    -- drop 'instance' marker.
13119if not is_identifier(gen_name)
13120
13121or kind = 'procedure' and nature(gen_name)
13122                        {'generic_procedure', 'generic_procedure_spec'}
13123or kind = 'function'  and nature(gen_name)
13124                        {'generic_function', 'generic_function_spec'}
13125then
13126    errmsg(original_name(gen_name),
13127                        ' is not a generic ' + kind,'12.1, 12.3') ;
13128    return [ ] ;
13129end if ;
13130TO_XREF(gen_name) ;
13131
13132[generics, formals,  stats] := signature(gen_name) ;
13133return_type := type_of(gen_name) ;
13134
13135-- Now match generic specification with instantiation.
13136
13137[instantiation, type_map, must_constrain] :=
13138                        instantiate_generics(generics, instance) ;
13139
13140-- Use the instantiated generic types to obtain the actual signature and
13141-- return type of the new procedure.
13142
13143new_signature := replace(formals, type_map) ;
13144new_return    := replace(return_type, type_map) ;
13145
13146new_specs := [kind, new_return, new_signature] ;
13147
13148-- Create new overloadable object with these specs.
13149
13150proc_name := chain_overloads(new_id, new_specs) ;
13151
13152-- in the body of the procedure, replace the generic name with the
13153-- instantiated name. (it appears on the return statement, and of
13154-- course in any recursive call).
13155-- Also, map the names of the formals parameters into the names they
13156-- have in the instantiated procedure (the actual formals ?)
13157
13158newer := build_signature(proc_name);                    --
13159
13160(forall [n,-] = newer(i))
```

```
13161    [id,-] := new_signature(i) ;
13162    type_map(declared(gen_name)(id)) := n ;
13163end ;
13164type_map(gen_name) := proc_name ;
13165
13166-- Now replace generic parameters by their instantiations throughout.
13167
13168if stats ≠ om then
13169    stats := replace(stats, type_map) ;
13170
13171else
13172    -- Only the subprogram specification has been seen. Emit a stub for
13173    -- the binder.
13174    stats := [ 'subprogram_instance', gen_name, type_map, {},
13175                                must_constrain, spans(current_node)] ;
13176
13177end if ;
13178
13179save_subprog_info ;                                          --
13180
13181-- The instantiation code is emitted in front of the procedure, so
13182-- that it is elaborated in the instantiation environment.
13183
13184return instantiation +
13185    [ [ nature(proc_name), proc_name, newer,  stats ] ] ; --
13186
13187end proc subprog_instance ;
13188
```

package_instance

```
13190proc package_instance(pack_head, gen_name, instance) ;
13191
13192-- Create an instantiation of a generic package. -name- is the aggregate
13193-- which instantiates the generic  parameters. The specification and the
13194-- body of the generic package have been stored  in the signature of the
13195-- generic  object. We  instantiate the  generics, place  the  resulting
13196-- type  and subprogram definitions  in front of  the specification, and
13197-- emit the specification and the body as one block of code .
13198-- Generic instantiation of body is done as for procedure_instantiation.
13199
13200[-,package] := pack_head ;
13201
13202-- new_compunit will have already been called under the asssumption
13203-- that the current compilation unit is a non-generic package.  This
13204-- may be inefficient, but the second calls to new_compunit and
13205-- establish_context will act correctly.
13206new_compunit('spec instance', package);
13207
```

```
13208
13209if not is_identifier(gen_name) or
13210   nature(gen_name)  {'generic_package', 'generic_package_spec'}
13211then
13212   errmsg(original_name(gen_name), ' is not a generic package','12.1');
13213   popscope() ;
13214   return [ ] ;
13215
13216elseif gen_name ∈ open_scopes then
13217   errmsg('Recursive instantiation not allowed','','12.3') ;
13218   popscope() ;
13219   return [ ] ;
13220end if ;
13221TO_XREF(gen_name) ;
13222
13223[generics, p_specs, body] := signature(gen_name) ;
13224
13225-- Create the proper prefix for the unique names of instantiated objects
13226-- (This might be done by actually establishing a new package scope).
13227
13228o_pref := prefix ;
13229prefix := package + '.' ;
13230
13231[instantiation, type_map, must_constrain] :=
13232                        instantiate_generics(generics, instance) ;
13233
13234-- The visible part of the new instance, and the body of the new
13235-- instance, are obtained from the generic , by name replacement. We
13236-- replace the occurence of the generic name everywhere by the inst-
13237-- ance name. In this fashion, the unique names of the instance follow
13238-- the same formation rule as those of a normal package. (the name of
13239-- the generic is the prefix of all unique names of objects local to the
13240--  generic package) .
13241--  The generic types are replaced by their instances, using the map
13242--  -type_map- produced by the instantiate_generics procedure . New
13243-- 'locations' have to be created for instantiated 'in' parameters. On
13244-- the other hand, generic types, inout parameters and subprograms are
13245-- simply renamed. The discriminants of private types are mapped into
13246-- into the discriminant of the actuals as a renaming also.
13247
13248gen_list := generics(1) ;
13249
13250renamed_params := { n : [n,m,-,-] ∈ gen_list | m ≠ 'in'} +
13251   {gen_d : [gen_d, -] ∈ type_map | nature(gen_d) = 'discriminant'} ;
13252
13253-- An additional complication has to do with nested declarations(records,
13254-- other packages) within the generic object. For these we must also
13255-- create instances of their symbol tables, so that type checking of
13256-- their uses can be performed. We therefore traverse recursively all
```

```
13257-- nested declarations within the generic object, to collect every object
13258-- whose symbol table field must be instantiated. This may be done at
13259-- generic definition time, and will be more efficient than here. For
13260-- procedures and functions, only their signature is needed to perform
13261-- type-checking, and therefore no symbol table entries need be created
13262-- for ther formals and local variables.
13263
13264rename_map := type_map with [gen_name, package] ;
13265packs := {} ;                    -- to collect names of nested packages.
13266instantiated_scopes := {} ;      -- All of which have declared maps.
13267workpile := {[gen_name, package, prefix ]} ;
13268
13269(while workpile ≠ {})
13270
13271   [g_n, new_pn, pref] from workpile ;
13272   instantiated_scopes with:= g_n ;
13273
13274
13275   -- Iterate over all items declared in g_n, the generic object,
13276   -- (or any object nested within and which has declarations : package,
13277   -- record, task) and collect declarations for instantiated items.
13278
13279   old_decls := declared(g_n) ;
13280   new_decls := {} ;
13281
13282   (forall [id, old_n] ∈ old_decls )
13283
13284
13285      if old_n ∈ renamed_params then
13286         -- generic parameter which was renamed already. Whether
13287         -- it should be nameable from outside is an open question.
13288         -- Here we assume that it is, and place in declared.
13289         new_decls(id) := rename_map(old_n) ;
13290
13291       elseif (new_n := rename_map(old_n)) ≠ om then
13292         -- id renames an object which has been instantiated already.
13293         -- The instantiation of id will point to the instantiation of
13294         -- that object.
13295         renamed_params with:= old_n ;
13296         new_decls(id) := new_n ;
13297
13298       elseif scope_of(old_n) ≠ g_n then
13299         -- old_n is a renaming of some other entity, generic or other-
13300         -- wise, which is defined in some outer scope. The instantia-
13301         -- tion of old_n must rename the same entity.
13302         if (new_n := rename_map(old_n)) = om then
13303            new_n := rename_map(old_n) := old_n ;
13304         end if ;
13305         renamed_params with:= old_n ;
```

```
13306          new_decls(id) := new_n ;
13307
13308       else
13309          new_n := pref + id + str newat ;  -- New unique name
13310          rename_map(old_n) := new_n ;      -- map generic to actual.
13311           -- Create entry in declared for instantiated item, and symbol
13312           -- table fields.
13313
13314          new_decls(id) := new_n ;
13315          nature(new_n) := nature(old_n) ;
13316          scope_of(new_n) := new_pn ;
13317
13318          if scope_of(old_n) ≠ old_n and declared(old_n) ≠ om and
13319             not can_overload(old_n)
13320           and not is_anonymous_task(old_n)
13321           then
13322              -- Nested record, package,  or task.
13323              -- Put on workpile with appropriate prefix for new names.
13324              workpile with:= [old_n, new_n, pref + id + '.'] ;
13325           end if ;
13326        end if;
13327
13328
13329    end forall ;
13330
13331    -- Assign new declarations to package, record or task entity.
13332
13333    declared(new_pn) := new_decls ;
13334    nat := nature(g_n) ;
13335
13336    if match(nat, 'package') ≠ om or
13337       match(nat, 'generic_package') ≠ om then
13338       packs with:= g_n ;
13339
13340       -- construct visible map for it, so that the proper instantiated
13341         -- entities within new package become accessible.
13342         visible(new_pn) := { [id, rename_map(old_n) ? old_n] :
13343                                        [id, old_n ] ∈  visible(g_n) } ;
13344    end if ;
13345
13346end while workpile ;
13347prefix := o_pref ;                          -- Restore naming environment
13348
13349-- The rename_map contains the generic items and the names of their
13350-- instantiations. We must now complete the symbol table entries for
13351-- the later, to insure that  type information is correct. Entities
13352-- that have been renamed (generic types, inout parameters, or actual
13353-- renaming within the generic object) have no symbol table entry in
13354-- it, and are skipped.
```

```
13355
13356(forall [old_n, new_n] ∈ rename_map | old_n  renamed_params)
13357
13358    type_of(new_n)   := rename_map(type_of(old_n)) ? type_of(old_n) ;
13359    signature(new_n) := replace(signature(old_n), rename_map) ;
13360    overloads(new_n) := replace(overloads(old_n), rename_map) ;
13361    alias(new_n)     := rename_map(alias(old_n))   ? alias(old_n)   ;
13362end forall ;
13363
13364-- Now instantiate the private declarations of packages. This is needed
13365-- in fact only for types with discriminants, but is done here for the
13366-- whole of the private part.
13367
13368(forall g_p ∈ packs)
13369
13370    new_p := rename_map(g_p) ;
13371    private_decls(new_p) := replace(private_decls(g_p), rename_map) ;
13372
13373    -- Nested packages (which are not generic) are now visible: their
13374    -- local objects are nameable using qualified names.
13375    if nature(g_p)  {'generic_package', 'generic_package_spec'}
13376    then
13377        vis_mods with:= new_p ;
13378    end if;
13379    -- The top level package is added to vis_mods in end_specs, called
13380    -- at the end of this instantiation.
13381
13382end forall ;
13383
13384-- Finally, apply same replacement rule to body of package. The body has
13385-- the same format as that produced for a non-generic package.
13386
13387if body = om then
13388    -- emit an instance stub for the binder. The package spec may
13389    -- actually not require a body, but the binder will find out.
13390    new_body := [['package_instance', gen_name,
13391          instantiation + p_specs, rename_map, must_constrain,
13392                                                spans(current_node)]] ;
13393else
13394    [decls, stmts, except] := body ;
13395    new_decls := instantiation + p_specs + decls ;
13396--
13397    new_body :=
13398      replace( [['package_body', package, [new_decls ,stmts, except]]],
13399                                                        rename_map) ;
13400end if;
13401
13402declared(package) := visible(package);
13403
```

```
13404
13405end_specs(pack_head, visible(package));
13406
13407-- The object just created is a package, although it appears syntactically
13408-- as a package spec. Its nature must be set correctly, to avoid spurious
13409-- error messages on exit from an enclosing package.
13410
13411nature(package) := 'package' ;
13412return new_body ;
13413
13414
13415
13416end proc package_instance ;
13417
```

instantiate_generics

```
13419proc instantiate_generics (generics, instance) ;
13420
13421-- Produce the list of renamings which transforms generic parameters
13422-- into actual ones.
13423-- Generic types play a special role in this renaming. We collect the
13424-- Instantiations of generic types into the map -type_map-and use it
13425-- in a substitution procedure to obtain the signature of generic
13426-- subprogram arguments.
13427-- Generic subprograms are also renamed by the actual subprograms, and
13428-- the mapping from one to the other is also added to the same renaming
13429-- map.
13430
13431const error_instance = [ [ ],{},{} ] ;          --
13432
13433
13434[gen_list, gen_code] := generics ;
13435instantiation_code := gen_code ;          -- Anonymous types, etc.
13436type_map := {} ;
13437must_constrain := {} ;                    --
13438
13439if # instance > # gen_list then
13440    errmsg('Too many actuals in generic instantiation','','12.3') ;
13441    return error_instance;
13442end if ;
13443
13444-- Values may be supplied either positionally or by name.
13445
13446if exists [formal, actual] = instance(i) | formal ≠ [ ]
13447  then
13448
13449    first_named := i ;
13450    if exists k ∈ [i..#instance] | instance(k)(1) = [ ] then
```

```
13451          errmsg('Positional association after named one', '', '12.3') ;
13452          return error_instance ;
13453       end if ;
13454
13455    else first_named := #instance + 1 ;
13456    end if ;
13457
13458    -- Process positional parameters.
13459
13460    (forall i ∈ [1..first_named-1])
13461       [g_name, g_mode, g_type, g_init] := gen_list(i) ;
13462       [-, actual] := instance(i) ;
13463
13464       case g_mode of
13465
13466       ('in', 'inout'):
13467          -- 'in' generic parameters can be used as constants and generated
13468          -- anonymous types may depend on them, so that their instantiation
13469          -- must be elaborated first.
13470           instantiation_code :=
13471            [instantiate_object(gen_list(i), actual, type_map) ]
13472                                                  + instantiation_code;
13473
13474       ('type'):
13475
13476          actual_type := instantiate_type(actual, g_name, type_map) ;
13477          if actual_type = '' then return error_instance ;
13478          else
13479             type_map(g_name) := actual_type ;
13480             if type_of(g_name) ∈ priv_types and
13481                         can_constrain(actual_type) then                --
13482                must_constrain with:= g_name ;
13483             end if ;
13484             if is_scalar_type(g_name) then                --
13485                type_map(type_of(g_name)) := base_type(actual_type) ;
13486             end if ;
13487          end if;
13488
13489       ('procedure', 'function'):
13490          -- Actual may be given by an operator symbol, which appear as
13491           -- string literal.
13492
13493        if is_tuple(actual) and actual(1) = 'string_literal' then
13494             actual := desig_to_name('"' + actual(2) + '"') ;
13495        end if ;
13496
13497          generic_subprog_instance(find_old(actual), g_name, g_type,
13498                                                        type_map);
13499       end case ;
```

```
13500
13501end forall ;
13502
13503-- now process named parameters.
13504
13505named_instance := instance(first_named..) ;
13506
13507(forall i ∈ [first_named..#gen_list])
13508
13509   [g_name, g_mode, g_type, g_init] := gen_list(i) ;
13510
13511   if exists [id, i_expr] = named_instance(k) |
13512                                        id = original_name(g_name)
13513   then
13514   TO_XREF(g_name) ;
13515   -- Remove this named association to catch duplicate associations.
13516   named_instance := named_instance(1..k-1) + named_instance(k+1..) ;
13517   case g_mode of
13518
13519   ('in', 'inout'):
13520
13521         instantiation_code :=
13522            [instantiate_object(gen_list(i) ,i_expr, type_map)]
13523                                              + instantiation_code ;
13524
13525
13526   ('type'):
13527
13528       actual_type := instantiate_type(i_expr, g_name, type_map) ;
13529      if actual_type = '' then return error_instance ;
13530       else
13531          type_map(g_name) := actual_type ;
13532          if type_of(g_name) ∈ priv_types and
13533                        can_constrain(actual_type) then                --
13534             must_constrain with:= g_name ;
13535          end if ;
13536          if is_scalar_type(g_name) then                --
13537             type_map(type_of(g_name)) := base_type(actual_type) ;
13538          end if ;
13539       end if ;
13540
13541   ('procedure','function'):
13542
13543     if #(overloads(declared(scope_of(g_name))(id)) ? {}) > 1 then
13544          errmsg('named associations not allowed for overloaded names',
13545                                        '', '12.3(3)') ;
13546     end if ;
13547
13548     if is_tuple(i_expr) and i_expr(1) = 'string_literal' then
```

```
13549           i_expr := desig_to_name('"' + i_expr(2) + '"') ;
13550       end if ;
13551
13552         generic_subprog_instance(find_old(i_expr), g_name, g_type,
13553                                                    type_map) ;
13554
13555       end case ;
13556
13557     elseif g_mode ≠ 'type' and g_init ≠ [ ] then
13558         -- A default was supplied for generic parameter. This can be the
13559         -- case for 'in' parameters and for subprograms.
13560       case g_mode of
13561
13562       ('in'):
13563
13564           -- The default expression may depend on generic types, e.g.
13565           -- by using an attribute of the type. replace generic type
13566           -- by the actual before elaborating.
13567           i_expr := replace(g_init, type_map) ;
13568           instantiation_code with:=
13569               instantiate_object(gen_list(i), i_expr, type_map) ;
13570
13571       ('procedure', 'function'):
13572
13573           generic_subprog_instance(g_init, g_name, g_type, type_map);
13574       end case ;
13575     else
13576         errmsg('Missing instantiation for generic parameter ',
13577                           original_name(g_name),'12.3') ;
13578         return error_instance ;
13579     end if;
13580
13581
13582end forall ;
13583
13584if named_instance ≠ [ ] then
13585     -- Not all named associations were processed.
13586     errmsg('duplicate or erroneous named associations in instantiation',
13587                           '','12.3') ;
13588end if ;
13589
13590-- The fragments in code_gen contain definitions of anonymous types,
13591-- etc. which may still refer to generic types. Replace them with
13592-- the actuals also.
13593
13594instantiation := replace(instantiation_code, type_map) ;
13595
13596
13597
```

```
13598--
13599return [instantiation, type_map, must_constrain] ;
13600
13601end proc instantiate_generics ;
13602
13603
```

instantiate_object

```
13605proc instantiate_object(gen_info, actual_expr, rw type_map) ;
13606
13607-- Unpack information about generic parameter.
13608[g_name, g_mode, g_type, -] := gen_info;
13609
13610-- The  type of a generic  object may be a generic type, a subtype of a
13611-- generic type, or a normal (non-generic) type. In the first two cases
13612-- we  must  determine or compute  the equivalent  instantiated type to
13613-- complete type checking.
13614
13615if nature(g_type) ∈ {'subtype','access'} and   -- May be subtype
13616    type_map(type_of(g_type)) ≠ om and      -- of a generic type.
13617    type_map(g_type) = om
13618then
13619    -- Create new subtype, after substitution.
13620    actual_type :=
13621            promote_subtype(replace(SYMBTAB(g_type), type_map)) ;
13622else
13623    actual_type := type_map(g_type)          -- If generic.
13624                                ? g_type;    -- Otherwise.
13625end if ;
13626
13627-- For each instantiation we must create locations for the generic
13628-- parameters, and replace in the body of the object the generic ones
13629-- with the actual ones.
13630
13631actual_name := prefix + original_name(g_name) + str newat ;
13632type_map(g_name) := actual_name ;
13633
13634if g_mode = 'in' then
13635    actual := check_type(actual_type, actual_expr) ;
13636    if is_deferred_constant(actual) then
13637      errmsg('Instantiation of a generic in parameter',
13638            ' cannot be a deferred constant','7.4.3');
13639      return [ ];
13640    end if;
13641    SYMBTAB(actual_name) := ['constant', actual_type, actual] ;
13642    return  ['constant', [actual_name], actual_type, actual];
13643else                                          -- in out parameter.
13644    if not is_tuple(actual_expr) or actual_expr(1) ≠ 'name' then
```

```
13645       errmsg('Instantiation of generic in out parameter ',
13646                                ' must be a variable','12.1.1, 12.3.1') ;
13647       return [ ] ;
13648    else
13649       actual_expr := find_old(actual_expr) ;
13650    end if ;
13651
13652    if is_tuple(actual_expr) and actual_expr(1) = 'convert'
13653    then
13654       errmsg('Instantiation of generic in  out parameter ',
13655                              'cannot be a conversion','12.3.1') ;
13656       return [ ] ;
13657    end if ;
13658    out_context := false ;                                          --
13659    actual := check_type(actual_type, actual_expr) ;
13660    -- A constraint may have been imposed on the variable, in the
13661    -- process of type checking. This is an artifact of type checking, as
13662    -- the only constraint on the actual come from its own type.
13663
13664    if is_tuple(actual) and actual(1) ∈ qualifiers then
13665       actual := actual(3) ;
13666    end if ;
13667
13668    if not is_variable(actual) then
13669       errmsg('Instantiation of generic in out parameter ',
13670                                ' must be a variable','12.1.1, 12.3.1') ;
13671       return [ ] ;
13672 elseif is_discriminant_dependent ( actual ) then
13673       errmsg('existence of generic in out parameter '
13674             + str original_name ( g_name ) ,
13675             ' depends on a discriminant','12.3.1') ;
13676       return [ ];
13677    else
13678       return ['renames', actual_name, actual] ;
13679    end if ;
13680end if;
13681
13682end proc instantiate_object ;
13683
13684
```

is_discriminant_dependent

```
13686proc is_discriminant_dependent ( expn ) ;
13687
13688--  Function :
13689--       this (non-recursive) procedure accepts as parameter an
13690--       expression that has been parsed as a valid 'name', and
13691--       return true if the existence of the object designated
```

```
13692--      may depend on a discriminant. See LRM 8.5, 3.7.1, 12.3.1.
13693--   Usage :
13694--      for generic in out parameter
13695--      for renaming
13696
13697
13698--   the expression is first 'flattened' :
13699
13700   expn := linear ( expn ) ;
13701
13702   first fromb expn ;
13703   is_first_element := true ;
13704   current_type := type_of ( first ) ;
13705   discr := [ ] ;
13706
13707--  the guess along that loop is that it is not dependent :
13708
13709   ( while (expn?[ ]) ≠ [ ] )
13710     case op_name fromb expn of
13711
13712--  Record case : check that component is in fixed part
13713--                keep discriminants in case of array component
13714      ('.'):
13715        base_type_rec := base_type ( current_type ) ;
13716        field_name fromb expn ;
13717--
13718        if (nature ( current_type ) = 'subtype')
13719          or
13720--  if it is a formal parameter of some unconstrained type, the actual
13721--  parameter must have been constrained...
13722             (     is_first_element
13723               and is_formal ( first )
13724               and is_unconstrained ( current_type ))
13725        then
13726          discr := discriminant_list ( base_type_rec );
13727        else
13728          if not exists
13729            [ -,nam_list,- ] ∈ invariant_part ( base_type_rec ) ,
13730            name ∈ nam_list | name = field_name
13731          then return true;
13732          end if;
13733          discr := [ ] ;
13734        end if;
13735        current_type := type_of ( field_name ) ;
13736
13737-- Array or Slice case : if bound is dynamic, is must be constrained
13738      ('[ ]','[..]'):
13739        bounds := [ ] ;
13740        (for i ∈ index_types(current_type))
```

```
13741            [ -, low, high] := signature (i) ;
13742              bounds +:= [low, high] ;
13743          end for ;
13744          if exists bound ∈ bounds | is_tuple(bound)
13745            and ( bound(1) =='discr_ref' ) and ( bound(2)   discr )
13746          then return true ;
13747          end if;
13748          if op_name = '[ ]' then
13749            current_type := component_type ( current_type ) ;
13750          end if;
13751-- Access case : cannot depend on a discriminant !
13752-- Function call : idem
13753       ('@','call'):
13754          return false ;
13755
13756-- Possible gap here
13757       else
13758          return false ;
13759       end case;
13760       is_first_element := false ;
13761
13762    end while;
13763
13764    return false ; -- the initial guess
13765
13766end proc is_discriminant_dependent ;
13767
13768
13769proc linear ( expn );
13770
13771--   Recursive function used by 'is_discriminant_dependent' to
13772--   flatten its argument. The grammar of interest for expn is :
13773--       expn ::= identifier
13774--              | '.' rec_expr field_name
13775--              | '[ ]' arr_expr index
13776--              | '[..]' arr_expr slice
13777--              | '@' expr
13778--              | 'call' identifier
13779
13780    if is_identifier ( expn ) then
13781       return [ expn ] ;
13782    else
13783       [ op_name, exp1, exp2 ] := expn ;
13784       case op_name of
13785          ('.'):
13786             return linear(exp1)+[op_name]+linear(exp2);
13787          ('[ ]','[..]','@','call'):
13788             return linear(exp1)+[op_name];
13789       else
```

```
13790         return [ ];
13791       end case;
13792    end if;
13793
13794end proc linear;
```

instantiate_type

```
13796--
13797proc instantiate_type(type_expr, g_name, rw type_map) ;
13798
13799-- Validate the instantiation of a generic type. The actual is a type
13800-- name or a subtype expression. The grammar allows a general expression
13801-- at this point, and we must insure that it is only a name, or a name
13802-- followed by a numeric constraint.
13803-- We must then check that the actual belongs to the same class as the
13804-- formal, and that in the case of arrays, constrained generics are ins-
13805-- tantiated with a constrained type.
13806
13807if is_tuple(type_expr) then
13808
13809    [tag, t] := type_expr ;
13810    if tag = 'name' and is_identifier(t)  then      -- Type identifier.
13811         actual_type := find_old(t) ;
13812
13813    elseif tag = 'name' then                        -- Index constraint
13814       actual_type := promote_subtype(make_subtype([t,[ ]])) ;
13815
13816    elseif tag = 'subtype' then                     -- numeric constraint
13817         actual_type := promote_subtype(make_subtype(t)) ;
13818    else
13819       errmsg('invalid expression for instantiation of ',
13820                    original_name(g_name),'12.3, 12.3.2-12.3.5') ;
13821       return '' ;
13822    end if ;
13823else errmsg('invalid expression for instantiation of ',
13824                         original_name(g_name),'12.3') ;
13825    return '' ;
13826end if ;
13827
13828if not is_type(actual_type) then
13829    errmsg('Expect type to instantiate ', original_name(g_name),
13830                  '12.3.2-12.3.5') ;
13831    return '' ;
13832else return valid_type_instance(g_name, actual_type, type_map) ;
13833end if ;
13834
13835
13836end proc instantiate_type ;
```

```
13837
13838--
13839proc valid_type_instance(g_name, actual_type, rw type_map) ;
13840
13841
13842if is_scalar_type(g_name) then
13843    return valid_scalar_instance(g_name, actual_type, type_map) ;
13844
13845elseif is_access(g_name) then
13846    return valid_access_instance(g_name, actual_type, type_map) ;
13847
13848elseif is_array(g_name) then
13849    return  valid_array_instance(g_name, actual_type, type_map) ;
13850
13851else
13852    return   valid_priv_instance(g_name, actual_type, type_map) ;
13853end if ;
13854
13855end proc valid_type_instance ;
13856
13857
```

valid_scalar_instance

```
13859proc valid_scalar_instance(g_name, actual_type, type_map) ;
13860
13861-- Complete the validation of the instantiation of a generic scalar type.
13862-- This procedure is also used to emit constraint checks on access types
13863-- and array types.
13864
13865--
13866g_type := root_type(g_name) ;                    -- INTEGER,FLOAT,-FIXED, etc.
13867
13868if g_type = root_type(actual_type) and is_generic_type(g_name)
13869then
13870    return actual_type ;
13871
13872elseif base_type(g_type) = base_type(actual_type)
13873then
13874    -- Checking instantiation of the designated type of an access type
13875    -- or  index type of an  array type. Verify that constraints match.
13876
13877    check_actual_constraint(g_name, actual_type) ;
13878    return actual_type ;
13879
13880elseif (g_type = 'discrete_type' and is_discrete_type(actual_type))
13881        then return actual_type ;
13882
13883else
```

```
13884    errmsg('Invalid type for instantiation of ',
13885                    original_name(g_name),'12.3.2 - 12.3.5') ;
13886    return '' ;
13887end if ;
13888
13889end proc valid_scalar_instance ;
13890
```

check_actual_conatraint

```
13892proc check_actual_constraint(g_type, a_type) ;
13893
13894-- Verify that the constraint on the designated type of an access type,
13895-- or an index type of an array type, match the constraints on the cor-
13896-- responding formal generic type. The types are known to be compatible.
13897
13898if is_scalar_type(g_type) then
13899    if not is_tuple(g_sig := signature(g_type)) then return ; end if ;
13900
13901    if not is_tuple(a_sig := signature(a_type)) then
13902    -- Bounds cannot be correct.
13903            instantiation_code with:= constraint_error() ;
13904    else
13905        [-, gen_lo, gen_hi] := g_sig ;
13906        [-, act_lo, act_hi] := a_sig ;
13907
13908          if gen_lo ≠ act_lo or gen_hi ≠ act_hi then
13909             -- Need run-time check.
13910           instantiation_code with:=
13911              ['if', [[['or',
13912                                  ['≠', gen_lo, act_lo],
13913                                    ['≠', gen_hi, act_hi]   ],
13914                              [['raise', 'CONSTRAINT_ERROR']] ]]  ] ;
13915        end if ;
13916
13917    end if ;
13918
13919elseif is_record(g_type) and nature(g_type) = 'subtype' then
13920    -- Check that discriminants match.
13921    if nature(a_type) ≠ 'subtype' then
13922        -- Mismatch was already signalled.
13923          return ;
13924    end if ;
13925
13926    [-, g_discr_map] := signature(g_type)  ;
13927    [-, a_discr_map] := signature(a_type) ;
13928    -- Compare the values of each discriminant.
13929
13930    (forall gv = g_discr_map(d))
```

```
13931          if (av := a_discr_map(d)) ≠ gv then
13932            instantiation_code with:=
13933           ['if',
13934              [[ ['≠', gv, av], [['raise', 'CONSTRAINT_ERROR']] ]] ] ;
13935          end if ;
13936      end forall ;
13937
13938elseif is_array(g_type) then
13939      g_info := get_array_info(g_type) ;
13940      a_info := get_array_info(a_type) ;
13941
13942      (forall i ∈ [1..#g_info])
13943         check_actual_constraint(g_info(i), a_info(i)) ;
13944      end forall ;
13945
13946elseif is_access(g_type) then
13947       check_actual_constraint(designated_type(g_type),
13948                               designated_type(a_type)) ;
13949end if ;
13950
13951end proc check_actual_constraint ;
13952
```

valid_priv_instance

```
13954--
13955proc valid_priv_instance(g_name, actual_type, rw type_map) ;
13956
13957g_type := type_of(g_name) ;
13958actual_base := base_type(actual_type) ;
13959
13960if type_of(actual_base) = 'incomplete' then
13961      errmsg('Invalid use of incomplete type in instantiation of ',
13962                                          original_name(g_name),'12.3');
13963      return '' ;
13964elseif private_ancestor(actual_base) ≠ om then
13965      errmsg('Invalid use of private type in instantiation of ',
13966                                          original_name(g_name),'12.3') ;
13967      return '' ;
13968elseif g_type = 'private' and is_limited_type(actual_type)
13969      then errmsg('Expect non-limited type to instantiate ',
13970                                        original_name(g_name),'12.3.2');
13971      return '' ;
13972
13973elseif has_discriminants(g_name)
13974and (not is_record(actual_base) or not has_discriminants(actual_base)
13975      or not is_valid_disc_instance(g_name, actual_base, type_map))
13976
13977      then  errmsg('discriminant mismatch in instantiation of ',
```

```
13978                                   original_name(g_name), '12.3.2');
13979    return '' ;
13980
13981elseif has_discriminants(g_name) and nature(actual_type) = 'subtype'
13982    then errmsg('Instantiation of ' + original_name(g_name),
13983                                    ' must be unconstrained', '12.3.2') ;
13984    return '' ;
13985
13986elseif '$constrain' ∈ misc_type_attributes(g_name) and
13987    can_constrain(actual_type) then                          --
13988    errmsg('Usage of private type ' + original_name(g_name),
13989            ' requires instantiation with constrained type','12.3.2') ;
13990    return '' ;
13991
13992else return actual_type ;
13993end if ;
13994
13995end proc valid_priv_instance ;
13996
```

valid_access_instance

```
13998proc valid_access_instance(g_name, actual_type, type_map) ;
13999
14000g_type := designated_type(g_name) ;
14001
14002if is_access(actual_type)  then
14003    -- the accessed actual type must be the proper instantiation
14004    -- of the accessed generic.
14005
14006    designated_formal := type_map(g_type) ? g_type ;
14007    designated_actual := designated_type(actual_type) ;
14008
14009
14010    if base_type(designated_formal) ≠ base_type(designated_actual)
14011    then
14012        errmsg('expect access to ' + original_name(designated_formal),
14013                ' to instantiate ' + original_name(g_name), '12.3.3') ;
14014        return '' ;
14015    end if ;
14016    if is_access(designated_formal) then
14017        designated_formal := designated_type(designated_formal) ;
14018        designated_actual := designated_type(designated_actual) ;
14019    end if ;
14020    if (can_constrain(designated_formal) ≠
14021        can_constrain(designated_actual)) then
14022        errmsg('formal and actual designated types must be both ',
14023                ' constrained or unconstrained', '12.3.3') ;
14024        return '' ;
```

```
14025   end if ;
14026--
14027   check_actual_constraint(designated_formal, designated_actual) ;
14028
14029   return actual_type ;
14030
14031else
14032   errmsg('Expect access type to instantiate ',
14033                              original_name(g_name),'12.3.5') ;
14034   return '' ;
14035end if ;
14036
14037end proc valid_access_instance ;
14038
```

valid_array_instance

```
14040proc valid_array_instance(g_name, actual_type, type_map) ;
14041
14042g_type := type_of(g_name) ;
14043
14044if not is_array(actual_type) then
14045        errmsg('Expect array type to instantiate ',
14046                              original_name(g_name),'12.3.4') ;
14047   return '' ;
14048
14049elseif can_constrain(actual_type) and
14050     not  can_constrain(g_name) then
14051   errmsg('Expect constrained array type to instantiate ',
14052                              original_name(g_name),'12.3.4') ;
14053   return '' ;
14054
14055elseif not can_constrain(actual_type) and
14056             can_constrain(g_name) then
14057   errmsg('Expect unconstrained array type to instantiate ',
14058                        original_name(g_name),'12.3.4') ;
14059
14060elseif no_dimensions(actual_type) ≠ no_dimensions(g_type)  then
14061   errmsg('Dimensions of actual type do not match those of ',
14062                                    original_name(g_name),'12.3.4') ;
14063   return '' ;
14064
14065else
14066   -- Collect index types and component type.
14067   g_info := get_array_info(g_type) ;
14068   a_info := get_array_info(actual_type) ;
14069   new_info := replace(g_info, type_map) ;
14070   g_comp := top(new_info) ;
14071   a_comp := top(a_info) ;                                    --
```

```
14072
14073    if exists t = new_info(i) |
14074                          not compatible_types(t, a_info(i)) then --
14075        errmsg('index or component type mismatch in instantiation',
14076                '  ' of array type ' + original_name(g_name),'12.3.4') ;
14077        return '' ;
14078--
14079    elseif  (if is_access(g_comp) then
14080                 can_constrain(designated_type(g_comp)) ≠
14081                   can_constrain(designated_type(a_comp))
14082             else
14083               can_constrain(g_comp) ≠
14084               can_constrain(a_comp)
14085             end)
14086        then
14087        errmsg('formal and actual array component type must be ',
14088                  'both constrained or unconstrained', '12.3.4');
14089        return '';
14090---
14091    else
14092        (for i ∈ [1..#new_info])
14093            check_actual_constraint(new_info(i), a_info(i)) ;
14094        end for ;
14095        return actual_type ;
14096    end if ;
14097end if ;
14098
14099
14100end proc valid_array_instance ;
14101
14102
```

is_valid_disc_instance

```
14104--
14105proc is_valid_disc_instance(g_name, a_name, rw type_map) ;
14106
14107-- checks that the formal and actual discriminant lists match in type
14108-- and position.
14109
14110g_list := discriminant_list(g_name) ;
14111a_list := discriminant_list(a_name) ;
14112if #g_list ≠ #a_list then return false ;
14113else
14114    (for gd = g_list(i))
14115        ad := a_list(i) ;
14116        t := type_of(gd) ;                    -- Type of discriminant
14117          gt := type_map(t) ? t ;                    -- may be formal generic.
14118          at := type_of(ad) ;                 -- Base type of actual
```

```
14119         if base_type(gt) ≠ base_type(at) then  -- must match.
14120            return false ;
14121         else
14122            check_actual_constraint(gt, at) ;  -- and constraints also.
14123            -- The discriminant names of the formal may have been used
14124            -- in a selector in the generic body.They must be mapped into
14125            -- the actual discriminants.
14126            type_map(gd) := ad ;
14127      end if ;
14128   end for ;
14129end if ;
14130
14131return true ;
14132
14133end proc is_valid_disc_instance;
14134
```

get_array_info

```
14136proc get_array_info(a_type) ;
14137
14138-- Make sequence of index and component type marks, for comparing a
14139-- generic array type with its instantiation.
14140
14141return index_types(a_type) with component_type(a_type) ;
14142
14143end proc get_array_info ;
14144
```

generic_subprogram_instance

```
14146proc generic_subprog_instance(instance, g_name, g_info, rw type_map);
14147
14148
14149-- Determine the operator, procedure, or attribute which is used to
14150-- instantiate a given generic subprogram parameter .
14151
14152
14153-- g_info is the signature and the return type of the generic sub-
14154-- program. To validate the new instance, we must first replace
14155-- generic types by actual types, to find the instantiated signature
14156-- of the subprogram .
14157
14158[new_signature, new_type] := replace(g_info, type_map) ;
14159
14160
14161-- If the box was used to specify a default subprogram, we must now
14162-- retrieve the visible instances of that identifier.
14163
```

```
14164if is_tuple(instance) and instance(1) = 'box' then
14165    name := find_old(instance(2)) ;
14166else
14167    name := instance ;
14168end if;
14169
14170if is_overloaded(name) then
14171    -- It must be the name of an operator or user-defined procedure.
14172    -- We must find the correct one by matching with the desired
14173    -- signature . This is equivalent to the matching of a renamed
14174    -- subprogram.
14175
14176    proc_name := renamed(name, new_signature, new_type) ;
14177
14178    if proc_name = '' then
14179        -- Ambiguous or no match.
14180        errmsg('ambiguous or invalid match for generic subprogram ',
14181                                    original_name(arb name),'12.3.6') ;
14182      proc_name := [ ] ;
14183    end if ;
14184
14185elseif is_tuple(name) and name(1) = '''' then
14186
14187    -- Instantiation by an attribute. This is correct if the procedure
14188    -- has a single argument, and the proper type relation holds
14189    -- between it and the return type.
14190    [-,attr, typ] := name ;
14191    typ := replace(typ, type_map) ;
14192    proc_name := ['''', attr, typ] ;
14193
14194    -- Only attributes that are functions can be used as default actual.
14195    -- The type of their actual parameter is determined as follows:
14196
14197    if attr ∈ {'VAL', 'PRED', 'SUCC', 'VALUE'} then
14198        a_type := typ;
14199    elseif attr = 'POS' then
14200        a_type := 'universal_integer' ;
14201    elseif attr = 'IMAGE' then
14202        a_type := 'STRING' ;
14203    else
14204        a_type := 'none';                -- not a function attribute.
14205    end if ;
14206
14207    if nature(attr) ≠ 'attribute' or not is_type(typ)
14208        or # new_signature > 1 or
14209--
14210        not compatible_types(new_type , a_type) then                --
14211        errmsg('Attribute does not match generic subprog.spec.',
14212                                    '','12.3.6') ;
```

```
14213          proc_name := [ ] ;
14214      end if ;
14215
14216elseif new_type = 'none' and is_tuple(name) then
14217
14218      -- possible renaming by entry expression.
14219      proc_name := renamed_entry(name, new_signature) ;
14220
14221else errmsg('Invalid object for instantiation of generic subprog',
14222                        '','12.3.6') ;
14223      proc_name := [ ] ;
14224end if ;
14225
14226if is_identifier(proc_name) then
14227      type_map(g_name) := proc_name ;
14228elseif proc_name ≠ [ ] then
14229
14230      if proc_name(1) = '"' then
14231            instantiation_code with:=
14232                  renamed_attribute(g_name, new_signature, proc_name) ;
14233
14234       else                                    -- Instance is an entry.
14235            new_name := g_name + str newat ; -- Procedure whose body
14236            type_map(g_name) := new_name ;          -- is an entry call.
14237            instantiation_code with:=
14238                  entry_as_procedure(new_name, new_signature, proc_name)  ;
14239       end if ;
14240end if ;
14241
14242
14243end proc generic_subprog_instance ;
14244
```

replace

```
14246proc replace(expn, mapping) ;
14247
14248-- General subtitution procedure. Replace all identifiers in expn
14249-- by their images under -mapping- .
14250
14251return ( if expn = om then om
14252       elseif is_identifier(expn) then
14253            mapping(expn) ? expn
14254       elseif is_tuple(expn) then
14255            [replace(twig, mapping) : twig ∈ expn]
14256       elseif is_set(expn) then
14257            {replace(obj, mapping)  : obj  ∈ expn}
14258       else
14259            expn
```

```
14260       end ) ;
14261
14262end proc replace ;
14263
```

replace_prefix

```
14265proc replace_prefix(body, old_prefix, new_prefix) ;
14266end if ;
14267
14268-- This procedure creates new names for the identifiers in a generic
14269-- module, by replacing the generic package name, which appears as a
14270-- prefix of all local names, with name of instantiated package.
14271
14272if body = om then return om ;
14273elseif is_identifier(body) then return
14274    (if match(body,old_prefix) ≠ om then new_prefix + body
14275                                    else body end) ;
14276elseif is_tuple(body) then return
14277    [replace_prefix(branch, old_prefix,new_prefix) : branch ∈ body] ;
14278elseif is_set(body) then return
14279    {replace_prefix(branch, old_prefix,new_prefix) : branch ∈ body} ;
14280else return body;
14281end if ;
14282end proc replace_prefix;
```

13. Representation Clauses

```
14284
14285
14286-- Representation clauses are checked for semantic correctness and
14287-- AIS code is emitted for them. With the exception of enumeration
14288-- representation clauses the clauses have no effect on the internal
14289-- representations of the objects.
14290
```

check_repr_tyep

```
14292proc check_repr_type (type_name);
14293
14294-- Check whether the type to which a representation clause applies
14295-- occurs within the same declarative part as the clause itself and
14296-- precedes it. If the type is a private type it cannot be used before
14297-- its full type declaration. The unique name of the type is returned
14298-- or '?' in case of error.
14299
14300if (unq_name := DECLARED (SCOPE_NAME)(type_name)) = om
14301   then errmsg (type_name,' does not appear in same declarative part',
14302               '13.1');
14303        return '?';
14304elseif type_of(unq_name) ∈ {'private', 'limited private','incomplete'}
14305   then errmsg(type_name,' appears before its full type declaration',
14306              '7.4.1');
14307        return '?';
14308else return unq_name;
14309end if;
14310
14311end proc check_repr_type;
```

length_clause

```
14313proc length_clause (attr_type, attr_prefix, expn);
14314
14315-- This procedure processes a length clause.
14316
14317if (attr_prefix := check_repr_type(attr_prefix)) = '?' then
14318   return [ ];
14319end if;
14320length_result := [ ];
14321
14322if attr_type = 'SIZE' then
```

```
14323
14324   if is_type(attr_prefix) and nature(attr_prefix)≠'subtype'
14325         or is_first_named_subtype(attr_prefix) then
14326     size_value := check_type ('INTEGER', expn);
14327     if is_value(size_value) then
14328       length_result := [['size_repr', attr_prefix, size_value(2) ]];
14329     else
14330       errmsg('Expression in size spec is not static','','13.2');
14331     end if;
14332   else
14333     errmsg('Prefix of attribute is not type or ',
14334           'first named subtype','13.2');
14335   end if;
14336
14337elseif attr_type = 'STORAGE_SIZE' then
14338
14339   if is_task_type(attr_prefix)  or  is_anonymous_task(attr_prefix) then
14340     task_size := check_type ('INTEGER', expn);
14341     length_result := [['task_size_repr', attr_prefix, task_size ]];
14342   elseif is_access(attr_prefix) then
14343     access_size := check_type ('INTEGER', expn);
14344     length_result :=[['access_size_repr', attr_prefix, access_size ]];
14345   else
14346     errmsg('Prefix of attribute is not task type or',
14347           'access type','13.2');
14348   end if;
14349--
14350elseif attr_type = 'SMALL' then
14351    if root_type(attr_prefix) ≠ '$FIXED' then
14352        errmsg('expect fixed type for repr.spec. of SMALL','','13.2') ;
14353    end if ;
14354
14355    small := check_type('real_type', expn) ;
14356    if not is_value(small) then
14357        errmsg('length specification must be a static expression','',
14358                                                '13.2') ;
14359--
14360    end if ;
14361    signature(attr_prefix) with:= small ;
14362
14363end if;
14364
14365return length_result;
14366
14367end proc length_clause;
```

enum_rep_clause

```
14369proc enum_rep_clause (type_name, aggregate);
14370
14371-- This procedure checks the validity of the representation clause for
14372-- enumeration types.
14373
14374if (type_name := check_repr_type(type_name)) = '?' then return [ ]; end;
14375if nature(type_name) ≠ 'enum' then
14376   errmsg ('Identifier is not an enumeration type','','13.3');
14377   return [ ];
14378end if;
14379
14380-- The representation is given by an aggregate, whose index type is the
14381-- given enumeration type, and whose component type is integer. We
14382-- build such an array type for type checking, but emit no code for it.
14383
14384enum_aggr_type := str newat;
14385array_info := [[type_name], 'INTEGER'];
14386new_constrained_array(enum_aggr_type, array_info) ;
14387sequence_type frome top(NEWTYPES) ;                    -- discard
14388aggr_object := adasem (aggregate);
14389aggr_val := check_type (enum_aggr_type, aggr_object);
14390if is_value (aggr_val) then
14391  [-, seq, -] := aggr_val(2);
14392  (for i ∈ [1..#seq -1])
14393     if seq(i) ≥ seq(i+1) then
14394        errmsg ('Integer code is not distinct or violates ',
14395              'predefined ordering relation of type','13.3');
14396        return [ ];
14397     end if;
14398  end for;
14399  new_lit_map :=
14400   {[lit_name, seq(pos +1)] : [lit_name,pos] ∈ literal_map(type_name)};
14401  SYMBTAB(type_name) :=
14402         ['enum', type_name, ['range', ['ivalue', seq(1)],
14403                        ['ivalue', seq(#seq)]], new_lit_map] ;
14404
14405  return
14406    [['enum_size_repr', type_name, seq(1), seq(#seq), new_lit_map]];
14407
14408else
14409  errmsg ('Component of aggregate in enumeration representation clause',
14410         'is not static','13.3');
14411  return [ ];
14412end if;
14413
14414end proc enum_rep_clause;
```

rec_rep_clause

```
14416proc rec_rep_clause (type_name; align_clause, comp_clause_list);
14417
14418if (type_name := check_repr_type(type_name)) = '?' then return [ ]; end;
14419if not is_record (type_name) then
14420   errmsg ('Identifier is not a record type','','13.4');
14421   return [ ];
14422end if;
14423
14424repr_err := false;
14425if align_clause = [ ]
14426  then modulus_value := 0;
14427elseif is_value (modulus_value := check_type('INTEGER', align_clause))
14428  then modulus_value := modulus_value (2);
14429else
14430  errmsg('Alignment clause must contain a static expression','','13.4');
14431  repr_err := true;
14432end if;
14433
14434location_lists := {};
14435field_names := domain (declared_components (type_name));
14436duplic_list := {};
14437
14438(for [field, rel_addr, bit_range] ∈ comp_clause_list)
14439
14440  if field   field_names then
14441     errmsg ('Component '+ field,' does not appear in record type',
14442           'none');
14443     repr_err := true;
14444  elseif field ∈ duplic_list then
14445     errmsg ('Component '+ field,' already occurs in clause','none');
14446     repr_err := true;
14447  else
14448     duplic_list with := field;
14449  end if;
14450
14451  rel_addr := check_type ('INTEGER', rel_addr);
14452  if is_value (rel_addr) then
14453     rel_addr := rel_addr(2);
14454  else
14455     errmsg ('Expression for component ' + field,
14456           ' must be static','13.4');
14457     repr_err := true;
14458  end if;
14459
14460  if bit_range(1) = 'range' then
14461     first_bit := check_type ('INTEGER', bit_range(2));
```

```
14462     last_bit  := check_type ('INTEGER', bit_range(3));
14463     if is_value (first_bit) and is_value (last_bit) then
14464       location_lists with :=
14465             [field, rel_addr, [first_bit(2), last_bit(2)]];
14466     else
14467       errmsg('Range for component '+field,' must be static','13.4');
14468       repr_err := true;
14469     end if;
14470
14471  -- The constraint is given by a RANGE attribute.
14472  -- We get the bounds of the range to construct the new
14473  -- subtype.
14474
14475  elseif not is_range_attribute(bit_range) then
14476     errmsg('Invalid expression for range constraint','','3.3');
14477     repr_err := true;
14478  else
14479     bit_range := range_type(bit_range) ;
14480
14481     if is_identifier(bit_range) then
14482       -- get the bounds from the signature of the range type.
14483       [-, lo, hi] := signature(bit_range) ;
14484       typ := bit_range ;
14485     else                -- Unpack subtype expression.
14486       [-, typ, [-, lo, hi]] := bit_range ;
14487     end if ;
14488
14489     if not compatible_types('INTEGER', typ) then          --
14490       errmsg('RANGE attribute has wrong type for constraint',
14491                                          '','3.3') ;
14492       repr_err := true;
14493     else
14494         lo:= eval_static(['qual_range', type_mark, lo]) ;
14495         hi := eval_static(['qual_range', type_mark, hi]) ;
14496         location_lists with := [field, rel_addr, [lo, hi]] ;
14497     end if ;
14498  end if ;
14499
14500end for;
14501
14502return
14503  if repr_err then [ ]
14504  else [['record_repr', type_name, modulus_value, location_lists ]] end;
14505
14506end proc rec_rep_clause;
14507
14508
14509
14511--
```

Appendix A. Predefined language attributes.

check_range_attribute

```
14513proc check_range_attribute(expn) ;
14514
14515-- Syntactic check: the RANGE attribute, which is a name, cannot appear
14516-- as a primary except as the second argument of a membership operator.
14517
14518if is_tuple(expn) and expn(1) = 'name' and is_tuple(nam := expn(2))
14519and is_range_attribute(nam) then
14520    errmsg('Invalid use of attribute RANGE in expression', '', '4.4') ;
14521end if ;
14522
14523end proc check_range_attribute ;
14524--
```

Constant folding

const_val

```
14527proc CONST_VAL(obj);
14528
14529-- Return the constant value of the object if it has one; else return
14530-- obj unchanged.  A constant value has the format:
14531--                          ['ivalue', value]
14532-- The constant value of a user-defined constant is derived from its
14533-- SIGNATURE, when this is a static value. The constant value of a
14534-- literal is obtained from the literal map of its type. When perfor-
14535-- ming conformance checks, folding is desabled by the global flag
14536-- fold_context.
14537
14538if is_literal(obj) then
14539    return ['ivalue', literal_map(type_of(obj))(original_name(obj))] ;
14540end if;
14541sig := signature(obj);
14542if is_constant(obj) and is_scalar_type(type_of(obj)) and
14543                    is_value(sig) and fold_context then
14544    return sig ;
14545else
14546    return obj;
14547end if;
14548
14549end proc CONST_VAL;
```

eval_static

```
14551proc eval_static (expn);
14552--
14553-- This procedure is called to evaluate a static expression.
14554-- If expn is static, its internal value is returned, otherwise
14555-- expn is returned.  If the static evaluation shows that the
14556-- expression would raise an exception, a ['raise' exception]
14557-- value is returned.
14558---
14559-- Utility macros
14560--
14561
14562-- num and den access parts of a rational number
14563
14564    macro num(x); x(1) endm;
```

```
14565   macro den(x); x(2) endm;
14566
14567   macro b_not(x);
14568       (1 - (x))
14569   endm;
14570
14571   macro test(x);
14572       if x then 1 else 0 end
14573   endm;
14574
14575   macro bool_op(opn,op1,op2);
14576       case opn of
14577         ('and') :      ((op1) min (op2)),
14578         ('or')  :      ((op1) max (op2)),
14579         ('xor') :      test((op1) ≠ (op2))
14580       else
14581                        expr assert false; end
14582       end
14583   endm;
14584
14585
14586   macro is_simple_value(t);
14587       (is_integer(t) or is_real(t))
14588   endm;
14589
14590   macro check_const_val(x);
14591       if is_constant(x) and is_value(signature(x)) and fold_context
14592       then signature(x) else x end
14593   endm;
14594
14595
14596
14597--
14598-- If the expression is a scalar constant or a literal, return that value
14599--
14600   if expn = om or is_value(expn) then
14601       return expn;
14602   elseif is_identifier(expn) then
14603       return CONST_VAL(expn);
14604--
14605-- If expn is an expression, whose operands are ivalues, then
14606-- perform the operation.
14607--
14608-- Check whether the operator is unary:
14609--
14610   elseif is_tuple(expn) and #expn = 2 then
14611       [opn,op1] := expn;
14612       case opn of
14613
```

```
14614          ('int_literal'):
14615              return ['ivalue', ADAVAL ('universal_integer', op1)];
14616
14617          ('real_literal'):
14618              return ['ivalue', ADAVAL ('universal_real', op1)];
14619
14620          ('string_literal'):
14621              return ['ivalue', ['string', op1]];
14622
14623          ('CHARACTER'):
14624              return ['ivalue', abs op1];
14625
14626          ('+ui', '+ufl', '+ufx'):
14627              if is_value(op1) then
14628                  return op1;
14629              else
14630                  return expn;
14631              end if;
14632
14633          ('-ui', '-ufl', '-ufx'):
14634              if is_value(op1) then
14635                  op1 := op1(2);
14636              else
14637                  return expn;
14638              end if;
14639              return
14640                  if is_simple_value(op1) then
14641                      ['ivalue', - op1]
14642                  else
14643                      ['ivalue', rat_umin(op1)]
14644                  end;
14645
14646          ('not'):
14647              if is_value(op1) then
14648                  op1 := op1(2);
14649              else
14650                  return expn;
14651              end if;
14652              return
14653                  if is_simple_value(op1) then
14654                      ['ivalue', b_not(op1)]
14655                  else -- - /TBSL/
14656                      expn
14657                  end;
14658
14659          ('absi', 'absfl', 'absfx'):
14660              if is_value(op1) then
14661                  op1 := op1(2);
14662              else
```

```
14663                   return expn;
14664               end if;
14665               return
14666                   if is_simple_value (op1) then
14667                       ['ivalue', abs op1]
14668                   else
14669                       ['ivalue', rat_abs(op1)]
14670                   end;
14671
14672           ('row'):      -- - /TBSL/ Bounds may be incorrect!
14673               op1 := check_const_val(op1);
14674               if is_value(op1) then
14675                   return ['ivalue', ['array_ivalue', [op1(2)], 1, 1]];
14676               else
14677                   return expn;
14678               end if;
14679
14680           ('()'):
14681              -- If the parenthesised expression is evaluable, return
14682               -- its value. Otehrwise leave it parenthesised.
14683               e1 := eval_static(op1) ;
14684               if is_value(e1) then return e1 ;
14685               else return expn ;
14686               end if ;
14687           else
14688               return expn;
14689       end case;
14690
14691--
14692-- Here we handle special binary operators
14693--
14694   elseif is_tuple(expn) and #expn = 3 then
14695       [opn,op1,op2] := expn;
14696
14697       case opn of
14698
14699           ('qual_range'):
14700--
14701             rv := eval_qual_range(op1,op2,expn);
14702             if rv = 'constraint_error' then
14703               return constraint_error();
14704             else
14705               return rv;
14706               end if ;
14707--
14708
14709           ('qual_length'):
14710               if is_value(op2) then
14711                   op2 := op2(2);
```

```
14712               else
14713                   return expn;
14714               end if;
14715               if can_constrain(op1) then
14716                   return ['ivalue', op2] ;
14717               end if ;
14718               [-, lo, hi] := SIGNATURE(index_type(op1));
14719               if is_value(lo) and is_value(hi) then
14720                   lo := lo(2);
14721                   hi := hi(2);
14722               else
14723                   return expn;
14724               end if;
14725
14726               [-, -, lov, hiv] := op2;
14727               if (hi < lo and hiv < lov) or hi - lo = hiv - lov then
14728                   return ['ivalue', op2];
14729               else
14730                   return constraint_error();
14731               end if;
14732
14733           ('in', 'notin') :
14734               return expn ;
14735
14736
14737           ('record_aggregate'):
14738               if nature(op1) ≠ 'record' or type_of(op1)(2) ≠ [ ] then
14739                   return expn;        -- Could be static but let it go.
14740               end if;
14741
14742               if exists [-, v] ∈ op2 | not is_value(v) then
14743                   return expn;        -- Not all components are static
14744               end if;
14745
14746               return ['ivalue', ['record_ivalue',
14747                       {[f, v(2)] : [fn, v] ∈ op2, f ∈ fn} ] ];
14748
14749
14750           ('.'):-- - /TBSL/ Case for discriminants needed
14751               op1 := check_const_val(op1);
14752               if is_value(op1) then
14753                   return ['ivalue', op1(2)(2)(op2)];
14754               else
14755                   return expn;
14756               end if;
14757
14758           ('[..]'):  -- - /TBSL/  (not clear this can ever be static)
14759               return expn;
14760
```

```
14761         else
14762            pass;
14763         end case;
14764         op1 := check_const_val(op1);
14765         if is_value(op1) then
14766            op1 := op1(2);
14767         else
14768            return expn;
14769         end if;
14770
14771-- short-circuit Boolean expressions are not static.
14772
14773         if opn ∈ ['andthen', 'orelse'] then return expn ; end if ;
14774
14775         op2 := check_const_val(op2);
14776         if is_value(op2) then
14777            op2 := op2(2);
14778         else
14779            return expn;
14780         end if;
14781
14782         case opn of
14783
14784            ('[ ]'):
14785               [-, agg, lb, ub] := op1;
14786               if op2 ≥ lb and op2 ≤ ub then
14787                  return ['ivalue', agg(op2 - lb + 1)];
14788               else
14789                  return constraint_error();
14790               end if;
14791
14792            ('+i', '+fl'):
14793               return check_overflow(op1 + op2);
14794
14795            ('+fx'):
14796               return ['ivalue', rat_add (op1, op2)];
14797
14798            ('-i', '-fl'):
14799               return check_overflow(op1 - op2);
14800
14801            ('-fx'):
14802               return ['ivalue', rat_sub (op1, op2)];
14803
14804            ('*i'):
14805               return
14806                 check_overflow(op1 * op2);
14807
14808            ('*fl'):
14809               if (abs op1 < ADA_SMALL_REAL)
```

```
14810           or (abs op2 < ADA_SMALL_REAL) then
14811             return 0.0;
14812           elseif log abs op1 + log abs op2 > ADA_MAX_REAL then
14813                return numeric_error();
14814             else
14815                return check_overflow(op1 * op2);
14816             end if;
14817
14818         ('*fx'):
14819             return ['ivalue', rat_mul (op1, op2)];
14820
14821         ('*fxi','*fli'):
14822             return
14823             ['ivalue', rat_fri ((num(op1) * op2), den(op1))];
14824
14825         ('*ifx'):
14826             return
14827             ['ivalue', rat_fri ((op1 * num(op2)), den(op2))];
14828
14829         ('/i'):
14830             if op2 = 0 then
14831                return numeric_error();
14832             end if;
14833             return ['ivalue', op1 div op2];
14834
14835         ('/fl'):
14836           if abs op2 < ADA_SMALL_REAL then
14837             return numeric_error();
14838             elseif abs op1 < ADA_SMALL_REAL then
14839             return 0.0;
14840           elseif log abs op1 - log abs op2 > log ADA_MAX_REAL then
14841                return numeric_error();
14842             else
14843                return check_overflow(op1 / op2);
14844             end if;
14845
14846         ('/fx'):
14847             if num(op2) = 0  then
14848                return numeric_error();
14849             end if;
14850             return ['ivalue', rat_div (op1, op2)];
14851
14852         ('/fxi','/fli'):
14853             if op2 = 0 then
14854                return numeric_error();
14855             end if;
14856             return
14857             ['ivalue', rat_fri (num(op1),  (den(op1) * op2))];
14858
```

```
14859          ('remi'):
14860              if op2 = 0 then
14861                 return numeric_error();
14862              end if;
14863              return ['ivalue', op1 - (op1 div op2)*op2];
14864
14865          ('modi'):
14866              if op2 = 0 then
14867                 return numeric_error();
14868              end if;
14869              rm := op1 mod op2 ;
14870              return ['ivalue', if rm = 0 or op2 > 0 then rm
14871                                else rm + op2 end ];          --
14872
14873          ('**i'):
14874              if op2 < 0 then
14875                 return constraint_error();
14876              end if;
14877              return
14878                check_overflow((op1) ** (op2));
14879
14880          ('**fl'):
14881              if abs op1 < ADA_SMALL_REAL or
14882                float abs op2 * log abs op1 > log ADA_MAX_REAL then
14883                 return numeric_error();
14884              else
14885                 return check_overflow(op1 ** op2);
14886              end if;
14887
14888          ('&'):                -- - /TBSL/ Bounds may not be correct!
14889              [-, agg1, lb1, ub1] := op1;
14890              [-, agg2, lb2, ub2] := op2;
14891              agg := agg1 + agg2;
14892              lb := lb1 min lb2;
14893              return
14894               ['ivalue', ['array_ivalue', agg, lb, lb + #agg - 1]];
14895
14896          ('and','or','xor'):
14897              if is_simple_value(op1) then
14898                 return ['ivalue', bool_op(opn,op1,op2)];
14899              else             -- - /TBSL/
14900                 return expn;
14901              end if;
14902
14903          ('='):
14904              if is_universal_real(op1) and
14905                                    is_universal_real(op2) then
14906                 return ['ivalue', test(rat_eql(op1, op2))];
14907              elseif is_simple_value(op1) then
```

```
14908                return ['ivalue', test(op1 = op2)];
14909             else return expn ;
14910             end if;
14911
14912          ('≠'):
14913             if is_universal_real(op1) and
14914                                        is_universal_real(op2) then
14915                return ['ivalue', test(rat_neq(op1, op2))];
14916             elseif is_simple_value(op1) then
14917                return ['ivalue', test(op1 ≠ op2)];
14918             else return expn ;
14919             end if;
14920
14921          ('<'):
14922             if is_simple_value(op1) then
14923                return ['ivalue', test(op1 < op2)];
14924             else              -- - /TBSL/ need array types
14925                return expn;
14926             end if;
14927
14928          ('≤'):
14929             if is_simple_value(op1) then
14930                return ['ivalue', test(op1 ≤ op2)];
14931             else              -- - /TBSL/ need array types
14932                return expn;
14933             end if;
14934
14935          ('>'):
14936             if is_simple_value(op1) then
14937                return ['ivalue', test(op1 > op2)];
14938             else              -- - /TBSL/ need array types
14939                return expn;
14940             end if;
14941
14942          ('≥'):
14943             if is_simple_value(op1) then
14944                return ['ivalue', test(op1 ≥ op2)];
14945             else              -- - /TBSL/ need array types
14946                return expn;
14947             end if;
14948
14949          else
14950             return expn;
14951       end case;
14952
14953
14954
14955
14956    elseif is_tuple(expn) and #expn = 4 then
```

```
14957
14958   case expn(1) of
14959
14960     ("'"):                          -- static evaluation of attributes.
14961       [-, attr, typ1, op1] := expn;
14962
14963       -- The array attributes apply to types as well as objects. The
14964      -- interpreter distinguishes among these two by means of the
14965       -- prefixes  T_ and O_. Same is done for CONSTRAINED.
14966
14967     if attr ∈ {'FIRST', 'LAST', 'RANGE', 'LENGTH', 'CONSTRAINED'}
14968     then
14969          if is_identifier(typ1) and is_type(typ1) then
14970               pref := 'T_' ;
14971               if can_constrain(typ1) and attr ≠ 'CONSTRAINED' then
14972                  errmsg('attribute ' + attr + ' is undefined on ',
14973        ' unconstrained type ' + original_name(typ1),'3.6.2') ;
14974               end if ;
14975          else
14976               pref := 'O_' ;
14977          end if ;
14978          expn(2) := pref + attr ;
14979       end if ;
14980
14981      if not is_identifier(typ1) then
14982          return expn;
14983      end if ;
14984
14985--
14986       -- Attributes that are functions take the type of their argument.
14987       if attr ∈ base_attributes then
14988          expn(3) := base_type(typ1) ;
14989       end if ;
14990       -- They are evaluable statically only if the subtype typ1 itself
14991       -- is static.
14992--
14993      if is_type(typ1) and is_static_subtype(typ1)
14994       or is_task_type(type_of(typ1))
14995       or attr = 'CONSTRAINED' then  pass;      -- try to evaluate
14996       else
14997        return expn;                           -- not static (RM 4.9 (8)
14998      end if;
14999--
15000       if is_generic_type(typ1) then
15001          return expn;
15002       end if;
15003
15004       if is_tuple (tsig := signature(typ1)) then
15005          [-, first, last, d] := tsig;
```

```
15006       end if;
15007
15008
15009       case attr of
15010
15011    -- Attributes of SCALAR types or ARRAY types:
15012
15013          ('FIRST'):
15014             if is_value(first) then
15015                return first;
15016             else
15017                return expn;
15018             end if;
15019
15020          ('LAST'):
15021             if is_value(last) then
15022                return last;
15023             else
15024                return expn;
15025             end if;
15026
15027          ('RANGE', 'LENGTH'):          -- Non-static.   ES139
15028             return expn;
15029
15030          -- Attributes of DISCRETE types:
15031
15032          ('IMAGE'):
15033             if is_value(op1) then
15034                op1 := op1(2);
15035             else
15036                return expn;
15037             end if;
15038             btyp1 := root_type(typ1);
15039             if (btyp1 = 'INTEGER') then
15040              image := str op1 ;
15041             if op1≥0 then
15042                image := ' '+image;
15043             end if;
15044             else              -- Enumeration type
15045                image :=
15046                  if exists [nam, v] ∈ literal_map(btyp1)| op1 = v
15047                     then nam else '' end ;
15048             end if;
15049             return ['ivalue', ['array_ivalue',
15050                   [abs c : c ∈ image], 1, #image]] ;
15051
15052          ('VALUE'):
15053             if is_value(op1) then
15054                [-,seq,l_val,u_val] := op1(2);
```

```
15055              else
15056                 return expn;
15057              end if;
15058            setl_string := +/[char(seq(i)): i ∈ [l_val..u_val]];
15059            leading_blanks := span(setl_string,' ');
15060            trailing_blanks := rspan(setl_string,' ');
15061               btyp1 := root_type(typ1);
15062               if (btyp1 = 'INTEGER') then
15063              v := ADAVAL('integer',setl_string);
15064              if v = 'OVERFLOW' then
15065                 return constraint_error();
15066              else
15067                 return ['ivalue',v];
15068                 end if;
15069
15070               else              -- Enumeration type
15071              if #setl_string>0 and setl_string(1) ≠ '"' then
15072                setl_string := UPPER_CASE_OF(setl_string);
15073              end if;
15074                 if (i := literal_map(btyp1)(setl_string)) ≠ om then
15075                return ['ivalue',i];
15076              else
15077                return constraint_error();
15078                 end if ;
15079               end if;
15080
15081
15082         ('POS'):
15083              if is_value(op1) then
15084                 return ['ivalue',op1(2)] ;          --
15085              else
15086                 return expn ;
15087              end if ;
15088
15089         ('VAL','PRED','SUCC'):
15090              if is_value(op1) then
15091                op1 := op1(2);
15092              else
15093                 return expn;
15094              end if;
15095              rv := if attr = 'VAL' then op1
15096                    elseif attr = 'PRED' then op1-1
15097                    else op1 + 1 end;
15098              [-,lo, hi] := signature(base_type(typ1)) ? [ ];
15099              if is_value(lo) and is_value(hi) then
15100                 if rv ≥ lo(2) and rv ≤ hi(2) then
15101                    return ['ivalue', rv];
15102                 else
15103                    return constraint_error();
```

```
15104                end if;
15105             else
15106                return expn;
15107             end if;
15108
15109
15110          ('WIDTH'):
15111             if root_type(typ1) = 'INTEGER' then
15112                if is_value(first) and is_value(last) then
15113                   max_val := abs first(2) max abs last(2) ;
15114                   return ['ivalue', # str max_val + 1] ;
15115                else
15116                   return expn ;
15117                end if ;
15118             else
15119             d := literal_map (typ1);
15120                -- Must find longest name in enumeration type.
15121                return ['ivalue', [max/[# lit: [lit, -] ∈ d]]] ;
15122             end if ;
15123
15124
15125          -- Miscellaneous attributes.
15126          -- none of the following are static.
15127
15128          ('ADDRESS', 'FIRST_BIT', 'LAST_BIT', 'POSITION',
15129                                        'STORAGE_SIZE'):
15130             return expn ;
15131
15132          ('SIZE'):
15133             return ['ivalue', 0] ;
15134
15135          ('BASE'):
15136             return base_type(typ1);
15137
15138          ('CONSTRAINED'):
15139             -- parameters, false on non-generic types with discrs.
15140             if is_identifier(typ1)
15141               and nature(typ1) ∈ {'constant','in'} then
15142                return ['ivalue', 1] ;
15143
15144             elseif not is_generic_type(typ1) and
15145             is_record(typ1) and has_discriminants(typ1) then
15146                return ['ivalue', 0] ;
15147             else
15148                return expn;
15149             end if ;
15150          ('TERMINATED','CALLABLE') :          --
15151             return expn ;
15152
```

```
15153           else
15154
15155           -- Attributes of FIXED and FLOATing point types:
15156
15157           return eval_real_type_attribute(expn);  --
15158
15159       end case;
15160
15161
15162   ('array_aggregate'):
15163
15164       [-, indices, pos_list, nam_list] := expn ;              --
15165     typ := indices(1) ;
15166       if typ = 'any' then              -- Previous error.
15167           return expn ;
15168       end if ;
15169
15170       [-, lo, hi] := signature(typ);
15171       if not is_value(lo) or not is_value(hi) then
15172           static_bounds := false;
15173       else
15174           lo := lo(2);
15175           hi := hi(2);
15176           static_bounds := true;
15177       end if;
15178
15179-- Get and save the others choice
15180
15181       others := [ ];
15182       if #nam_list > 0 and nam_list(#nam_list)(1) = 'others' then
15183           others frome nam_list;
15184
15185           if (not static_bounds or not is_static_subtype(typ))
15186             and (#nam_list > 0 or #pos_list > 0) then
15187               errmsg('non-static "others" must appear alone',
15188                                                   '', '4.3.2(3)') ;
15189               return expn;
15190           end if ;
15191
15192       end if;
15193
15194-- At this point either nam_list or pos_list is empty and OTHERS has
15195-- been removed from nam_list and held separately in others.
15196
15197-- process named associations first.  Transform into positional ones.
15198-- All the named choices at this point must be static.
15199
15200       loe := hie := {};              -- Bounds for null range
15201       nam_map := {};
```

```
15202       duplicate_choice := false;
15203       (for [choice_list, cvalue] ∈ nam_list)
15204          (for c ∈ choice_list)
15205             if is_tuple(c) and c(1) = 'tname' then
15206                [-, ctyp] := c;
15207                if is_type(ctyp) then
15208                   [-,loc, hic] := signature(ctyp);
15209                else
15210                      loc := hic := om ;              -- not static.
15211                end if ;
15212             elseif is_tuple(c) and c(1) = 'range' then
15213                [-, loc, hic] := c;
15214             else
15215                loc := hic := c;
15216             end if;
15217
15218             if not is_value(loc) or not is_value(hic) then
15219                if #nam_list = 1 and #choice_list = 1
15220                and #pos_list = 0 and others = [ ]                --
15221                then
15222                      return expn;            -- Single choice
15223                else
15224                 errmsg('non_static choice must appear alone',
15225                                        '','4.3.2(3)');
15226                end if;
15227             else
15228                loc := loc(2);
15229                hic := hic(2);
15230                if loc > hic then                                 --
15231                   -- Null range: must be the only association. Keep
15232                   -- bounds to construct ghost element.
15233                  if #nam_list = 1 and #choice_list = 1
15234                  and #pos_list = 0 and others = [ ] then
15235                     loe with:= loc;
15236                     hie with:= hic;
15237                     else
15238                        errmsg('null range must appear alone', '',
15239                                                    '4.3.2(3)') ;
15240                     end if ;                                     --
15241                else
15242                   (for i ∈ [loc .. hic])
15243                      if nam_map(i) ≠ om then
15244                         duplicate_choice := true;
15245                      else
15246                         nam_map(i) := cvalue;
15247                      end if;
15248                  end for;
15249                end if;
15250
```

```
15251              end if;
15252           end for;
15253        end for;
15254
15255        if duplicate_choice then
15256           errmsg('Aggregate contains duplicate choices', '','4.3');
15257        end if;
15258
15259        if nam_map ≠ {} then          -- Some named associations
15260           lon := min / (domain nam_map);
15261           hin := max / (domain nam_map);
15262           if others = [ ] and hin - lon + 1 ≠ #nam_map then
15263              errmsg('Missing values in aggregate', '','4.3');
15264              return expn ;           -- dangerous to continue
15265           end if;
15266
15267           if others ≠ [ ] then
15268              if not static_bounds then
15269                 return expn;                 -- Don't know bounds
15270              else
15271                 lon min:= lo;
15272                 hin max:= hi;
15273              end if;
15274           else
15275              -- Verify that the named associations belong to the
15276              -- index subtype. If the context is not sliding, the
15277              -- bounds of the index apply. In any case, the bounds of
15278              -- the parent type of the index apply.
15279              -- (NB this is still incomplete: in the non-static case
15280              -- the interpreter does not have full information).
15281
15282              [-, lob, hib] := signature(type_of(typ)) ;
15283              if static_bounds and                              --
15284              ((lon < lo or hin > hi) and full_others
15285                 or (is_value(lob) and lon < lob(2))
15286                 or (is_value(hib) and hin > hib(2)))
15287                 then
15288                    return expn ;          -- Let interpreter check it.
15289              end if ;                                          --
15290           end if ;
15291
15292           ov := if others ≠ [ ] then others(2) else ['ivalue', 0] end;
15293           pos_list := [nam_map(i) ? ov : i ∈ [lon .. hin]];
15294
15295           if exists v ∈ pos_list | not is_value(v) then
15296              return expn;
15297
15298           else
15299              [-,sample_value ]:= pos_list(1);
```

```
15300             if is_tuple(sample_value)
15301             and sample_value(1) ='array_ivalue' then
15302               [-,-,lo_sample,hi_sample] := sample_value;
15303               if exists [-,[-,-,lox,hix]] ∈ pos_list |
15304                 ((lox ≠ lo_sample) or (hix ≠ hi_sample)) then
15305                 Warning('All subbaggregates must have identical '+
15306                       'bounds (4.3.2)');
15307                 return constraint_error();
15308               end if;
15309             end if;
15310--
15311               return check_null_aggregate([v(2) : v ∈ pos_list],
15312                                                   lon, indices)  ;
15313           end if;
15314         end if;
15315
15316-- In case we encountered only empty ranges while processing the
15317-- named associations, and no others choice has been used, set up
15318-- an appropriate null array ivalue, with a ghost element
15319
15320         if loe ≠ {} then
15321           if others = [ ] then
15322             [-,sample] := nam_list(1);
15323             if is_value(sample) then                    --
15324               return
15325                 ['ivalue', ['array_ivalue',
15326                       [make_ghost_out_of(sample(1))],   --
15327                       min / loe, max / hie, 'null']];
15328             else
15329               return expn;
15330             end if;
15331           else
15332               return expn;                  -- Probably not static
15333           end if;
15334         end if;
15335
15336-- At this point the aggregate is a positional aggregate with a possible
15337-- OTHERS choice.
15338
15339         if not static_bounds then
15340           return expn;                                  --
15341         end if;
15342
15343         if (others ≠ [ ] and not is_value(others(2))) or
15344               exists v ∈ pos_list | not is_value(v) then
15345           return expn;
15346         end if;
15347
15348         av := [v(2) : v ∈ pos_list];
```

```
15349       if others ≠ [ ] then
15350          av +:= [others(2)(2) : i ∈ [#pos_list + 1 .. hi-lo+1 ]];
15351       end if;
15352
15353--
15354       return check_null_aggregate(av, lo, indices) ;
15355
15356
15357
15358       -- Type conversion:
15359       -- - /TBSL/ These conversions are not properly checked!
15360    ('convert'):
15361       [-, typ1, typ2, opd] := expn;
15362
15363       if is_value(opd) then
15364          opd := opd(2);
15365       else
15366          if typ1 = 'universal_integer' then
15367             -- The expression is an attribute. We will assume that
15368             -- all such expressions fit in the bounds of INTEGER.
15369             expn(2) := 'INTEGER';
15370          end if ;
15371          return expn;
15372       end if;
15373       typ1 := root_type(typ1);
15374       typ2 := root_type(typ2);
15375       case typ1 of              -- from type
15376
15377          ('INTEGER'):
15378             case typ2 of
15379
15380                ('INTEGER'):
15381                   return ['ivalue', opd];
15382
15383                ('FLOAT'):
15384                   return ['ivalue', float opd];
15385
15386                ('universal_integer'):
15387                   return ['ivalue', (opd)];
15388
15389                ('universal_real', 'universal_fixed', '$FIXED'):
15390                   return ['ivalue', rat_fri(opd, 1)];
15391
15392             else
15393                   return expn;
15394             end case;
15395
15396          ('FLOAT'):
15397             case typ2 of
```

```
15398                ('INTEGER'):
15399                    [x, y] := rat_frr(opd + 0.5);
15400                    return check_overflow((x / y));
15401
15402                ('FLOAT'):
15403                    return ['ivalue', opd];
15404
15405                ('$FIXED', 'universal_real', 'universal_fixed'):
15406                    return ['ivalue', rat_frr(opd)];
15407
15408                ('universal_integer'):
15409                    [x, y] := rat_frr(opd + 0.5);
15410                    return ['ivalue', (x / y)];
15411
15412            else
15413                    return expn;
15414            end case;
15415
15416        ('universal_integer'):
15417            case typ2 of
15418                ('INTEGER'):
15419                    return check_overflow(opd);
15420
15421                ('FLOAT'):
15422                    return check_overflow([opd, 1]);
15423
15424                ('universal_integer'):
15425                    return ['ivalue', opd];
15426
15427                ('universal_real', 'universal_fixed', '$FIXED'):
15428                    return ['ivalue', rat_fri (opd, 1)];
15429
15430            else
15431                    return expn;
15432            end case;
15433
15434        ('universal_real', 'universal_fixed', '$FIXED'):
15435            case typ2 of
15436                ('FLOAT'):
15437                    return check_overflow(opd);
15438
15439                ('universal_real', 'universal_fixed', '$FIXED'):
15440                    return ['ivalue', opd];
15441
15442                ('INTEGER'):
15443                    return check_overflow(rat_toi(opd));
15444
15445            else
15446                    return expn;
```

```
15447                end case;
15448
15449          else
15450                return expn;
15451          end case typ1 ;
15452
15453     else
15454          return expn;
15455     end case ;                    -- For operators with 3 arguments.
15456     else
15457          return expn ;             -- For all other operators.
15458
15459     end if;
15460
15461     return expn ;                 -- In case we forgot anything.
15462
15463drop
15464     num, den, b_not, bool_op,
15465     is_simple_value;
15466
15467end proc eval_static;
15468proc eval_qual_range(op1,op2,expn);
15469--
15470-- This has been separated from the main body of eval_static because
15471-- it is used for two differents operators: 'qual_range' proper,
15472-- and 'in' and 'notin'
15473--
15474-- If the expression is not static it return the former expression expn.
15475-- If the expression evaluates to a ['raise','CONSTRAINT_ERROR'] because
15476-- op1 is not in the range op2, it returns the string 'contraint_error'
15477-- without emitting any warning; this is left to the caller
15478-- responsibility.
15479--
15480               if is_value(op2) then
15481                  [-, op2, typ] := op2 ;
15482               elseif is_tuple(op2) and
15483                        (op2(1) = 'subtype' or op2 = [ ]) then
15484--
15485-- This is wrong, but it is not clear how we should handle
15486-- constraints in slices and case choices. For example,
15487--   when st range 2 .. 4 => ... where st is a subtype.
15488-- If st is static a check could be made, but what do we do
15489-- if st is dynamic?  In any case we don't want 'qual_range'.
15490
15491                  return op2;
15492               else
15493                  return expn;
15494               end if;
15495               oop1 := op1 ;          -- May just be a type name.
```

```
15496          if is_tuple(op1) and op1(1) = 'subtype' and
15497                                  is_scalar_type(op1(2)) then
15498             [-, op1, [-, lo, hi]] := op1;
15499          elseif is_scalar_type(op1) then
15500             [-,lo,hi] := SIGNATURE(op1)?[ ];
15501          else
15502             return expn;
15503          end if ;
15504
15505          -- If the argument is universal, convert it to
15506          -- standard representation. A qual_range indicates
15507          -- a constrained type, i.e. non-universal.
15508          if is_universal_integer(op2) then
15509----        op2 := int_toi(op2) ;
15510             expn(3) := ['ivalue', op2, typ] ;
15511          elseif is_universal_real(op2)
15512             and root_type(op1) ≠ '$FIXED' then
15513             op2 := rat_tor(op2, ADA_REAL_DIGITS) ;
15514             expn(3) := ['ivalue', op2, typ] ;
15515          end if ;
15516
15517          if is_value(lo) and is_value(hi) then
15518             lo := lo(2);
15519             hi := hi(2);
15520          else
15521             return expn;
15522          end if;
15523
15524
15525          if op2 = 'OVERFLOW' then
15526             return constraint_error() ;
15527          end if ;
15528
15529          if oop1 ∈ {'INTEGER','FLOAT','$FIXED','CHARACTER'}
15530             or NATURE(oop1) = 'enum'
15531             -- Predefined type without constraint.
15532          then
15533             return ['ivalue', op2];
15534          else
15535
15536             -- At this point everything is known to be constant.
15537             -- If the constraint is obeyed, return the value without
15538             -- a range qualification. Otherwise emit a constraint
15539             -- exception.
15540             c_error :=
15541                  if root_type(op1) ≠ '$FIXED' then
15542                    (op2 < lo or op2 > hi)
15543                  else
15544                    (rat_lss(op2, lo) or rat_gtr(op2, hi))
```

```
15545                       end ;
15546                   if c_error then
15547                       return 'constraint_error';
15548                   else
15549                       return ['ivalue', op2];
15550                   end if;
15551               end if;
15552end proc eval_qual_range;
15553proc eval_real_type_attribute(expn);
15554  macro rat_1; [1, 1] endm; -- rational 1
15555  macro rat_2; [2, 1] endm; -- rational 2
15556--
15557--   Static evaluation of real types characteristics
15558--   ================================================
15559--
15560   macro test(x);
15561       if x then 1 else 0 end
15562   endm;
15563
15564  [-,attr,arg_type,-] := expn;
15565  result := om;
15566  sig := signature(arg_type);
15567  [ kind,-] := sig;
15568  case kind of
15569--
15570--   Part A : FLOATING POINT REAL
15571--
15572--   For a floating point real type FL, we have the folowing
15573--   basic informations:
15574--       digits  (SETL_integer)
15575--       fl_high (SETL_real)
15576--       fl_low  (SETL_real)
15577--
15578  ('digits'):
15579    [-,[-,fl_low],[-,fl_high],[-,fl_digits]] := sig;
15580    case attr of
15581
15582--====================
15583--
15584--       FL'DIGITS    → universal_integer
15585--
15586--           The minimum number of significant decimal digits.
15587--
15588    ('DIGITS'):
15589        result := fl_digits;                              --
15590
15591--====================
15592--
15593--       FL'MANTISSA  → universal_integer
```

```
15594--
15595--          The minimum number of binary digits required for DIGITS:
15596--               ceil(fl_digits*log(10)/log(2))+1)
15597--
15598    macro fl_mantissa;
15599       (ceil((float(fl_digits)*(log 10.0))/(log 2.0) + 1.0))
15600    endm;
15601    ('MANTISSA'):
15602       result := fl_mantissa;
15603
15604--====================
15605--
15606--       FL'EPSILON   → universal_real
15607--
15608--          The absolute value of the difference between the nuber 1.0
15609--          and the next model number above :
15610--               = 2.0**(1-FL'MANTISSA)
15611--
15612    ('EPSILON'):
15613       result := rat_exp(rat_2,(1 - fl_mantissa));
15614
15615--====================
15616--
15617--       FL'EMAX      → universal_integer
15618--
15619--          The largest exponent value in binary canonical form:
15620--               = 4*FL'MANTISSA
15621--
15622    macro fl_emax;
15623       (4 *  fl_mantissa)                                    --
15624    endm;
15625    ('EMAX','SAFE_EMAX'):
15626       result := fl_emax;
15627
15628--====================
15629--
15630--       FL'SMALL     → universal_real
15631--
15632--          The smallest positive non-zero number :
15633--               = 2.0**(- FL'EMAX -1)
15634--
15635    ('SMALL','SAFE_SMALL'):
15636       result := rat_exp(rat_2, -(fl_emax + 1));
15637
15638--====================
15639--
15640--       FL'LARGE     → universal_integer
15641--
15642--          The largest positive number:
```

```
15643--                 = 2.0**FL'EMAX * (1.0 - 2.0**(-FL'MANTISSA))
15644--
15645   ('LARGE','SAFE_LARGE'):
15646       result := rat_mul(
15647                   rat_exp(rat_2, fl_emax),
15648                   rat_sub(rat_1, rat_exp(rat_2, -(fl_mantissa))));
15649
15650--====================
15651--
15652--      FL'SAFE_EMAX  =  FL'BASE'EMAX
15653--      FL'SAFE_SMALL =  FL'BASE'SMALL
15654--      FL'SAFE_LARGE =  FL'BASE'LARGE
15655--
15656--       cf. FL'EMAX, FL'SMALL, FL'LARGE
15657
15658--====================
15659--
15660--      FL'MACHINE_ROUNDS → boolean
15661--
15662   ('MACHINE_ROUNDS'):
15663       result := test(false);
15664
15665--====================
15666--
15667--      FL'MACHINE_OVERFLOWS → boolean
15668--
15669   ('MACHINE_OVERFLOWS'):
15670       result := test(true);
15671
15672--====================
15673--
15674--      FL'MACHINE_RADIX     → universal_integer
15675--
15676   ('MACHINE_RADIX'):
15677       result := (2);
15678
15679--====================
15680--
15681--      FL'MACHINE_MANTISSA  → universal_integer
15682--
15683   ('MACHINE_MANTISSA'):
15684       result := (24);
15685
15686--====================
15687--
15688--      FL'MACHINE_EMAX      → universal_integer
15689--
15690   ('MACHINE_EMAX'):
15691       result := (127);
```

```
15692
15693--====================
15694--
15695--      FL'MACHINE_EMIN      → universal_integer
15696--
15697   ('MACHINE_EMIN'):
15698      result := (-128);
15699
15700--
15701   end case attr;
15702
15703--
15704--   Part B : FIXED POINT REAL
15705--
15706--   For a fixed point real type FX, we have the following basic
15707--   informations:
15708--      delta    (universal_real)
15709--      fx_low   (universal_real)
15710--      fx_high  (universal_real)
15711--   but the bounds may not be static...
15712--
15713  ('delta'):
15714   [-,exp_low,exp_high,[-,delta]] := sig;
15715   static_bounds := is_value(exp_low) and is_value(exp_high);
15716   if static_bounds then
15717      [-,fx_low] := exp_low;
15718      [-,fx_high] := exp_high;
15719   end if;
15720
15721   case attr of
15722--====================
15723--
15724--      FX'DELTA      → universal_real
15725--
15726--          The absolute value of the error bound.
15727--
15728   ('DELTA'):
15729      result := delta;
15730
15731--====================
15732--
15733--      FX'SMALL      → universal_real
15734--
15735--          The largest power of 2 not greater than the delta:
15736--             = 2.0**floor(log(delta)/log(2.0))
15737--
15738   macro fx_small;
15739      rat_exp(rat_2,
15740            (floor(log(rat_tor(delta,ADA_REAL_DIGITS))
```

```
15741                          /log(2.0))))
15742     endm;
15743     ('SMALL','SAFE_SMALL'):
15744        result := fx_small;
15745
15746--=====================
15747--
15748--       FX'MANTISSA  → universal_integer
15749--
15750--          The number of binary digits required:
15751--          = ceil(log(max(abs(fx_high),abs(fx_low))/FX'SMALL)/log(2.0)))
15752--
15753     macro fx_max;
15754       if rat_geq(rat_abs(fx_high),rat_abs(fx_low)) then
15755         rat_abs(fx_high)
15756       else
15757         rat_abs(fx_low)
15758       end
15759     endm;
15760     macro fx_mantissa;
15761        (ceil(log(rat_tor(rat_div(fx_max,fx_small),
15762                          ADA_REAL_DIGITS))/log(2.0)))
15763     endm;
15764     ('MANTISSA'):
15765        if static_bounds then
15766          result := fx_mantissa;
15767        end if;
15768
15769--=====================
15770--
15771--       FX'LARGE      → universal_real
15772--
15773--          The largest positive number :
15774--            = (2.0**FX'MANTISSA - 1) * FX'SMALL
15775--
15776     ('LARGE','SAFE_LARGE'):
15777        if static_bounds then
15778          result := rat_mul(
15779                rat_sub(rat_exp(rat_2,fx_mantissa),rat_1),
15780                fx_small );
15781        end if;
15782
15783--=====================
15784--
15785--       FX'FORE       → universal_integer
15786--
15787--          The minimum number of characters needed for the integer
15788--          part of the decimal representation (including sign).
15789--            = floor(log(max(abs(fx_high),abs(fx_low)))/log(10.0))+2.0)
```

```
16061  ( while count ≤  ( #array_of_nature  max #array_of_lines ))
16062    printa(REFFILE,( 19*' ' + (array_of_nature(count) ? (17*' '))
16063          + 16*' ' + (array_of_lines(count) ? (20*' ')) ));
16064   count+:= 1;
16065  end while ;
16066
16067  printa(REFFILE);
16068
16069end for ;
16070
16071end proc print_xref;
16072
```

sort

```
16074procedure sort(s);
16075
16076 if #s = 0 then return [ ];
16077 elseif #s = 1 then return [arb s];
16078 else  a from s;
16079       return ( sort( {l: l ∈ s | l(1) ≤ a(1) } )
16080              + [a]
16081              + sort( {g : g ∈ s | g(1) > a(1) } ));
16082 end if;
16083 end procedure sort ;
16084
16085 procedure quicksort(s);
16086
16087 if #s = 0 then return [ ];
16088 elseif #s = 1 then return [arb s];
16089 else  a from s;
16090       return ( quicksort( {l: l ∈ s | l < a} )
16091              + [a]
16092              + quicksort( {g : g ∈ s| g > a } ));
16093 end if;
16094 end procedure quicksort ;
16095
```

make_lines

```
16097proc make_lines(strin, avail_space);
16098
16099-- Creates one or two strings of length avail_space out of
16100-- the parameter string strin.  If #strin ≤ avail_space
16101-- then padded strin is returned, together with the flag true.
16102-- If strin does not fit on one line, it is broken at the appropriate
16103-- underline (_) and the rest of the strin is placed in the second line.
16104-- Both lines and the flag false are returned .
16105
```

```
16106if (num:=  #strin ) ≤ avail_space
16107then
16108   return [rpad(strin,avail_space),' ', true];
16109                          -- true here stands for one-liness.
16110end if;
16111
16112next_line := '';
16113(while #strin > avail_space)
16114    if strin(#strin)= '_' then
16115        strin:= strin(1..(#strin -1)); next_line := '_' + next_line;
16116    end if;
16117    next_line := rbreak(strin,'_') + next_line ;
16118end;
16119
16120if avail_space < #next_line
16121then strin +:= next_line(1..(num:= avail_space - #strin - 1)) + '-';
16122 next_line := next_line(num+1 .. (#next_line min (avail_space + num)));
16123end if;
16124
16125return [ rpad(strin,avail_space), rpad(next_line,avail_space), false ];
16126
16127end proc make_lines;
16128
```

format_numbers

```
16130proc format_numbers(line_nums, avail_space);
16131
16132-- Formats the line numbers as a tuple of strings each of length
16133-- avail_space.
16134
16135array_of_lines:= [ ];
16136i:= 1;
16137transit := str line_nums(1) ;
16138
16139(for num ∈ line_nums(2..) )
16140    num:= str num;
16141    if (#transit + #num + 2) ≤ avail_space
16142    then transit +:= ' ' + num;
16143    else  array_of_lines(i) := rpad(transit, avail_space);
16144        i +:= 1;  transit:= num;
16145    end if;
16146end;
16147
16148array_of_lines(i) := rpad(transit, avail_space);
16149
16150return array_of_lines ;
16151
16152end proc format_numbers;
```

```
16153

      format_subprog

16155proc format_subprog(u_name, avail_space);
16156
16157-- Called when the nature of the next item on the XREF list
16158-- is 'function' or 'procedure'. It locates the formal
16159-- parameters of the subprogram and returns them as a tuple
16160-- of strings, each of length of avail_space.
16161
16162array_of_specs := [ ];
16163array_of_specs with :=
16164          rpad( NATURE(u_name) , avail_space ) ;
16165counter := 1;
16166
16167(for form_parameter ∈ SIGNATURE(u_name) |
16168              is_tuple(form_parameter) and #form_parameter = 4 )
16169
16170  [name, p_mode, p_type, - ] := form_parameter ;
16171  name:= (if counter = 1 then '(' else '' end) +
16172            original_name(name) + ' : ' ;
16173  mode_n_type := p_mode + ' ' + full_type_name(p_type);
16174
16175  if #(name + mode_n_type) ≤ avail_space
16176  then array_of_specs with := rpad(name + mode_n_type, avail_space);
16177  else array_of_specs with := rpad(name, avail_space)(1..avail_space) ;
16178       s:= lpad(mode_n_type, avail_space);
16179       array_of_specs with := s( (#s - avail_space +1) .. #s );
16180  end if;
16181  counter +:= 1;
16182
16183end;
16184
16185if NATURE(u_name) = 'function'
16186then last_lin := rpad('RETURN : ' + full_type_name(type_of(u_name)),
16187                     avail_space );
16188     array_of_specs with := last_lin(1..#last_lin);
16189end if;
16190
16191if #array_of_specs = 1
16192then last_lin := ( rpad('(no parameters)' , avail_space)) ;
16193     array_of_specs with:=  last_lin ;
16194else
16195     last_lin frome array_of_specs ;
16196
16197     if last_lin(1) = ' ' then array_of_specs with:=
16198                          ( last_lin(2..#last_lin) + ')' );
16199     elseif top(last_lin) = ' ' then last_lin(#last_lin) := ')' ;
```

```
16200        array_of_specs with:= last_lin ;
16201    else array_of_specs with:= ( break(last_lin,':') + ':' );
16202        array_of_specs with:= ( last_lin(2..#last_lin) + ')' );
16203    end if;
16204end if;
16205
16206return array_of_specs ;
16207
16208end proc format_subprog;
16209
```

DROP SEMANTICS MACROS.

```
16211drop
16212         find,
16213         top,
16214         char_to_name,
16215         component_type,
16216         index_type,
16217         no_dimensions,
16218       literal_map,
16219         declared_components,
16220         record_declarations,
16221         all_components,
16222         discriminant_list,
16223         invariant_part,
16224         variant_part,
16225         has_discriminants,
16226         designated_type,
16227         init_scope_info,
16228         is_access,
16229         is_comp_unit,
16230         is_constant,
16231         is_empty,
16232         is_identifier,
16233         is_formal,
16234         is_generic,
16235         is_generic_type,
16236         is_literal,
16237         is_universal_integer,
16238         is_universal_real,
16239         is_value,
16240         is_overloaded,
16241         is_proc,
16242         is_task_type,
16243         is_type,
16244         is_array,
16245         is_record,
16246         is_integer_type,
16247         node_kind,
16248         private_decls,
16249         scope_info;
16250
16251end module ada - semantics;
16252
```