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

Ada/Ed INTERPRETER†

EXECUTABLE SEMANTIC MODEL 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.

DYNAMIC SEMANTIC ANALYSER

EXECUTABLE SEMANTIC MODEL FOR

A D A

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

module ada - interpreter;

Table of Contents

INTERPRETER

EXECUTABLE SEMANTIC MODEL FOR

A D A

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

module ada - interpreter;

Global declarations and definitions

Global macros for the use of all modules:

```
28 macro find;    assert exists          endm;
29
30 macro top(x); (x)(#(x)) endm;
```

Constants

```
32 const UNDEF_PRIO = 9;
```

Variables

```
34 var
35    GLOBAL_EMAP,    -- global EMAP at initialization -
36    DEBUGGING,
37    CURRENT_TIME,
38    LAST_CLOCK_TICK,
39    LAST_CLOCK_TICK_CURTASK,
40    NEXT_INTERRUPT_TIME,
41    TASK_COUNTS,
42    CURRENT_STMT_COUNT,
43    QUAL_MAP,
44    TASKNAME_MAP,
45    NULL,
46    FULL_TASKNAME_MAP,
47    TASKS_WITH_RAISED_EXC;
```

Utility macros

```
49 macro absent(arg); ((arg) = Ω) endm;      --
50
51 --
52 macro empty(arg); ((arg) = {}) endm;
53
54 macro nonempty(arg); ((arg) /= {}) endm; --
55
56 macro present(arg); ((arg) /= Ω) endm;
57
58 --
59 --
60  -- The following macros are used to test for various
61  --  internal interpreter objects. The intent is to make
```

```
62  -- the code somewhat clearer, and partially independant
63  -- of minor changes in representation.
64  --
65  macro location(t); ((t)(2)) endm; --
66  --
67  macro opcode(t); ((t)(1)) endm;
68  --
69  macro type_mark(t); ((t)(1)) endm;
70  --
71  macro is_access_iobject(t);
72    (is_tuple (t) and type_mark(t) = 'access_iobject')
73  endm;
74  --
75  macro is_access_ivalue(t);
76    (is_tuple (t) and type_mark(t) = 'access_ivalue')
77  endm;
78  --
79  macro is_ais_stmt;
80    (opcode(STM)(#opcode(STM)) /= '_')
81  endm;
82  --
83  macro is_array_iobject(t);
84    (is_tuple (t) and type_mark(t) = 'array_iobject')
85  endm;
86  --
87  macro is_array_ivalue(t);
88    (is_tuple (t) and type_mark(t) = 'array_ivalue')
89  endm;
90  --
91  macro is_constrained_type(t);
92    (if type_mark(t) = 'array'
93     then (t)(4)
94     elseif type_mark(t) = 'record'
95        then present((t)(3))
96        else false
97        end )
98  endm;
99  --
100 macro is_discr_ref(t);
101   (is_tuple (t) and type_mark(t) = 'discr_ref' )
102 endm;
103 --
104 macro is_fixed_iobject(t);
105   (is_tuple(t) and type_mark(t) = 'fixed_iobject')
106 endm;
107 --
108 macro is_fixed_ivalue(t);
109   (is_tuple (t) and #(t) = 2 and
110   is_integer (t)(1) and is_integer (t)(2)) --
```

```
111 endm;
112 --
113 macro is_instruction(t);  (is_tuple (t) and is_string opcode(t) ) endm;
114 --
115 --
116 macro is_label_iobject(t);
117   (is_tuple (t) and type_mark(t) = 'label_iobject')
118 endm;
119 --
120 macro is_label_ivalue(t);
121   (is_tuple (t) and type_mark(t) = 'label_ivalue')
122 endm;
123 --
124 macro is_literal_ivalue(t);
125   (is_integer (t) or is_real (t) or (t) = UNINITIALIZED )
126 endm;
127 --
128 macro is_location(t);  (is_atom (t)) endm;
129 --
130 macro is_operation(t);  (is_tuple (t) and is_string opcode(t) ) endm;
131 --
132 macro is_proc_iobject(t);
133   (is_tuple (t) and type_mark(t) = 'proc_iobject')
134 endm;
135 --
136 macro is_proc_ivalue(t);
137   (is_tuple (t) and type_mark(t) = 'proc_ivalue')
138 endm;
139 --
140 macro is_range(x);
141   (is_tuple(x) and (x)(1) = 'range')
142 endm;
143 --
144 --
145 macro is_record_iobject(t);
146   (is_tuple (t) and type_mark(t) = 'record_iobject')
147 endm;
148 --
149 macro is_record_ivalue(t);
150   (is_tuple (t) and type_mark(t) = 'record_ivalue')
151 endm;
152 --
153 macro is_simple_name(t);  (is_string (t)) endm;
154 --
155 macro is_task_ivalue(t);
156   (is_string (t) and (t)(1..5) = '$task')
157 endm;
158 --
159 macro is_task_iobject(t);
```

```
160   (is_tuple (t) and type_mark(t) = 'task_iobject')
161 endm;
162 --
163 macro is_uninitialized(arg);
164    (arg) = ['ivalue', UNINITIALIZED ]
165 endm;
166 --
167 --
168  -- These macros allow the use of the rational arithmetic
169  -- package to do fixed arithmetic.
170 --
171 macro fix_abs(u);   RAT_ABS(u)   endm; -- fixed absolute value
172 macro fix_add(u, v); RAT_ADD(u, v) endm; -- fixed add
173 macro fix_div(u, v); RAT_DIV(u, v) endm; -- fixed divide
174 macro fix_eql(u, v); RAT_EQL(u, v) endm; -- fixed equal
175 macro fix_exp(u, v); RAT_EXP(u, v) endm; -- fixed exponentiation
176 macro fix_fri(i);   [i, 1]         endm; -- Integer to rational conv. -
177 macro fix_frr(u);   RAT_FRR(u)   endm; -- real to fixed      -
178 macro fix_frs(u);   RAT_FRS(u)   endm; -- fixed from string
179 macro fix_geq(u, v); RAT_GEQ(u, v) endm; -- fixed >=
180 macro fix_gtr(u, v); RAT_GTR(u, v) endm; -- fixed >
181 macro fix_leq(u, v); RAT_LEQ(u, v) endm; -- fixed <=
182 macro fix_lss(u, v); RAT_LSS(u, v) endm; -- fixed <
183 macro fix_mul(u, v); RAT_MUL(u, v) endm; -- fixed multiply
184 macro fix_neq(u, v); RAT_NEQ(u, v) endm; -- fixed /=
185 macro fix_sub(u, v); RAT_SUB(u, v) endm; -- fixed minus
186 macro fix_toi(u);   RAT_TOI(u)   endm; -- fixed to integer
187 macro fix_tor(u);   RAT_TOR(u, ADA_REAL_DIGITS) endm; -- fixed to real
188 macro fix_tos(u, n); RAT_TOS(u, n) endm; -- fixed to string
189 macro fix_umin(u);  RAT_UMIN(u)  endm; -- fixed unary minus
190 --
191 --
192  -- These macros support the representation of false as 0
193  --  and true as 1
194 --
195 macro boolean_false;  0 endm;
196 --
197 macro boolean_true;  1 endm;
198 --
199 macro test(t);
200   if (t) then boolean_true
201          else boolean_false
202   end
203 endm;
204 --
205 macro cond_schedule; if sched_flag then SCHEDULE; end if endm;
206
207 macro convert_duration(d); (fix (fix_tor(d) * 1000.0)) endm;
208
```

```
209 macro get_bounds_of(t);
210 --
211  -- get_bounds_of is a convenience for range testing.
212  -- Parameter t is an itype.
213 --
214   case type_mark(t) of
215   ('integer'):
216     [-, lbd, ubd] := (t);
217   ('enum'):
218     [-, lbd, ubd, enumlist] := (t);       --
219   ('fixed'):
220     [-, lbd, ubd, delta] := (t);
221   ('float'):
222     [-, lbd, ubd, digits] := (t);
223 --
224   ('range'):
225     [-, lbd, ubd] := (t);
226   else
227     [lbd, ubd, digits, delta] := [0, 0, 0, fix_fri(0)]; --
228     SYSTEM_ERROR('get_bounds_of');
229   end case
230 endm;
231 --
232 --
233 procedure push_entryid(entryexpr);
234   if is_simple_name(entryexpr) then
235     exec([['push_', entryexpr],         --
236           ['push_', 1 ]] );
237   else
238     if entryexpr(1) = '.' then
239       [-, -, e_name] := entryexpr;
240     else
241       e_name := entryexpr;
242     end if;
243     if is_simple_name(e_name) then
244       exec([['push_', e_name],
245             ['push_', 1] ]);
246     else
247       [-, entry, [index]] := e_name;
248       exec([['push_', entry],
249             ['veval_', index] ]);
250     end if;
251   end if;
252 end procedure;
253 --
254 ---T+ DOCUMENTATION OF THE MODULE
```

TYPE MODEL IN THE INTERPRETER

```
256 --
257 -- First a summary of the terminology. There are three kinds of
258 -- things which are manipulated at run time:
259 --
260 --  IVALUE   Internal value, corresponding to the Ada
261 --           notion of value.
262 --
263 --  IOBJECT  Internal object, corresponding to the Ada
264 --           notion of object.
265 --
266 --  ITYPE    Internal type, corresponding to some mixture of
267 --           the Ada concepts of type and subtype. An ITYPE
268 --           provides all the information for construction
269 --           of an IOBJECT of the corresponding type, and this
270 --           is one of the main uses of ITYPE's at run time.
271 --
272 --
273 -- Corresponding to these three kinds of things are three kinds
274 -- of expressions:
275 --
276 --  VEXPR    which when evaluated yields an IVALUE
277 --
278 --  OEXPR    which when evaluated yields an IOBJECT
279 --
280 --  TEXPR    which when evaluated yields an ITYPE
281 --
282 -- An important principle is that the interpreter can always tell
283 -- from context which of the three expression types it is evaluating.
284 --
285 -- Note that there is some overlap between the forms of OEXPR and
286 -- VEXPR expressions. For example, an array reference could be
287 -- either one or the other. In the interpreter there are three
288 -- separate expression evaluators ('veval_', 'oeval_', and 'teval_')
289 -- which take an expression of the appropriate type, and return a
290 -- result of the corresponding type. Note that in this description,
291 -- value and object are informal terms which can be used in any
292 -- of the three cases, we always use IOBJECT and IVALUE when
293 -- specifically referring to the corresponding object.
294 --
295 --
296 -- NOTE: The following design principle is adopted throughout this
297 -- document. If an object is represented as a SETL Tuple, and the
298 -- first element is a string, then the value of this string uniquely
299 -- identifies the abstract type of the object.
300 --
301 --
```

ITYPE (internal type)

```
303 --
304 -- An ITYPE is an internal representation of a type, corresponding
305 -- roughly to the union of the notions of type and subtype in Ada.
306 -- However, only the structural information is used at run time,
307 -- since all type identity checking is performed by the front end.
308 -- An ITYPE is obtained at run time by evaluating a TEXPR
309 -- and is used for creating IOBJECT's and in some other
310 -- expression contexts (e.g. doing the range check for the in
311 -- and notin operators).
312 --
313 --
314 -- Integer ITYPE
315 --
316 --    ['integer', lower_bound_ivalue, upper_bound_ivalue]
317 --
318 --    The bounds are IVALUE's specifying the values of the
319 --    first and last entries in the range.
320 --
321 --
322 -- Float ITYPE
323 --
324 --    ['float', lower_bound_ivalue, upper_bound_ivalue, digits_ivalue]
325 --
326 --    The bounds values here are both float IVALUE's which give
327 --    the lower and upper bounds of the range. Digits_ivalue is an
328 --    integer number of digits.
329 --
330 --
331 -- Fixed ITYPE
332 --
333 --    ['fixed', lower_bound_ivalue, upper_bound_ivalue, delta_ivalue]
334 --
335 --    The bounds values and the delta value are fixed IVALUES.
336 --
337 --
338 -- Enumeration ITYPE
339 --
340 --    ['enum', lower_bound_ivalue, upper_bound_ivalue, literal_map]
341 --
342 --      literal_map  ::=  {index_value → literal_name}
343 --
344 --    The bounds values are integer IVALUE's. The index_value appearing
345 --    in the literal_map is a SETL integer in the range of the bounds,
346 --    and the literal-name is a SETL string giving the literal name.
347 --
348 --
```

```
349 -- Access ITYPE
350 --
351 --   ['access', itype]
352 --
353 --   The itype entry specifies the ITYPE of the access IOBJECT.
354 --
355 --
356 -- Task ITYPE
357 --
358 --    ['task', entryids, priority, taskbody]
359 --
360 --    Entryids is a tuple of pairs [ename, index] where ename is the
361 --    entry name and index is one for ordinary entries, and is the
362 --    entry's number if it is in an entry family. Priority is the
363 --    task's priority. Taskbody is the name of an entry in EMAP that
364 --    contains the body of the task. Due to considerations in the
365 --    binder, the body of the task must be provided separatly from the
366 --    declaration of the task type which creates the ITYPE entry in
367 --    EMAP. The body of the task is entered by a task_body statement.
368 --
369 --
370 --
371 -- Array ITYPE
372 --
373 --    ['array', index_itype_list, element_itype, is_constrained]
374 --
375 --    The index_itype is the ITYPE for an integer which represents
376 --    the type of the index. A special case occurs when a dynamic
377 --    array (with at least one unconstrained bound) is involved.
378 --    In this case, the index_itype has a special form given
379 --    below. The element_itype specifies the ITYPE of the elements.
380 --    The is_constrained field is true when the index_itype is
381 --    constrained, and false when it is not.
382 --
383 --
384 -- Dynamic Bounds ITYPE
385 --
386 --    [discrete_type, lower_bound, upper_bound]
387 --
388 --    This ITYPE form appears only as the index_itype for a dynamic
389 --    array. The bounds are either discrete IVALUE's (in the case
390 --    where the bound is not dynamic), or they have the form:
391 --
392 --      ['discr_ref', discrim_name, discrim_type]
393 --
394 --    indicating that the corresponding bound is dynamic and in this
395 --    case, discrim_name is the name of the corresponding discriminant.
396 --    At least one of the two bounds must have this form to indicate
397 --    the case of a dynamic array index type.
```

```
398 --
399 --
400 -- Record ITYPE
401 --
402 --  ['record', component_list, constr_disc_list, disc_list]
403 --
404 --    component_list  ::=  [objdec_list, variant_part]
405 --    constr_disc_list::=  {constr_discr_name} | Ω | ['discr_ref', dmap]
406 --    disc_list       ::=  {discrim_name}
407 --    dmap            ::=  {discrim_name → discrim_value}
408 --    discrim_value   ::=  ivalue | ['discr_ref', discrim_name]
409 --    objdec_list     ::=  [[field_name, [field_itype, field_vexpr], ...]
410 --    variant_part    ::=  [variant_name, altern_list] | [ ]
411 --    altern_list     ::=  { [{choice}, component_list], ... }
412 --    choice          ::=  choice_ivalue | ['range', lbd, ubd]
413 --                                       | 'others'
414 --
415 --  The field_name is a SETL string giving the name of the field,
416 --  whose ITYPE is specified by field_itype, and whose initial
417 --  value is specified by field_vexpr, which is omitted if there
418 --  is no initial value .
419 --
420 --  The variant_name is a SETL string giving the name of the
421 --  discriminant field for a record variant. This is one of
422 --  the names appearing as a field_name in the objdec_list.
423 --  If there is no variant, then the variant_part is a null
424 --  tuple. If there is a variant, then the altern_list gives
425 --  a map from sets of discriminant values to component_lists
426 --  which give the corresponding fields. Note that since
427 --  component_list can itself contain a variant_part that
428 --  nested variants are represented by the corresponding
429 --  nested structure.
430 --
431 --  The choice_ivalue entries are IVALUE's for choices in the
432 --  non range case, and the 'range' entries are ITYPE's
433 --  for discrete types giving the bounds in the case where
434 --  a range is given. The 'others' choice, if it appears, must
435 --  be the last in the altern_list.
436 --
437 --  For a record type declaration, constr_disc_list is Ω. For
438 --  a record subtype, created by a discriminant constraint, the
439 --  constr_disc_list contains the names of all the discriminants,
440 --  since RM 3.7.2 requires that a discriminant constraint provide
441 --  values for all discriminants. For a constrained record ITYPE
442 --  which is the ITYPE of a component of another record, one or more
443 --  of whose constraints are discriminants of the enclosing record,
444 --  constr_disc_list has the form ['discr_ref', dmap]. The discrim_names
445 --  in dmap are the names of discriminants in the enclosing record.
446 --
```

```
447 --  For all record types disc_list is the set of discriminant names.
448 --  This set is empty if the type has no discriminants.  In general
449 --  this is redundant information, but is needed in those cases when
450 --  a value is used to constrain the discriminants -- specifically,
451 --  constant objects and allocated objects.  A discriminant constraint
452 --  cannot be explicitly formulated here because the value need not
453 --  be an aggregate.
454 --
```

IOBJECT (internal object)

```
456 --
457 -- An internal object corresponds to the Ada notion of
458 -- object, and is a location which may or may not correspond to
459 -- a value. IOBJECT's appear in the range of EMAP, as subparts of
460 -- record and array IOBJECT's, and in the range of CONTENTS for
461 -- the case of access OBJECT's (whose IVALUE contains an IOBJECT).
462 --
463 --
464 --  Scalar IOBJECT
465 --
466 --    A scalar IOBJECT (integer, float, enumeration )
467 --    is a SETL atom. If the IOBJECT has an IVALUE, then it appears in
468 --    the domain of CONTENTS, mapped to the IVALUE representing
469 --    the value of the object.
470 --
471 --  Fixed IOBJECT
472 --
473 --    ['fixed_iobject', location]
474 --
475 --    The location is a SETL atom. CONTENTS(location) is a pair
476 --    [num, den], which is a rational number formated accrding to
477 --    the requirements of the rational arithmetic package (q.v.) .
478 --
479 --  Access IOBJECT
480 --
481 --    ['access_iobject', location]
482 --
483 --    The location is a SETL atom. CONTENTS(location) is the IOBJECT
484 --    which is the current value of this object, or NULL, or
485 --    UNINITIALIZED.
486 --
487 --
488 --  Task IOBJECT
489 --
490 --    ['task_iobject', location]
491 --
492 --    The location is a SETL atom. CONTENTS(location) is the taskid,
493 --    a unique task identifier generated for each object created of a
```

```
494 --    task type. The taskid is used in interpreter tables containing
495 --    information about the task. Note that the rules of the language
496 --    require tasks always be declared as constants.
497 --
498 --  Procedure IOBJECT
499 --
500 --    ['proc_iobject', taskid_loc, index_loc, formals_loc, stsq_loc]
501 --
502 --    where CONTENTS(taskid_loc) is the task in which the procedure was
503 --    defined, and CONTENTS(index_loc) is an environment index (as for
504 --    labels). CONTENTS(formals_loc) is a list (tuple) of triples, one
505 --    for each formal parameter, in the form [name, pmode, type]. The
506 --    first element of the triple, name, is the name of the parameter.
507 --    The second element, pmode, is in the set {'in', 'inout', 'out'} and
508 --    indicates the parameter type. The third component, type, is the
509 --    name of the type or subtype of the parameter. CONTENTS(stsq_loc)
510 --    is the statement sequence for the procedure. Note that the rules
511 --    of the language require procedures always to be declared as
512 --    constants. (See the proc operator in veval_ for further details.)
513 --
514 --
515 --  Label IOBJECT
516 --
517 --    ['label_iobject', location]
518 --
519 --    The location is a SETL atom. CONTENTS(location) is the tuple:
520 --
521 --       [index, stsq]
522 --
523 --    where index is an environment index (see description of ENVSTACK),
524 --    and stsq is the sequence of statements following the label, i.e.
525 --    the sequence of statements to be executed following a goto
526 --    referencing the label. (See goto and the labdef statements
527 --    for further details.)
528 --
529 --  Non-dynamic Array IOBJECT
530 --
531 --    The IOBJECT for a non-dynamic array (i.e. an array other than
532 --    one whose bounds are specified by non constrained discriminants)
533 --    appears as:
534 --
535 --      ['array_iobject', seq, lowbound, highbound]
536 --
537 --    The bounds are SETL atoms which are mapped in CONTENTS to the
538 --    integer IVALUE's of the corresponding array subscript bounds.
539 --    The entry seq is a SETL tuple, whose components are IOBJECT's
540 --    corresponding to the elements.
541 --
542 --
```

```
543 -- Dynamic Array IOBJECT
544 --
545 --   The IOBJECT for a dynamic array (i.e. an array which has at least
546 --   one bound specified by a non constrained discriminant) appears
547 --   as:
548 --
549 --     ['array_iobject', seq, lowbound, highbound, lowdiscr, highdiscr]
550 --
551 --   The bounds are SETL atoms which are mapped in CONTENTS to the
552 --   integer IVALUE's of the corresponding array subscript bounds.
553 --   In the case of a dynamic bound, it is possible for the bound
554 --   to be undefined if the corresponding discriminant value is
555 --   not given. In this case, no entry exists for the bound in
556 --   CONTENTS.
557 --
558 --   The seq entry is a SETL tuple containing IOBJECT's for the
559 --   elements of the array. The length of this tuple is determined
560 --   by the possible discriminant values to be the longest which
561 --   could ever be required. At a given point in time, only part
562 --   of this sequence will be accessible (as indicated by the
563 --   current bounds settings).
564 --
565 --   The discriminant names are SETL strings which give the field
566 --   names of the corresponding discriminants. At least one of these
567 --   must be present (to distinguish this as a dynamic array).
568 --
569 --
570 -- Record IOBJECT
571 --
572 --   The IOBJECT for a RECORD appears as:
573 --
574 --     ['record_iobject', objdeclist, constr_disc_list]
575 --
576 --       objdeclist       ::=  {field_name → field_iobject}
577 --       constr_disc_list ::=  {constr_discr_name}
578 --
579 --   The domain elements field_name are SETL strings giving the name
580 --   of the corresponding field in the record. The field_iobject value
581 --   is an IOBJECT for the corresponding field.
582 --
583 --   If this record is a constrained variant of another record type,
584 --   the constr_disclist is a list of field names of discriminants.
585 --   This is used for checking discriminant constraint violations in
586 --   assignments.
587 --
588 --
589 -- Dynamic Record IOBJECT
590 --
591 --   ['record_iobject', objdeclist]
```

```
592 --
593 --  A dynamic record IOBJECT is one in which the discriminants are
594 --  unconstrained. The objdeclist contains entries for all fields in
595 --  all possible variants.
596 --
597 --  The dynamic record form is distinguished by the presence of
598 --  an extra component in the objdeclist called fields_present .
599 --  The fields_present component is a special object which is a SETL
600 --  atom. The value of this object, which is stored in CONTENTS, is
601 --  a set of names of fields which are actually present for the current
602 --  settings of the discriminants controlling variants. An entry in
603 --  objdeclist should only be considered to be valid if the
604 --  corresponding field name is in CONTENTS(objdeclist(fields_present)).
605 --
606 --
```

IVALUE (internal value)

```
608 --
609 -- An IVALUE corresponds to the Ada notion of value. Typically
610 -- values appear as the result of evaluating expressions or
611 -- obtaining the value of an object.
612 --
613 --
614 -- Integer IVALUE
615 --
616 --  An integer IVALUE is simply the corresponding SETL integer
617 --  value, and is (in the VAX implementation) constrained to
618 --  lie in the range -2**30 to +2**30.
619 --
620 --
621 -- Enumeration IVALUE
622 --
623 --  An enumeration IVALUE is simply the SETL integer corresponding
624 --  to the index (starting from zero) of the value in the range of
625 --  the enumeration base type.
626 --
627 --
628 -- Float IVALUE
629 --
630 --  A float IVALUE is simply the corresponding SETL real value
631 --  and is constrained by the accuracy and range of the SETL
632 --  float values.
633 --
634 --
635 -- Fixed IVALUE
636 --
637 --   [num, den]
638 --
```

```
639 --  A fixed IVALUE is a pair [num, den] whose format is as
640 --  specified in the rational arithmetic package (q.v.).
641 --  Note that checks are not needed to ensure that the accuracy
642 --  provided by this representation is enough to correspond to the
643 --  requirement specified by the delta value, since the rational
644 --  package allows arbitrarily large precision.
645 --
646 --
647 -- Access IVALUE
648 --
649 --    ['access_ivalue', accessed_iobject]
650 --
651 --  Accessed_iobject is the IOBJECT for the referenced object.
652 --
653 -- Task IVALUE
654 --
655 --    ['task_ivalue', taskid]
656 --
657 --  See Task IOBJECT for definition of taskid.
658 --
659 -- Procedure IVALUE
660 --
661 --    ['proc_ivalue', taskid, index, formals, stsq]
662 --
663 --  See Procedure IOBJECT for definitions.
664 --
665 -- Label IVALUE
666 --
667 --    ['label_ivalue', index, stsq]
668 --
669 --  See Label IOBJECT for definitions.
670 --
671 -- Array IVALUE
672 --
673 --  An array IVALUE has the form:
674 --
675 --    ['array_ivalue', seq, lowbound, highbound]
676 --
677 --  where seq is a SETL tuple containing the IVALUE's of the
678 --  components, lowbound is the IVALUE of the lower bound,
679 --  and highbound the IVALUE of the upper bound.
680 --
681 --
682 -- Record IVALUE
683 --
684 --  A record IVALUE has the form:
685 --
686 --    ['record_ivalue', {field → field_ivalue}]
687 --
```

```
688 -- The second component is a SETL map which maps field names into
689 -- their corresponding IVALUE's.
690 --
```

REPRESENTATION OF ADA EXECUTION ENVIRONMENT

```
692 --
693 -- The current execution environment for a single task is defined
694 -- by the top of a stack called ENVSTACK. The stacks for all the active
695 -- tasks in the system are contained in a map, called ENVSTACKT, from
696 -- task identities to the ENVSTACK for that task. Each slot in an
697 -- ENVSTACK contains five items: EMAP, STSQ, VALSTACK, HANDLER, and
698 -- TASKS_DECLARED. Access to the top of the stack for each task is
699 -- provided by a set of macros (EMAPT, STSQT, etc.) which make it appear
700 -- that the items in the top slot are in five separate maps, indexed by
701 -- the task identity. A further set of macros is provided that allow the
702 -- items for the currently executing task (CURTASK) to be refered to
703 -- simply by their names (i.e. EMAP, STSQ, etc.). Thus, code that does
704 -- not control tasking itself, or intertask communication can be written
705 -- as if it were for a single sequential process. The details are as
706 -- follows:
707 --
708 -- Environment of all active tasks:
709 --
710 --  CONTENTS
711 --          A map from location identifiers (atoms) to values which
712 --          models the memory of the object machine. In this
713 --          interpreter, no attempt is made to reuse memory
714 --          locations, so CONTENTS just grows as execution
715 --          proceeds, discarded values become inaccessible because
716 --          their location identifiers are no longer available.
717 --
718 --          The special value UNINITIALIZED (which is an atom) is
719 --          used to represent an uninitialized value.
720 --
721 --  ENVSTACKT
722 --              A map showing the ENVSTACK value for each task.
723 --
724 --
725 --  EMAPT
726 --              A map showing the current environment for each task.
727 --              The initial environment for a task which has just been
728 --              activated is the result of elaborating the declarations
729 --              of the task body.
730 --
731 --
732 --  STSQT
733 --              A map showing the current statement sequence for
734 --              each task. Set to the statements of the task body for
```

```
735 --              a task  which has just been activated.
736 --
737 --
738 --  VALSTACKT
739 --              A map  showing the  VALSTACK value for each  task. Set
740 --              to [ ] for a task which has just been activated.
741 --
742 --  HANDLERT
743 --              A map showing the current exception handler for each
744 --              task.
745 --
746 --
747 --  TASKS_DECLAREDT
748 --              A set containing the set of task objects declared
749 --              but not yet activated for the current environment
750 --              of each task.
751 --
752 --
753 -- Environment for a single task :
754 --
755 --  ENVSTACK
756 --              A tuple whose successive entries are saved environments
757 --              for previously stacked procedures. On a procedure call,
758 --              block entry, or other operation  which creates a  new
759 --              environment, an entry is made  at the  top of ENVSTACK,
760 --              consisting of a five element tuple:
761 --                 [emap, stsq, valstack, handler, tasks_declared]
762 --              This  environment is restored  on leaving the new
763 --              environment. The environment index  used in the emap
764 --              entry for a label and in certain other contexts is the
765 --              length of ENVSTACK for the corresponding environment.
766 --
767 --
768 --  Components of ENVSTACK (items):
769 --
770 --
771 --  EMAP
772 --              A  map  from identifers for types, constants, procedures
773 --              and variables to the objects representing them. EMAP
774 --              contains the following types of entry:
775 --
776 --                 EMAP (name)  →  IOBJECT
777 --                 EMAP (name)  →  ITYPE
778 --
779 --              The structure of the intermediate code and the
780 --              interpreter ensures that whenever an EMAP reference is
781 --              made, it is statically known which of the four cases is
782 --              involved. Thus for example if a name appears as a VEXPR,
783 --              then it is the case of a reference to an IOBJECT whose
```

```
784 --          value is to be returned.
785 --
786 --  STSQ
787 --          A sequence (tuple) of statements remaining to be
788 --          executed in the current environment. The next statement
789 --          to be executed is always at the head of this sequence.
790 --          Since compound statements are represented as single
791 --          statements, STSQ is essentially a tree which is
792 --          manipulated as the execution proceeds to reflect the
793 --          control. The exact format of the various statement
794 --          forms is documented in the main interpreter loop.
795 --
796 --
797 --  VALSTACK
798 --          A stack which is used to hold partial results of
799 --          expressions as they are evaluated. VALSTACK is also
800 --          used to hold arguments for procedure calls.
801 --
802 --
803 --
804   macro HEIGHT(task);          #ENVSTACKT(task)                endm;
805 --
806   macro EMAPT(task);          ENVSTACKT(task)(HEIGHT(task))(1) endm;
807   macro STSQT(task);          ENVSTACKT(task)(HEIGHT(task))(2) endm;
808   macro VALSTACKT(task);      ENVSTACKT(task)(HEIGHT(task))(3) endm;
809   macro HANDLERT(task);       ENVSTACKT(task)(HEIGHT(task))(4) endm;
810   macro TASKS_DECLAREDT(task); ENVSTACKT(task)(HEIGHT(task))(5) endm;
811   macro PROC_NAMET(task);     ENVSTACKT(task)(HEIGHT(task))(6) endm;
812 --
813 --
814   macro ENVSTACK;      ENVSTACKT(CURTASK)      endm;
815   macro EMAP;          EMAPT(CURTASK)          endm;
816   macro STSQ;          STSQT(CURTASK)          endm;
817   macro VALSTACK;      VALSTACKT(CURTASK)      endm;
818   macro HANDLER;       HANDLERT(CURTASK)       endm;
819   macro TASKS_DECLARED; TASKS_DECLAREDT(CURTASK) endm;
820   macro PROC_NAME;     PROC_NAMET(CURTASK)     endm;
821 --
822 --
823   macro POP_ENVSTACK;  ENVSTACK(HEIGHT(CURTASK)) := om endm;
824   macro PUSH_ENVSTACK; ENVSTACK with:= [EMAP, [ ], [ ], [ ], {}, ''] endm;
825   macro TOP_VALSTACK;  VALSTACK(#VALSTACK)              endm;
826 --
827   proc push(item);
828     VALSTACK with:= item;
829   end proc push;
830   proc pop(wr item);
831     item frome VALSTACK;
832   end proc pop;
```

```
833 --
834 --
```

IMPLEMENTATION OF TASKING

```
836 --
837 -- The following variables are used to implement the tasking mechanism:
838 --
839 --  ACTIVE_TASKS
840 --          A  set  whose elements  are the identities (atoms)  of
841 --          all  active  tasks  (i.e.  those  which  have  been
842 --          initiated and not  yet terminated). A task in this set
843 --          can be in any of the following queues:
844 --
845 --            ready       - ready to be executed when scheduled
846 --            delayed     - waiting for completion of delay
847 --            waiting     - waiting for access to a task entry
848 --            entering    - engaged in a  rendezvous
849 --            held        - waiting for subtasks to complete
850 --            terminatable - waiting at a selective wait with
851 --                           an open terminate alternative
852 --            open        - waiting for call on entry
853 --            activating  - waiting for subtasks to be all
854 --                          activated
855 --
856 --          Usually a task is in  exactly one  of these queues at
857 --          any  one time. The exception is that a task can be
858 --          open and delayed, or open and terminatable during
859 --          execution of a  selective wait statement, or be waiting
860 --          and delayed during execution of a timed entry call.
861 --
862 --
863 --  READY_TASKS
864 --          A  tuple whose  elements are identities of
865 --          those active tasks which are ready to be executed.
866 --
867 --
868 --  CURTASK
869 --          The identity  of the  currently executing  task.
870 --          Included in READY_TASKS unless it has just become
871 --          blocked.
872 --
873 --  WAITING_TASKS
874 --          A map from tasks to another map from entrynames to
875 --          task queues, where a task queue is is tuple of
876 --          pairs [taskid, stmseq], where taskid is the
877 --          identity of a task queued on the entry, and stmseq
878 --          is a list of statements to be inserted in front of
879 --          STSQ after the rendezvous is complete. For example,
```

```
880 --          WAITING_TASKS(t)(e)(1) is the first task queued on
881 --          entry e of task t.
882 --
883 --
884 --  HELD_TASKS
885 --          A set whose elements are the identities of tasks
886 --          which are waiting for completion of subtasks before
887 --          proceeding. These are tasks executing a return, end or
888 --          terminate statement.
889 --
890 --
891 --  ACTIVATING_SUBTASKS
892 --          A map whose domain is tasks waiting for their subtasks
893 --          to be activated, i.e. to complete their elaboration,
894 --          and whose range is the ids of the activating subtasks.
895 --
896 --
897 --  TERMINATABLE
898 --          A set whose elements are the identities of tasks
899 --          which are waiting at a selective wait statement
900 --          with an open terminate alternate.
901 --
902 --
903 --  OPEN_ENTRIES
904 --          A map from task identities onto sets of pairs
905 --          [entryname, [bodyr, body]], another map, which indicates
906 --          which entries are currently open, and for each open
907 --          entry, the body of statements for the rendezvous
908 --          (bodyr) and the
909 --          block of statements to be executed following the
910 --          rendezvous (body, non-null only in the case of an
911 --          accept within a select).
912 --
913 --
914 --  DELAYED_TASKS
915 --          A map from t, where t is a task which has been delayed
916 --          to pairs [d, b], where d is the remaining delay time,
917 --          and b is the body of statements to be executed when
918 --          the delay time interval runs out. The unit of time
919 --          for the delay is one interpreter loop.
920 --
921 --
922 --  ENTERING
923 --          A map which maps the identity of a task which is in
924 --          entering state to the identity of the task which is
925 --          accepting the entry call.
926 --
927 --
928 --  TERMINATED_TASKS
```

```
929 --              A set whose elements are the identities of tasks
930 --              which have been terminated.
931 --
932 --
933 --  COMPLETED_TASKS
934 --                  A set whose elements are the identities of task which
935 --                  are completed, but not yet terminated.
936 --
937 --
938 --  ABNORMAL_TASKS
939 --                  A set whose elements are the identities of tasks which
940 --                  have been aborted during an entry call.
941 --
942 --
943 --  MASTER
944 --              A map from task identities to the identity of the
945 --              master task. The main task is not mapped into any
946 --              master.
947 --
948 --
949 --  TASKENV
950 --              A map from task identities to environment indexes
951 --              which represent the environment in which the task was
952 --              created.
953 --
954 --
955 --  TASK_PRIO
956 --              A map from task identities to priority values.
957 --
958 --
```

DEBUGGING TABLES

```
960 --
961 --  TRACED_TASKS
962 --              A set containing the identities of tasks for
963 --              which statement tracing is active.
964 --
965 --  TRACED_TABLES
966 --
967 --              A map from task identities to tuples of
968 --              table names for which table tracing is active.
969 --
970 --  SYSTEM_CLOCK
971 --
972 --              The total processing time since system start.
973 --
974 --  TASK_CLOCKS
975 --
```

```
976 --              A map from task identities to accumulated
977 --              processing time for the task.
978 --
979 --
980 --
981 --
982 --
983 -- Since this interpreter runs on a single machine, only one
984 -- task actually runs at any one time, identified by the value in
985 -- CURTASK (the identity of the current task). The coding of the
986 -- interpreter essentially makes the execution of any
987 -- single statement uninterruptible. To indicate the regions where
988 -- uninterruptable access to global tasking tables is required, the two
989 -- null macros disable and enable are used, the sense being that
990 -- only one processor would be permitted to access these tables at any
991 -- point within a disable-enable window.
992 --
993   macro disable;
994     pass
995   endm;
996
997   macro enable;
998     pass
999   endm;
1000
```

ADA INTERPRETER PROCEDURE

```
1002 proc INTERP;
1003 --
1004 -- The input consists of three objects. The first is a sequence (tuple) of
1005 -- unit bodies. The second object is the (string) name of the procedure
1006 -- which is the main program. This procedure must have a null parameter
1007 -- list. The third is the priority of the main program if there is one.
1008 -- These objects are in the global variables AIS_CODE, AIS_MAIN, and
1009 -- AIS_PRIO which are created by the binder.
1010 --
1011 --
1012 -- The initial set of environment stacks contains entries for the
1013 -- maintask and a pseudo-task called idle. Since the main program
1014 -- executes at the outer level, there are no stacked environments for it.
1015 -- This is created during compiler initalization by INIT_ENV .
1016 --
1017 --
1018 -- The initial environment for the maintask contains the entries for
1019 -- the standard environment and the environment for Text_io (which is
1020 -- potentially available to all programs). These environments are
1021 -- provided by the routine INIT_ENV during compiler initialization.
1022 -- The entries into this initial environment are inherited by
1023 -- any other tasks which are created by the main task.
1024 --
1025 -- The initial value stack is empty, since entries are only made on this
1026 -- stack to hold temporary values during the course of execution.
1027 --
1028 -- The initial statement sequence consists of the list of units
1029 -- supplied by the binder with a finish statement added, thus
1030 -- forming a valid task body. This task, called MAINTASK, will be
1031 -- initiated. Execution of the finish statement will stop execution,
1032 -- even if there are still some tasks not dependant on the main
1033 -- program that may execute. In any case, execution stops when
1034 -- there can be no other activity but the idle task.
1035 --
1036 --
1037 --
1038
1039  if AIS_MAIN /= '' and AIS_CODE /= [ ] then
1040
1041    STSQT(MAINTASK) := [['block',
1042                         [ AIS_CODE,
1043                           [ ['call', AIS_MAIN, [ ]],
1044                             ['finish_'],
```

```
1045                                ['end']
1046                              ],
1047                              [ ]
1048                            ]    ]];
1049
1050  else                                -- used for interpreter initialization.
1051
1052   STSQT(MAINTASK) := AIS_CODE + [['terminate']];
1053
1054  end if;
1055 --
1056  TASK_PRIO(MAINTASK) := AIS_PRIO(1) ? UNDEF_PRIO;
1057 --
1058  INITIALIZE_PREDEF();
1059 --
1060  INITIALIZE_DEBUG_INFO;
1061 --
1062 --
1063  START_THE_CLOCK;
1064 --
1065 --
1066 -- INTERPRETER LOOP
1067 -- ---------------
1068 --
1069 -- The interpreter loop runs until a finish statement is executed.
1070 -- A finish statement performs termination procedures and returns to
1071 -- the caller of the interpreter. This statement is either
1072 -- one placed in the original statement sequence after a call
1073 -- to a main program, or executed by the idle task when it discovers
1074 -- that there are no ready or delayed tasks.
1075 --
1076  loop do
1077 --
1078 --
1079  -- The following invariant should hold:
1080  --  assert #ACTIVE_TASKS =
1081  -- a task may ready
1082  --   #READY_TASKS +
1083  -- or may have open accepts or be delayed or be terminatable
1084  --   #(domain OPEN_ENTRIES + domain DELAYED_TASKS + TERMINATABLE +
1085  -- or may be waiting on an entry queue
1086  --   { T : exists Q in (range(+/(range WAITING_TASKS))) | T in Q }) +
1087  -- or may be making an accepted entry call
1088  --   #ENTERING +
1089  -- or may be waiting for subtasks to complete their activation
1090  --   #(domain ACTIVATING_SUBTASKS) +
1091  -- or may be waiting for sub-tasks to terminate
1092  --   #HELD_TASKS;
1093 --
```

```
1094 --
1095 -- Called on each cycle thru the interpreter loop to
1096 -- simulate the effect of a hardware clock interrupt.
1097 --
1098   CURRENT_TIME := time;
1099   if CURRENT_TIME >= NEXT_INTERRUPT_TIME then
1100     SERVICE_CLOCK;
1101   end if CURRENT_TIME;
1102 --
1103   STM fromb STSQ; -- we always execute the statement at the head of STSQ
1104 --
1105 --
1106 -- Increment statement counts.
1107 --
1108   if is_ais_stmt then STMTCOUNT +:= 1; end if;
1109   CURRENT_STMT_COUNT +:= 1;
1110 --
1111 if DEBUGGING then DEBUGGING_CHECKS_1; end if;
1112 --
1113   EXECUTE_ONE_STMT;
1114 --
1115 if DEBUGGING then DEBUGGING_CHECKS_2;end if;
1116 --
1117 end loop;
```

INTERPRETER INSTRUCTIONS

```
1119 EXECUTE_ONE_STMT::
1120
1121 -- The instructions recognized by the interpreter are of two kinds:
1122 -- AIS statements and generated statements. The AIS statements are
1123 -- the intermediate code produced by the front end and supplied to
1124 -- the interpreter in AIS_MAIN. The generated statements are produced
1125 -- by the interpreter in the course of executing the AIS statements.
1126 -- They are placed at the head of STSQ and then executed themselves.
1127 -- Thus the interpreter treats STSQ as a stack of intructions. The
1128 -- generated statements may be recognized by the underline ( _ )
1129 -- which always forms the last character of their name.
1130 --
1131 -- The names of the AIS statements are usually the same as that of
1132 -- a non_terminal in the Ada grammar, of which the AIS statement is
1133 -- a translation. Their form is that of a tuple, in which the first
1134 -- entry is the name, and any parameters needed are the subsequent
1135 -- entries. The form of generated statements is the same, and their
1136 -- names are usually formed from the AIS statement of which they are
1137 -- an intermediate stage in execution by adjoining an underline.
1138 -- Some generated statements (e.g. 'oeval_') do not have any AIS
1139 -- counterpart, but are used as part of the execution sequence of
1140 -- certain AIS statements.
1141
1142 -- The statements are divided according to the chapters of the Ada
1143 -- Reference Manual
1144
1145 case opcode(STM) of
```

Internal Utility Instructions

```
1147 -- ['push_', value]
1148 --
1149 --  This is a subsidiary operation used to push a value onto the stack
1150 --  in the middle of an exec sequence. It can be regarded as a deferred
1151 --  push, since VALSTACK is not affected until the push is executed.
1152 --
1153   ('push_'):
1154     [-, value] := STM;
1155     push( value );
1156 --
1157 -- ['pop_', value]
1158 --
1159 --  This is a subsidiary statement used to discard the top of the stack
```

```
1160 --
1161   ('pop_'):
1162     pop ( junk );
1163 --
1164 --  ['repeat_', ntimes, stmt_list]
1165 --
1166 --  This is a subsidiary statement used for multidimensional array
1167 --  creation, to avoid filling the exec stack to quickly.
1168 --  stmt_list is executed ntimes times
1169
1170   ('repeat_'):
1171     [-, ntimes, stmt_list] := STM;
1172     exec( [ ] +/ [ stmt_list : i ∈ [1..ntimes]] ); --
```

Chapter 3. DECLARATIONS AND TYPES

3.2.1 Object declarations

```
1175 -- ['constant', [nam_list], type_name, exprv]
1176 --
1177 --  The constant statement  is used to make  an entry for constants in
1178 --  EMAP, corresponding to a constant declaration in the source.
1179 --  Nam_list is a list of names, and exprv is an expression for their
1180 --  value. The front end ensures that no assignment is made to this
1181 --  object after initialization.
1182 --
1183  ('constant'):
1184    [-, nam_list, type_name, init_expr] := STM;
1185    exec ( [['veval_', init_expr],  -- assume explicit qualification here
1186            ['constant_', nam_list, type_name] ]);
1187 --
1188  ('constant_'):
1189    [-, nam_list, type_name] := STM;
1190    type_disr := GET_ITYPE(type_name);
1191    pop( init_value );
1192    if is_constrained_type(type_disr) then
1193      loop forall name ∈ nam_list do
1194        exec([['create_obj_', type_disr, false, CURTASK, #ENVSTACK,
1195                              name],
1196              ['set_obj_', name, init_value] ]);
1197      end loop forall;
1198    else
1199      loop forall name ∈ nam_list do
1200        EMAP(name) := CREATE_COPY(init_value, type_disr, Ω);
1201      end loop forall;
1202    end if;
1203
1204 -- ['object', nam_list, type_name, init_expr]
1205 --
1206 -- The object statement is used to create an IOBJECT, or set of
1207 -- IOBJECTS given an ITYPE. Name_list is a tuple of names. Type_name
1208 -- designates the ITYPE of the objects to be created.
1209 -- Init_expr is an expression for a possible initial value. If not
1210 -- present, the object may be created with a default init value, if there
1211 -- is one.
1212 --
1213
1214  ('object', 'obj'):
```

```
1215    [-, nam_list, type_name, init_expr] := STM;
1216
1217    if not is_tuple(nam_list) then       -- for compatibility with 11.4
1218        nam_list := [nam_list];
1219    end if;
1220
1221    if present(init_expr) then
1222      exec( [['veval_', init_expr],
1223            ['object_', nam_list, type_name, false]] );
1224    else
1225      exec( [['object_', nam_list, type_name, true ]] );
1226    end if;
1227 --
1228  ('object_'):
1229    [-, nam_list, type_name, default_init] := STM;
1230    type_disr := GET_ITYPE(type_name);
1231    if default_init then
1232      init_value := Ω;
1233    else
1234      pop( init_value );
1235    end if;
1236    loop forall name ∈ nam_list do
1237      exec( [
1238        ['create_obj_', type_disr, default_init, CURTASK, #ENVSTACK,
1239                                name],
1240        ['set_obj_', name, init_value] ]);
1241    end loop forall;
1242
1243 --
1244 -- ['create_obj_', itype_name, default_init, t_master, t_taskenv, name]
1245 --
1246 -- The create_obj_ statement is generated when an object must be
1247 -- created from a type. The parameters are as follows:
1248 --
1249 --   itype_name      the name of the ITYPE giving the type of the object
1250 --
1251 --   default_init    set true if default initializations of record field
1252 --                   values are required.
1253 --
1254 --   t_master        the master task, used for creating task objects.
1255 --                   This is usually, but not always CURTASK.
1256 --
1257 --   t_taskenv       the task environment to be used for creating task
1258 --                   objects. This is usually, but not always, #ENVSTACK.
1259 --
1260 --   obj_name        used in task creation, for debugging purposes
1261 --
1262 -- On completion of execution of the create_obj_ statement, the IOBJECT
1263 -- which is created is left on VALSTACK, with all necessary entries
```

```
1264 -- made in the CONTENTS map.
1265
1266  ('create_obj_'):                              --
1267    [-, itype_name, default_init,
1268       t_master, t_taskenv, obj_name ] := STM;
1269    CREATE_OBJ( itype_name, default_init,
1270              t_master, t_taskenv, obj_name );
1271
1272 -- ['create_obj_r1_', objdec_list, disc_list, constr_disc_list]
1273 --
1274 -- This is the subsidiary statement used to create a record iobject
1275 -- from a sequence of component iobjects stored on VALSTACK. It is
1276 -- used in the case where default initializations are not performed.
1277 --
1278   ('create_obj_r1_'):
1279     [-, objdec_list, disc_list, constr_disc_list] := STM;
1280
1281  -- Strip the created component objects off the stack
1282
1283     obj := VALSTACK (#VALSTACK - #objdec_list + 1 ..);
1284     VALSTACK (#VALSTACK - #objdec_list + 1 ..) := [ ];
1285
1286  -- Build the object list for the resulting record object
1287
1288     objlist := {};
1289     loop for i ∈ [1 .. #obj] do
1290       [field_name, [field_itype, field_vexpr]] := objdec_list(i);
1291       field_iobject := obj(i);
1292       objlist(field_name) := field_iobject;
1293
1294  -- If the record is constrained, the values of the discriminant
1295  -- are ready in the field_vexpr (although we are not initializing)
1296 --
1297       if present(constr_disc_list) and
1298                              field_name ∈ constr_disc_list then
1299         [-, discr_value] := field_vexpr;
1300         CONTENTS(field_iobject) := discr_value;
1301       end if;
1302     end loop for;
1303
1304  -- If discriminants are present and the record is not constrained,
1305  -- then add a dummy field indicating no fields are actually present
1306  -- (remember that this is the case where we are not initializing)
1307
1308     if nonempty(disc_list) and absent(constr_disc_list) then
1309       special_object := newat;
1310       CONTENTS(special_object) := {};
1311       objlist('fields_present') := special_object;
1312     end if;
```

```
1313
1314  -- Return the resulting object and pop the environment stack if we
1315  -- have discriminants to remove the dummy scope for variant values.
1316
1317     if nonempty(disc_list) then --
1318        save_TD := TASKS_DECLARED;
1319        POP_ENVSTACK;
1320        TASKS_DECLARED +:= save_TD;
1321     end if;          --
1322     push(['record_iobject', objlist, constr_disc_list]);
1323
1324 -- ['create_obj_r2_', numobj]
1325 --
1326 -- This is a subsidiary statement used in the creation of records to
1327 -- peform default initializations. The top of the stack contains the
1328 -- sequence of objects, followed by the sequence of default values.
1329
1330   ('create_obj_r2_'):
1331     [-, numobj] := STM;
1332     default_values := VALSTACK(#VALSTACK - numobj + 1 ..);
1333     VALSTACK(#VALSTACK - numobj + 1 ..) := [ ];
1334     obj := VALSTACK(#VALSTACK - numobj + 1..);    --
1335     loop for i ∈ [1..numobj] | default_values(i) /= UNINITIALIZED do
1336 --
1337       SETVAL(obj(i), default_values(i), Ω);
1338     end loop for;
1339
1340 -- ['create_obj_r3_', field_name, field_vexpr]
1341 -- ['create_obj_r4_', field_name]
1342 --
1343 -- These two statement forms are used to initialize the value of a
1344 -- discriminant in EMAP when building a record with discriminants.
1345
1346   ('create_obj_r3_'):
1347     [-, field_name, field_vexpr] := STM;
1348     exec( [['veval_', field_vexpr],
1349           ['create_obj_r4_', field_name]]);
1350
1351   ('create_obj_r4_'):
1352     [-, field_name] := STM;
1353     pop( expr_value );
1354     EMAP(field_name) := expr_value;
1355
1356 -- ['create_obj_r5_', objdec_list, variant_part, disc_list]
1357 --
1358 -- This is a subsidiary statement used in building a record with
1359 -- discriminants and default initializations. At this stage, all
1360 -- the entries for the discriminant values are made in EMAP.
1361
```

```
1362  ('create_obj_r5_'):
1363    [-, objdec_list, variant_part, disc_list, obj_name] := STM;
1364
1365  -- What we want to do now is to build two lists, one of fields which
1366  -- are to be initialized (init_list) and the other of fields which
1367  -- are not to be initialized (no_initlist) since they represent
1368  -- fields in variants not selected by the initial discriminant values.
1369
1370    init_list := objdec_list;
1371    noinit_list := [ ];
1372
1373    loop while variant_part /= [ ] do
1374      [variant_name, altern_list] := variant_part;
1375      variant_value := EMAP(variant_name);
1376
1377  -- Find the variant which is to be used and withdraw it from
1378  -- the altern_list
1379
1380      if
1381        (exists component_list = altern_list (choices) |
1382          CONTAINS ( choices, variant_value))
1383      or
1384        present(component_list := altern_list(choices := {'others'}))
1385      then
1386        altern_list ( choices ) := Ω;
1387      else
1388        SYSTEM_ERROR('create_obj_r5_');
1389      end if;
1390
1391  -- The variant which is used does get initialized
1392
1393      [objdec_list, variant_part] := component_list;
1394      init_list +:= objdec_list;
1395
1396  -- All other variants do not get initialized
1397
1398      loop for [choices, component_list] ∈ altern_list do
1399        noinit_list +:= FLATTEN_RECORD (component_list);
1400      end loop for;
1401    end loop while;
1402
1403  -- At this stage we have all the fields which need initializing, so
1404  -- we first create the objects which are to be initialized, then
1405  -- evaluate the corresponding default initializations, then we
1406  -- perform the initializations, then we create all the other objects,
1407  -- and finally we can assemble the result.
1408
1409    exec (
1410      [['create_obj_', field_type, is_uninitialized(field_vexpr),
```

```
1411                    t_master, t_taskenv, obj_name]      --
1412          : [field_name, [field_type, field_vexpr]] ∈ init_list] +
1413      [['veval_', field_vexpr]
1414          : [field_name, [field_type, field_vexpr]] ∈ init_list
1415            | field_name ∉ disc_list] +
1416      [['create_obj_r2_', #init_list - #disc_list]] +
1417      [['create_obj_', field_type, false, t_master, t_taskenv, obj_name]
1418          : [field_name, [field_type, field_vexpr]] ∈ noinit_list] +
1419      [['create_obj_r6_', init_list, noinit_list, disc_list]] );
1420
1421 -- ['create_obj_r6_', init_list, noinit_list, disc_list]
1422 --
1423 -- This is a subsidiary statement used to finalize the construction
1424 -- of a record iobject with discriminants where default initialization
1425 -- is performed. At this stage, all initialization is complete, except
1426 -- that the objects corresponding to discriminant values have not been
1427 -- initialized.
1428
1429  ('create_obj_r6_'):
1430    [-, init_list, noinit_list, disc_list] := STM;
1431
1432  -- First get all the created component objects off VALSTACK
1433
1434    noinit_obj := VALSTACK(#VALSTACK - #noinit_list + 1 ..);
1435    VALSTACK(#VALSTACK - #noinit_list + 1 ..) := [ ];
1436    init_obj := VALSTACK(#VALSTACK - #init_list + 1 ..);
1437    VALSTACK(#VALSTACK - #init_list + 1 ..) := [ ];
1438
1439  -- Perform initializations of all the discriminant objects
1440  -- using the values which create_obj_r4_ saved in EMAP earlier on.
1441 --
1442    disc_list := [ field_name : [field_name, -] ∈ init_list
1443                  | field_name ∈ disc_list];
1444    loop for i ∈ [1 .. #disc_list] do
1445      SETVAL(init_obj(i), EMAP(disc_list(i)), Ω);
1446    end loop for;
1447
1448  -- Now build the resulting objdeclist for the iobject
1449
1450    objdeclist := {};
1451
1452    loop for i ∈ [1 .. #init_list] do
1453      [field_name, [field_itype, field_vexpr]] := init_list(i);
1454      objdeclist(field_name) := init_obj(i);
1455    end loop for;
1456
1457    loop for i ∈ [1 .. #noinit_list] do
1458      [field_name, [field_itype, field_vexpr]] := noinit_list(i);
1459      objdeclist(field_name) := noinit_obj(i);
```

```
1460    end loop for;
1461
1462  -- If the record is constrained, we just build the record iobject
1463  -- indicating the appropriate list of constrained discriminants.
1464  -- Return the resulting object and pop the environment stack, as we
1465  -- have discriminants, to remove the dummy scope for variant values.
1466
1467    constr_disc_list := EMAP('constr_disc_list');
1468    save_TD := TASKS_DECLARED;  --
1469    POP_ENVSTACK;               --
1470    TASKS_DECLARED +:= save_TD;
1471    if present(constr_disc_list) then
1472      push( ['record_iobject', objdeclist, constr_disc_list] );
1473
1474  -- Otherwise we build a dynamic record iobject, noting that the set
1475  -- of fields which we initialized corresponds to the set of fields
1476  -- which are to be marked present in the 'fields_present' entry.
1477
1478    else
1479      loc := newat;
1480      objdeclist('fields_present') := loc;
1481      CONTENTS(loc) :=
1482        {field_name : [field_name, [field_itype, field_vexpr]]
1483                      ∈ init_list};
1484      push( ['record_iobject', objdeclist] );
1485    end if;
1486
1487 -- ['create_obj_a_', lowobj, highobj, lowdiscr, highdiscr, numelmts]
1488 --
1489 -- This is a subsidiary statement used by create_obj_ to actually
1490 -- create an array object after all the component objects have been
1491 -- created and placed on the value stack in sequence. All that it
1492 -- does is to replace the sequence of created iobjects for the elements
1493 -- on VALSTACK with a single entry which is the iobject for the array.
1494
1495  ('create_obj_a_'):
1496    [-, lowobj, highobj, lowdiscr, highdiscr, numelmts] := STM;
1497    if numelmts = 0 then
1498      push(['array_iobject', [ ], lowobj, highobj, lowdiscr,highdiscr]);
1499    else
1500    seq := VALSTACK (#VALSTACK - numelmts + 1 ..);
1501    VALSTACK (#VALSTACK - numelmts + 1 ..) :=
1502      [['array_iobject', seq, lowobj, highobj, lowdiscr, highdiscr]];
1503    end if;
1504
1505 -- ['set_obj_', name, init_value]
1506 --
1507 -- This statement is typically generated following a 'create_obj_'
1508 -- statement at the outer level to associate the created object
```

```
1509 -- which is on the top of VALSTACK with the given name. If the
1510 -- init_value parameter is present, then it is a value which is
1511 -- to be assigned to the newly created object
1512 --
1513  ('set_obj_'):
1514    [-, name, init_value] := STM;
1515    pop( new_obj );
1516    EMAP (name) := new_obj;
1517    if present(init_value) then
1518      SETVAL (new_obj, init_value, Ω);
1519    end if;
```

3.3.1 Type declarations

```
1521 -- ['type', type_name, type_expr]
1522 --
1523 -- A type statement makes an EMAP entry for a type.
1524 -- Type_expr is a texpr which evaluates to the ITYPE
1525 -- to be bound to type_name.
1526 --
1527  ('type'):
1528    [-, type_name, type_expr] := STM;
1529    exec( [['teval_', type_expr],
1530           ['type_', type_name] ] );
1531
1532  ('type_'):
1533    [-, type_name] := STM;
1534    pop( type_val );
1535 --
1536 -- Executing delayed statements:
1537    [delay_flag, delayed_STM] := EMAP(type_name) ? [ ];
1538    if delay_flag = 'delayed' then
1539      exec(delayed_STM ? [ ]); --
1540    end if;
1541 --
1542 -- Setting type value in environement:
1543    EMAP(type_name) := type_val;
1544 --
1545 -- ['delayed_type', type_name, type_expr, parent_type ]
1546 --
1547 -- This is a type whose elaboration has to be delayed until some
1548 -- other 'parent_type' is elaborated, usually a private type, or
1549 -- another delayed type
1550 --
1551  ('delayed_type'):
1552    [-, type_name, type_expr, parent_type ] := STM;
1553    [-, delayed_STM] := EMAP(parent_type) ? [[ ], [ ]];
1554    EMAP(parent_type) := ['delayed', delayed_STM with
1555          ['type', type_name, type_expr ] ];
```

```
1556    EMAP(type_name) := ['delayed', [ ]]; --
1557
1558 --
1559 -- ['teval_', type_expr]
1560 --
1561 -- The 'teval_' instruction is used to evaluate TEXPR's to produce the
1562 -- ITYPE for the corresponding type value. TEXPR's appear as the operand
1563 -- of the TYPE statement, which places an ITYPE value in EMAP, and
1564 -- also in general expression contexts (e.g. the range for the in
1565 -- operator).
1566 --
1567 --
1568 --
1569 --
1570  ('teval_'):
1571    [-, type_expr] := STM;
1572 --
1573 -- Named ITYPE
1574 --
1575 --  string_name
1576 --
1577 --  In any context where an ITYPE is expected to appear, the use
1578 --  of a string name refers to a domain element of EMAP, whose
1579 --  current corresponding range element is the referenced ITYPE.
1580 --  This is required for handling mutually recursive types, and
1581 --  is permitted in any context. The GET_ITYPE procedure can be used
1582 --  to effect the necessary dereferencing to obtain the actual
1583 --  value of a named ITYPE.
1584 --
1585    if is_simple_name(type_expr) then
1586      push( GET_ITYPE(type_expr) );
1587    else
1588 --
1589 -- Most type evaluation processing is done in procedure TEVAL_PROC
1590 --
1591      TEVAL_PROC(type_expr);
1592    end if;
```

Chapter 4. NAMES AND EXPRESSIONS

4.1 Names

```
1595 -- ['oeval_', eval_action]
1596 --
1597 -- The 'oeval_' instruction is used to evaluate OEXPR's to
1598 -- produce the resulting IOBJECT. These instructions are
1599 -- never present explicitly in the generated intermediate
1600 -- code, but are generated dynamically as an expression
1601 -- is evaluated. The single parameter is the next action
1602 -- to be performed. General possibilities are subexpressions
1603 -- to be evaluated and pushed onto VALSTACK, or actions which
1604 -- process VALSTACK entries.
1605
1606 -- Essentially the steps taken by this procedure correspond
1607 -- to the steps in a conventional code generator. Only at
1608 -- the bottom level are actions performed which correspond
1609 -- to normal run time actions.
1610
1611 -- Control is initially received when an OEXPR is encountered
1612 -- which must be evaluated, and the initial step is to generate
1613 -- an 'oeval_' instruction with the expression as its parameter.
1614
1615   ('oeval_'):
1616     [-, eval_action] := STM;
1617     OEVAL_PROC(eval_action);
```

4.4 Expressions

```
1619 -- ['veval_', eval_expr]
1620 --
1621 -- The 'veval_' instruction is used to evaluate VEXPR'S to
1622 -- produce the resulting IVALUE. These instructions are never
1623 -- present explictly in the intermediate code, but are generated
1624 -- dynamically as an expression is evaluated. The single parameter
1625 -- is the next action to be performed.
1626 --
1627   ('veval_'):
1628     [-, eval_expr] := STM;
1629 --
1630     if is_literal_ivalue(eval_expr) then
1631 --
1632 -- A convenience feature allows literal integer and enumeration type
```

```
1633 -- IVALUE's (which are simply SETL integers) to be represented as
1634 -- themselves in VEXPR's (since an integer value would otherwise be an
1635 -- invalid VEXPR, no confusion arises). Thus 4 and ['ivalue', 4] are
1636 -- equivalent in a VEXPR context. Similarly float IVALUE's can
1637 -- also appear for themselves, but any more complicated values must
1638 -- be quoted using 'ivalue'.
1639 --
1640       push( eval_expr );
1641 --
1642     elseif is_simple_name(eval_expr) then
1643 --
1644 -- Named Object
1645 --
1646       object := GLOBAL_EMAP(eval_expr) ? EMAP(eval_expr);
1647       if absent(object) then
1648         exec([['raise', 'PROGRAM_ERROR',--
1649           'Access to ' + str eval_expr + ' before elaboration']]);
1650       else
1651         value := L_VALUE(object);
1652         if value = UNINITIALIZED
1653           or (is_tuple value and value(2) = UNINITIALIZED) then
1654           exec([['raise', 'PROGRAM_ERROR', 'uninitialized object']]);
1655         else
1656           push( value );
1657         end if value;
1658       end if absent(object);
1659 --
1660     elseif not is_operation(eval_expr) then
1661 --
1662       SYSTEM_ERROR('ill-formed expression ' + str eval_expr);
1663 --
1664     else
1665 --
1666 -- All other VEXPR's are evaluated by VEVAL_PROC (q.v.)
1667 --
1668       VEVAL_PROC(eval_expr);
1669
1670     end if is_literal_ivalue(eval_expr);
1671 --
1672 --
1673 -- ['opbinary_', opname]
1674 --
1675 -- The opbinary_ operation is generated as part of the sequence used to
1676 -- evaluate expressions. The two operands have already been evaluated
1677 -- and dereferenced and are on the top of VALSTACK. Opname is the name of
1678 -- the operation to be performed.
1679 --
1680   ('opbinary_'):
1681     [-, opname] := STM;
```

```
1682    OPBINARY(opname);
1683 --
1684 -- ['opunary_', opname]
1685 --
1686 -- The opunary_ operation is used to evaluate unary operators in a
1687 -- similar manner to opbinary_.
1688 --
1689  ('opunary_'):
1690    [-, opname] := STM;
1691    OPUNARY(opname);
```

Chapter 5. STATEMENTS

```
1693 -- ['null']
1694 --
1695 -- Acts as a no_op.
1696 --
1697   ('null'):
1698     pass;
```

5.1 Labels

```
1700 --       ['labdef', label_name]
1701 --
1702 --       In ADA itself, no label declarations are required, a label is
1703 --       declared implicitly by its occurence on a statement. In the
1704 --       language processed by the interpreter, label declarations are
1705 --       required, and have the form shown above.
1706 --
1707 --       Labdef captures the current environment (which is the one to be
1708 --       restored  when the goto occurs) by remembering the current
1709 --       environment nesting depth  (i.e. the length of ENVSTACK) and
1710 --       also captures the statement sequence to be executed when the
1711 --       goto is executed. Both of these values are stored in the EMAP
1712 --       entry for the label.
1713 --
1714         ('labdef'):
1715           [-, lname] := STM;
1716           if exists i ∈ [1..#STSQ] | STSQ(i) = ['label', lname] then
1717             EMAP(lname) := CREATE_COPY(
1718                 ['label_ivalue', #ENVSTACK, STSQ(i + 1...)], Ω, Ω);
1719           else
1720             SYSTEM_ERROR(' missing label statement for ' + str lname);
1721           end if;
1722
1723 -- ['label', lname]
1724 --
1725 -- The label statement is a dummy statement which has no effect when
1726 -- executed. It is a declaration of the form:
1727 --     ['labdef', lname]
1728 -- which actually defines a label. The label statement serves to mark the
1729 -- position of the label whose name is lname for the labdef declaration.
1730 --
1731   ('label'):
1732     pass;
```

5.2 Assignment

```
1734 -- ['':=', name, exprv]
1735 --
1736 -- This statement represents an  assignment . Name is an oexpr for the
1737 -- left hand side and exprv is a vexpr for the right hand side.
1738 --
1739   (':='):
1740     [-, name, exprv] := STM;
1741     exec( [['oeval_', name],
1742            ['veval_', exprv],
1743            ['assign_']        ]);
1744
1745   ('assign_'):
1746     pop( rhs_ );
1747     pop( lhs_ );
1748     SETVAL(lhs_, rhs_, Ω);
1749
1750 --
1751 -- Special assignment for slices: just for cases like:
1752 --       T(U..V) := (others => X);
1753 -- in order to avoid having the range evaluated twice.
1754 --
1755    ('[..]:='):
1756     [-, type_name, type_expr, obj_name, v_expr] := STM;
1757     exec( [[':=', ['[..]', obj_name, type_name], v_expr]] );
1758     if is_tuple(type_expr) then
1759       exec([['type', type_name, type_expr]]);
1760     end if;
1761
```

5.3 If statement

```
1763 -- ['if', cslist, elsebody]
1764 --
1765 -- The if instruction contains a tuple of condition-body pairs in cslist,
1766 -- of the form [cond, body], where cond is a boolean valued vexpr, and
1767 -- body is the sequence of statements to be executed if that condition is
1768 -- true. If an else part is present, then elsebody is the sequence for
1769 -- the else, otherwise elsebody is omitted.
1770 --
1771 -- If evaluates each condition in turn and places the body corrisponding
1772 -- to it at the head of the statement sequence if it is true. Note that
1773 -- no condition is evaluated unless the preceeding ones are false (as
1774 -- required by RM 5.3 para. 2). If none of the conditions are true, and
1775 -- there is an elsebody, then it is executed.
1776 --
1777   ('if'):
```

```
1778    [-, cslist, elsebody] := STM;
1779    if cslist /= [ ] then
1780      [cond, body] fromb cslist;
1781      exec( [['veval_', cond],
1782            ['if_', body, [['if', cslist, elsebody]] ]] );
1783    elseif present(elsebody) then
1784      exec(elsebody);
1785    end if;
1786 --
1787 -- ['if_', truebody, falsebody]
1788 --
1789 -- The if_ operation is used as part of the evaluation sequence for all
1790 -- operations which conditionally modify the flow of control. It tests
1791 -- the top value on VALSTACK (which must be either true or false) and
1792 -- executes the appropriate body depending on which value is stacked.
1793 --
1794  ('if_'):
1795    [-, truebody, falsebody] := STM;
1796    pop( value );
1797    if boolean_true = value then
1798      exec (truebody);
1799    else
1800      exec (falsebody);
1801    end if;
```

5.4 Case statement

```
1803 -- ['case', exprv, alist, othersblock]
1804 --
1805 -- The alternatives of a case statement are represented as pairs,
1806 -- [alts, body] where alts is a set consisting of discrete values, or
1807 -- ranges of values (represented as ['range', lbd, ubd]) for the branch,
1808 -- and body is the corresponding statement sequence. Exprv is the
1809 -- expression whose value determines which alternative will be executed.
1810 -- Othersblock is the statement sequence that will be executed if none
1811 -- of the alternatives match exprv.
1812 --
1813  ('case'):
1814    [-, exprv, alist, othersblock] := STM;
1815    exec ( [['veval_', exprv],
1816          ['case_', alist, othersblock]] );
1817
1818  ('case_'):
1819    [-, alist, othersblock] := STM;
1820    pop( value );
1821    if exists [alts, body] ∈ alist | CONTAINS(alts, value) then
1822      exec(body);
1823    else
1824      exec(othersblock);
```

```
1825    end if;
```

5.5 Loop statement

```
1827 -- ['loop', body]
1828 -- ['loop', body, label_name]
1829 --
1830 -- The loop instruction works by duplicating the body of the loop ahead
1831 -- of itself each time the loop is executed. Note that any loop which is
1832 -- the object of an exit instruction must be labeled.
1833 --
1834   ('loop'):
1835     [-, body, label_name] := STM;
1836     if present(label_name) then
1837       EMAP(label_name) := CREATE_COPY(
1838               ['label_ivalue', #ENVSTACK, STSQ], Ω, Ω);
1839     end if;
1840     exec(body with STM);
1841
1842 -- ['while', whilecond, body]
1843 -- ['while', whilecond, body, label_name]
1844 --
1845 -- The while statement represents a while loop and is similar to the
1846 -- loop statement except that a condition is present. Whilecond is a
1847 -- boolean vexpr, and body is a sequence of statements which is to be
1848 -- executed as long as the condition is true.
1849 --
1850   ('while'):
1851     [-, whilecond, body, label_name] := STM;
1852     if present(label_name) then
1853       EMAP(label_name) := CREATE_COPY(
1854               ['label_ivalue', #ENVSTACK, STSQ], Ω, Ω);
1855     end if;
1856     exec( [['veval_', whilecond],
1857          ['if_', body with STM, [ ] ]] );
1858
1859 -- ['for', var_name, range_name, range_def, stmt_list, label_name]
1860 -- ['forrev', var_name, range_name, range_def, stmt_list, label_name]
1861 --
1862 -- These statements implement the for statement by converting it into a
1863 -- loop. An object is created for the control variable, but a new
1864 -- scope is not created. The front end ensures that the control variable
1865 -- name is unique within the current scope, and no task creations or
1866 -- exception handlers depend on the loop, so a new scope is not
1867 -- necessary. var_name is the name of the control variable, range_def
1868 -- is a texpr for the (sub)type of the loop's range, and stmt_list is the
1869 -- statement sequence of the loop. Label_name is present in the case of
1870 -- an exit statement in the loop. The 'for' statement is used for a loop
1871 -- in increasing order, and 'forrev' for decreasing order.
```

```
1872 --
1873  ('for', 'forrev'):
1874    [for_kind, var_name, range_name, range_def, stmt_list, label_name]
1875                                                  := STM;
1876    if present(label_name) then
1877       EMAP(label_name) := CREATE_COPY(
1878             ['label_ivalue', #ENVSTACK, STSQ], Ω, Ω);
1879    end if;
1880    exec( [[ for_kind + '_', var_name, range_name, stmt_list]] );
1881 --
1882    if (range_def ? [ ]) /= [ ] then
1883      exec([['type', range_name, range_def]]);
1884    end if;
1885
1886 --
1887  ('for_'):
1888    [-, var_name, range_name, stmt_list] := STM;
1889    range_desc := GET_ITYPE( range_name );
1890    get_bounds_of(range_desc);
1891    if ubd >= lbd then
1892      exec( [['object', [var_name], range_name, lbd],
1893           ['loop', stmt_list + [['for_end_', var_name, ubd ]] ]
1894          ]   );
1895    end if;
1896 --
1897 ('for_end_'):
1898    [-, var_name, ubd] := STM;
1899    var_value := CONTENTS(EMAP(var_name));
1900    if var_value >= ubd then
1901      STSQ := STSQ(2..);
1902    else
1903      CONTENTS(EMAP(var_name)) := var_value + 1;
1904    end if;
1905 --
1906  ('forrev_'):
1907    [-, var_name, range_name, stmt_list] := STM;
1908    range_desc := GET_ITYPE( range_name );
1909    get_bounds_of(range_desc);
1910    if ubd >= lbd then
1911      exec( [['object', [var_name], range_name, ubd],
1912           ['loop', stmt_list + [['forrev_end_', var_name, lbd ]] ]
1913          ]   );
1914    end if;
1915 --
1916 ('forrev_end_'):
1917    [-, var_name, lbd] := STM;
1918    var_value := CONTENTS(EMAP(var_name));
1919    if var_value <= lbd then
1920      STSQ := STSQ(2..);
```

```
1921    else
1922      CONTENTS(EMAP(var_name)) := var_value - 1;
1923    end if;
```

5.6 Block statement

```
1925 -- ['block', body]
1926 --
1927 --  The block instruction is used to represent a declare, begin, end
1928 --  sequence in the source program. The body is terminated by an end
1929 --  instruction.
1930 --
1931   ('block'):
1932     [-, body] := STM;
1933     [decls, stmts, hndlr] := body;
1934     PUSH_ENVSTACK;
1935     STSQ := decls + hndlr + [['activate_']] + stmts; -- RM 9.3
1936
1937 -- ['end']
1938 --
1939 --  The end instruction is used at the end of a block body to restore
1940 --  the outer environment on terminating the block.
1941 --
1942   ('end'):
1943     exec( [['end_']] );
1944     WAIT_SUBTASKS;
1945
1946   ('end_'):
1947     POP_ENVSTACK;
```

5.7 Exit statement

```
1949 -- ['exit', loop_name ]
1950 -- ['exit', loop_name, when_cond]
1951 --
1952 -- An exit instruction is used to exit a loop. It is just like a goto
1953 -- except that the statement following the label (the loop statement)
1954 -- is skipped before proceeding with execution. loop_name is the name of
1955 -- a label object, and when_cond (if present) is a vexpr which evaluates
1956 -- to a boolean value, which determines if the exit is to be performed.
1957 --
1958   ('exit'):
1959     [-, loop_name, when_cond] := STM;
1960
1961     if present(when_cond) then
1962       exec( [['veval_', when_cond],
1963             ['if_', [['veval_', loop_name], ['exit_']], [ ] ]] );
1964     else
1965       exec( [['veval_', loop_name],
```

```
1966            ['exit_'] ]);
1967      end if;
1968  --
1969    ('exit_'):
1970      pop( label_value );
1971      [-, envindex, new_stsq]  := label_value;
1972      if #ENVSTACK > envindex then
1973        exec( [['exit__', label_value]] );
1974        WAIT_SUBTASKS;
1975      else
1976        STSQ := new_stsq;
1977      end if;
1978  --
1979    ('exit__'):
1980      [-, label_value] := STM;
1981      POP_ENVSTACK;
1982      push( label_value );
1983      exec( [['exit_']] );
```

5.8 Return statement

```
1985 -- ['return', procname, nesting_depth, rexpr]
1986 --
1987 -- The return statement  uses  VALSTACK  to rewrite the values of out and
1988 -- inout parameters. It also deals with the  returned value of a function
1989 -- or value returning procedure. Procname is the name of the procedure
1990 -- being returned from. Nesting_depth is the static number of blocks in
1991 -- which the return is embedded. Rexpr is an expression for the return
1992 -- value (if any).
1993
1994    ('return'):
1995      [-, procname, nesting_depth, rexpr] := STM;
1996      if absent(rexpr) then
1997        exec( [['oeval_', procname],
1998              ['return_', nesting_depth, false]] );
1999      else
2000        exec( [['oeval_', procname],
2001              ['veval_', rexpr],
2002                ['return_', nesting_depth, true]]);
2003      end if;
2004      WAIT_SUBTASKS;
2005
2006    ('return_'):
2007      [-, nesting_depth, valflg] := STM;
2008      if valflg then
2009        pop( rval );
2010      else
2011        rval := Ω;
2012      end;
```

```
2013    pop( proc_obj );
2014    if nesting_depth /= 0 then  -- return from inner block of procedure
2015      exec( [['return__', proc_obj, nesting_depth, rval]] );
2016    else                        -- return from outermost scope of procedure
2017      [-, -, -, formals_loc, -] := proc_obj;
2018      POP_ENVSTACK;
2019      BIND_EXIT(L_VALUE(formals_loc), VALSTACK);
2020      if valflg then push( rval ); end;
2021    end if;
2022 --
2023  ('return__'): -- pops stack to get to outermost scope of procedure
2024    [-, proc_obj, nesting_depth, rval] := STM;
2025    POP_ENVSTACK;                          --
2026    push( proc_obj );
2027    if present(rval) then
2028      push( rval );
2029      valflg := true;
2030    else
2031      valflg := false;
2032    end if;
2033    exec( [['return_', nesting_depth - 1, valflg]] ); --
2034
2035
2036 -- ['entry_return', entry_name, nesting_depth]
2037 --
2038 -- entry return is similar to a return, but used within an accept
2039 -- statement
2040 --
2041  ('entry_return'):
2042    [-, entry_name, nesting_depth] := STM;
2043      exec( [['oeval_', entry_name],
2044             ['entry_return_', nesting_depth]] );
2045
2046  ('entry_return_'):
2047    [-, nesting_depth] := STM;
2048    pop( entry_obj );
2049    if nesting_depth /= 0 then  -- return from inner block of accept
2050      exec( [['entry_return__', entry_obj, nesting_depth]] );
2051      WAIT_SUBTASKS;
2052    else
2053      STSQ := STSQ(#STSQ..); -- last statement is 'endrv_'  -
2054    end if;
2055 --
2056  ('entry_return__'): -- pops stack to get to outermost scope of accept
2057    [-, entry_obj, nesting_depth] := STM;
2058    POP_ENVSTACK;
2059    push( entry_obj );
2060    exec( [['entry_return_', nesting_depth - 1]] );
```

5.9 Goto statement

```
2062 -- ['goto', lname]
2063 --
2064 -- Goto works by  restoring the  environment to that  of the  label, and
2065 -- then  setting  the statement  sequence  to execute  the  appropriate
2066 -- statements starting  with  the one  which is  labelled. The IVALUE of
2067 -- the label contains  the information for  both steps. This entry is
2068 -- established  by  the labdef  statement. which should be
2069 -- examined for further details of how gotos are handled. Lname is the
2070 -- name of the label IOBJECT.
2071 --
2072   ('goto'):
2073     [-, lname] := STM;
2074     exec( [['veval_', lname],
2075           ['goto_']] );
2076 --
2077   ('goto_'):
2078     pop( label_value );
2079     [-, envindex, new_stsq]  := label_value;
2080     if #ENVSTACK > envindex then
2081       exec( [['goto__', label_value]] );
2082       WAIT_SUBTASKS;
2083     else
2084       STSQ := new_stsq;
2085     end if;
2086 --
2087   ('goto__'):
2088     [-, label_value] := STM;
2089     POP_ENVSTACK;
2090     push( label_value );
2091     exec( [['goto_']] );
```

Chapter 6. SUBPROGRAMS

6.1 Subprogram declaration

```
2094 -- ['procedure', name, formals, body]
2095 -- ['function', name, formals, body]
2096 --
2097 --      This instruction creates an IVALUE, then an IOBJECT, for a
2098 --      procedure or a function. In the case
2099 --      of a procedure whose body has been created using the predef
2100 --      pragma, this procedure is really a name for a predefined
2101 --      operation and an entry starting ['$predef_op', .... is returned
2102 --      instead of a procedure IVALUE.
2103 --
2104 --      Formals is a list of triples:
2105 --
2106 --        [formalname, pmode, itype]
2107 --
2108 --      where formalname is the name of the formal, pmode is 'in',
2109 --      'inout', or 'out', and itype is the type of the formal. Body is
2110 --      the declarations, statement list, and exception handler of the
2111 --      procedure.
2112 --
2113 --      If the body is absent, this represents just a procedure or
2114 --      function specification, and is not processed by the interpreter.
2115 --
2116     ('procedure', 'function'):
2117       [-, name, formals, body] := STM;
2118       if present(body) then
2119         [decls, stmts, hndlr] := body;
2120         if opcode(stmts(1)) = ('predef_') then
2121
2122 -- This is an optimization that recognizes non_generic predefined
2123 -- procedures during compiler initialization (i.e. when INIT_ENV
2124 -- is called) and instantiations of generic predefined procedures
2125 -- during normal execution, and loads their definitions
2126 -- into EMAP with the tag '$predef_op' . The 'call' operation takes
2127 -- advantage of this information to avoid an unnecessary scope change.
2128
2129           [-, op_name, actuals] := stmts(1);
2130           if #actuals > #formals then -- type parameters
2131             loop for i ∈ [#formals + 1..#actuals] do
2132               formals with:= [actuals(i), 'in', '$type'];
2133             end loop for;
```

```
2134          end if;
2135          EMAP(name) := ['$predef_op', op_name, formals];
2136
2137        else -- normal procedure
2138
2139          proc_body := decls + hndlr + [['activate_']] + stmts;
2140          EMAP(name) := CREATE_COPY (
2141         ['proc_ivalue', CURTASK, #ENVSTACK, formals, proc_body], Ω,
2142                                        Ω);
2143
2144        end if;
2145      end if;
```

6.4 Subprogram call

```
2147 -- ['call', procname, actuals]
2148 --
2149 --  The call instruction is used for procedure call statements in
2150 --  the source program, and is also generated during the
2151 --  evaluation of expressions containing function calls. In the
2152 --  latter cases, the called function will stack the result on
2153 --  VALSTACK. A call statement may also be generated for a predefined
2154 --  operator in a generic, an attribute that has been renamed, or a
2155 --  call to predef. In these cases, the call generates an 'veval_'
2156 --  statement or a 'predef_' statement. They can be distingushed from
2157 --  an ordinary procedure call because their entry in EMAP is a tuple
2158 --  whose first component is '$operator', '"', or '$predef_op'.
2159 --
2160  ('call'):
2161    [-, procname, actuals] := STM;
2162    exec( [['oeval_', procname],
2163           ['call_', actuals, procname] ]);
2164 --
2165  ('call_'):
2166    [-, actuals, procname] := STM;
2167    pop( object );
2168
2169    if is_proc_iobject(object) then -- normal procedure call
2170      [-, -, -, formals_loc, -] := object;
2171      PARAM_EVAL(actuals, L_VALUE(formals_loc),
2172        ['call__', object, procname] );
2173
2174    elseif type_mark(object) = '$predef_op' then -- predefined operation
2175      [-, op_name, formals] := object;
2176      exec( [['predef_', op_name, actuals, formals] ]);
2177
2178    elseif type_mark(object) = '$operator' then
2179                        -- renamed predefined operator
2180      [-, op_name] := object;
```

```
2181      exec( [['veval_', [op_name] + actuals ]]);
2182
2183    elseif type_mark(object) = "" then     -- renamed attribute
2184      exec( [['veval_', object + actuals]] );
2185
2186    else
2187      SYSTEM_ERROR('call');
2188    end if;
2189 --
2190   ('call__'):
2191      [-, object, procname] := STM;
2192      [-, task, index, formals, body] := L_VALUE(object);
2193      old_valstack := VALSTACK;
2194      PUSH_ENVSTACK;
2195      EMAP := ENVSTACKT(task)(index)(1);
2196      STSQ := body;
2197      PROC_NAME := procname; -- for debugging purposes
2198      BIND_ENTER(formals, EMAP, old_valstack);
2199
2200 --
2201 -- ['predef_', operation, actuals, formals]
2202 --
2203 -- A 'predef_' statement is generated whenever a  call is made to
2204 -- one of the (predefined)  procedures  in  the  packages TEXT_IO,
2205 -- CALENDER, etc. Operation determines which of the predefined
2206 -- procedures was called. Actuals is a list of expressions which
2207 -- yield values for the formal parameters to the procedure denoted
2208 -- by operation.
2209 --
2210   ('predef_'):
2211     [-, operation, actuals, formals] := STM;
2212     PARAM_EVAL(actuals, formals, ['predef__', operation]);
2213 --
2214     ('predef__'):
2215       [-, operation] := STM;
2216       PREDEF(operation);
2217
2218 -- ['bind', name, oexpr]
2219 --
2220 --   Bind is used in the parameter evaluation sequence for out or in out
2221 --   parameters to find and save the IOBJECT (identity) of the parameter.
2222 --   Name is the name of a temporary in which the IOBJECT corresponding
2223 --   to oexpr is saved.
2224 --
2225     ('bind'):
2226       [-, name, oexpr] := STM;
2227       exec( [['oeval_', oexpr],
2228              ['bind_', name] ]);
2229 --
```

```
2230    ('bind_'):
2231      [-, name] := STM;
2232      pop( iobject );
2233      EMAP(name) := iobject;
```

Chapter 7. PACKAGES

```
2235 --  ['package', package_name, decls]
2236 --
2237 --   This statement is used to make the "scope" of a package visible to
2238 --   the interpreter, in order to recognize tasks belonging to that
2239 --   package.
2240 --
2241   ('package'):
2242    [-, package_name, decls] := STM;
2243    push( TASKS_DECLARED );
2244    TASKS_DECLARED := {};
2245    exec ( decls with ['package_', package_name] );
2246 --
2247 --
2248 --   ['package_body', package_name, [decls, stmts, hndlr]]
2249 --
2250 --     This statement allows the interpreter to know that tasks
2251 --     declared within a package specification or a package body
2252 --     must be activated.
2253 --
2254  ('package_body'):
2255    [-, package_name, blk] := STM;
2256    push( TASKS_DECLARED );
2257    TASKS_DECLARED := EMAP(package_name) ? GLOBAL_EMAP(package_name)
2258                                         ? {};
2259    exec( [['package_', package_name]] );
2260    if blk = [ ] then                -- dummy package body
2261      if nonempty(TASKS_DECLARED) then
2262        exec( [['activate_']] );
2263      end if;
2264    else                             -- real package body
2265      [decls, stmts, hndlr] := blk;
2266      exec( decls + [ ['activate_'], ['block', [[ ], stmts, hndlr]] ]);
2267    end if;
2268 --
2269 -- restore TASKS_DECLARED
2270 --
2271  ('package_'):
2272    [-, package_name] := STM;
2273    EMAP(package_name) := TASKS_DECLARED;
2274    pop( TASKS_DECLARED );
```

Chapter 8. VISIBILITY RULES

8.5 Renaming declarations

```
2277 -- ['renames', newname, oldnameexpr]
2278 --
2279 -- The renames statement appears in the object program only for the case
2280 -- of variables (where it may effect partial evaluation). The other
2281 -- cases of renaming are simply name resolution issues. Newname is the
2282 -- new name which is to be made equivalent to the iobject to which
2283 -- oldnameexpr evalutates.
2284 --
2285  ('renames'):
2286    [-, newname, oldnameexpr] := STM;
2287    exec( [['oeval_', oldnameexpr],
2288           ['renames_', newname] ]);
2289
2290  ('renames_'):
2291    [-, newname] := STM;
2292    pop( oldname );
2293    EMAP(newname) := oldname;
```

Chapter 9. TASKS

9.1 Task specifications and task bodies

```
2296 -- ['current_task', name]
2297 --
2298 --  Puts name in the current environment and sets it to the current_task
2299 --  iobject (that is, the one executing the 'current_task' instruction).
2300 --  This instruction is generated at the top of each task body, to allow,
2301 --  within the body of task type T (including within task bodies nested
2302 --  within T), references to the current task object of type T using the type
2303 --  name T (LRM 9.?).  These references are converted by the front end to
2304 --  references to a unique internal variable (whose name is supplied to this
2305 --  instruction) for each task t and which contains t itself as value.
2306 --
2307   ('current_task'):
2308     [-, name] := STM;
2309     EMAP(name) := EMAP('current_task');
2310
2311 -- ['task_body', type_name, body]
2312 --
2313 -- Task_body picks up the body that corresponds to a task type
2314 -- and stores it in the EMAP entry reserved for it. Task_bodys
2315 -- are defined separately from the rest of a task type specification
2316 -- due to difficulties with binding separately compiled specifications
2317 -- and bodies. The task body statement must appear in the bound AIS code
2318 -- after the task type declaration and before the end of the first
2319 -- declaration list in which an object of that type is declared.
2320 --
2321   ('task_body'):
2322     [-, type_name, body] := STM;
2323     exec([['teval_', type_name],
2324           ['task_body_', body]]);
2325 --
2326   ('task_body_'):
2327     [-, body] := STM;
2328     pop( type_desc );
2329     [-, -, -, taskbody] := type_desc;
2330     EMAP(taskbody) := body;
```

9.3 Task execution - Task activation

```
2332 -- ['activate_']
2333 --
2334 --  The activate_ statement is placed at the end of delaration lists
2335 --  by block and proc statements, and called by VEVAL for 'new'. It
2336 --  calls ACTIVATE_TASK to do the activation of any task objects
2337 --  declared in the current scope. The names of these tasks are in
2338 --  TASKS_DECLARED (but we discard those which have been aborted since
2339 --  their elaboation). Then the current task is put to sleep, waiting
2340 --  for all its subtasks to complete their activation.
2341 --
2342  ('activate_'):
2343    if STRACE >= 5 then DO_DUMPS(['TASKS_DECLARED']); end if;
2344    TASKS_DECLARED := { [taskid, taskbody] ∈ TASKS_DECLARED
2345                      | taskid ∉ TERMINATED_TASKS  };
2346    if nonempty(TASKS_DECLARED) then
2347       disable;
2348         ACTIVATING_SUBTASKS(CURTASK) := {};
2349         loop forall task ∈ TASKS_DECLARED do
2350            ACTIVATE_TASK(task);
2351         end loop forall;
2352 -- If an exception is raised in ACTIVATE_TASK so that no task is
2353 -- to be activated, we do not give up control
2354         if nonempty(ACTIVATING_SUBTASKS(CURTASK)) then
2355           READY_TASKS .lesstup := CURTASK;
2356         end if;
2357         TASKS_DECLARED := {};
2358       enable;
2359       SCHEDULE;
2360    end if;
2361
2362 --  ['signal_', parent_task, status]
2363 --
2364 --  When a dependant task has completed its activation, it reports to
2365 --  the parent task, and continue its execution. The special_handler
2366 --  used during activation is discarded.
2367 --
2368 --  When a dependant task has raised an exception during its activation
2369 --  it reports to its parent, but raises a TASKING_ERROR exception in
2370 --  its parent environment, and is immediately terminated.
2371 --
2372  ('signal_'):
2373     [-, parent_task, status] := STM;
2374     case status of
2375     ('error'):
2376       STSQT(parent_task) :=
2377          [['raise', 'TASKING_ERROR', 'exception during activation']];
```

```
2378        READY_TASKS .lesstup:= CURTASK;
2379        ACTIVE_TASKS less:= CURTASK;
2380        TERMINATED_TASKS with:= CURTASK;
2381        UNCREATE(CURTASK);
2382      ('ok'):
2383        HANDLER := [ ];
2384      end case;
2385      ACTIVATING_SUBTASKS(parent_task) less:= CURTASK;
2386      if empty( ACTIVATING_SUBTASKS(parent_task) ) then
2387        if parent_task ∉ TERMINATED_TASKS then
2388          READY_TASKS with:= parent_task;
2389        end if;
2390        ACTIVATING_SUBTASKS(parent_task) := Ω;
2391      end if;
2392      SCHEDULE;
```

9.4 Task dependence - Task termination

```
2394 -- ['terminate']
2395 --
2396 -- The  terminate statement is the last statement of a task body. It
2397 -- terminates the current task.
2398 --
2399   ('terminate'):
2400     exec( [['terminate_']] );
2401     COMPLETED_TASKS with:= CURTASK; --
2402     WAIT_SUBTASKS;
2403
2404   ('terminate_'):
2405     disable;
2406       READY_TASKS .lesstup:= CURTASK;
2407       ACTIVE_TASKS less:= CURTASK;
2408       COMPLETED_TASKS less:= CURTASK; --
2409       CHECK_UNSERVICED_RENDEZVOUS(CURTASK); -- (RM 11.5, 11.5(b) )
2410       TERMINATED_TASKS with:= CURTASK;
2411       UNCREATE(CURTASK);
2412       CHECK_MASTER(CURTASK);
2413     enable;
2414     SCHEDULE;
2415
2416 -- ['finish_']
2417 --
2418 --   The finish_ statement unconditionally ends execution. It is placed
2419 --   in  the  statement sequence after the call to a main program, and
2420 --   issued by a finish check statement when execution can no longer
2421 --   continue.
2422 --
2423   ('finish_'):
2424     TERMINATE_EXECUTION;
```

```
2425    return;
2426 --
2427 -- ['finish_check_']
2428 --
2429 --  The finish_check_ instruction is executed only by the IDLE task.
2430 --  The IDLE task is executed only when there are no other ready tasks.
2431 --  If in addition, there are no delayed tasks, then there is no
2432 --  possible way a user task can ever be executed, and we are
2433 --  either finished or deadlocked.
2434 --
2435  ('finish_check_'):
2436    disable;
2437    if READY_TASKS = [IDLE] and empty(DELAYED_TASKS) then
2438      TO_LIST(' System inactive ');
2439      DUMP_TASKING_INFO(ERRFILE);
2440      DUMP_TASKING_INFO(LISFILE);
2441      exec( [ ['finish_'] ]);
2442    end;
2443    enable;
```

9.5 Entries, entry calls and accept statements

```
2445 -- ['ecall', taskexpr, entryexpr, actuals]
2446 --
2447 --  The  ecall statement  is used for a call on an  entry (ADA itself does
2448 --  not distinguish  procedure and entry calls,  but  the  distinction is
2449 --  made clearly at this level.) Taskexpr evaluates to the name of the
2450 --  task to be entered, and entryexpr evaluates to the entry. Actuals
2451 --  is a list of expressions for the parameters (as in procedure calls).
2452 --
2453  ('ecall'):
2454    [-, taskexpr, entryexpr, actuals] := STM;
2455    exec( [['veval_', taskexpr],
2456           ['ecall_', actuals] ]);
2457    push_entryid(entryexpr);
2458
2459  ('ecall_'):
2460    [-, actuals] := STM;
2461    pop( task );
2462    pop( index );
2463    pop( entry );
2464    [tag, taskid] := task;
2465    entryid := [entry, index];
2466    [-, formals] := EMAP(entry); --
2467    PARAM_EVAL(actuals, formals, ['ecall__', entryid, taskid]);
2468 --
2469  ('ecall__'):
2470    [-, entryid, taskid] := STM;
2471    disable;
```

```
2472 --
2473       if taskid ∈ TERMINATED_TASKS ∔ COMPLETED_TASKS + ABNORMAL_TASKS
2474       then
2475 -- RM 9.5, 11.5
2476 --
2477         enable;
2478         exec( [ ['raise', 'TASKING_ERROR', 'task terminated'] ]);
2479 --
2480       elseif present(entry_map := OPEN_ENTRIES(taskid))
2481             and nonempty(rendez_bodies := entry_map{entryid})
2482       then -- immediate rendezvous possible
2483
2484 -- Close the open entries in the called_task, cancel any delay, and take
2485 -- it out of the set of terminatable tasks.
2486
2487         OPEN_ENTRIES(taskid) := Ω;
2488         DELAYED_TASKS(taskid) := Ω;
2489         TERMINATABLE less:= taskid;
2490         enable;
2491         MAKE_RENDEZVOUS(CURTASK, taskid, entryid, arb rendez_bodies);
2492 --
2493       else -- wait for call to be serviced
2494 --
2495         READY_TASKS .lesstup:= CURTASK;
2496         WAITING_TASKS(taskid)(entryid) with:= [CURTASK, [ ]];
2497         SCHEDULE;
2498         enable;
2499 --
2500       end if;
2501 --
2502 --
2503 -- ['endrv_', entry, enteringtask, orig_prio]
2504 --
2505 --  The endrv_ (end  rendezvous) operation is executed on completion of
2506 --  execution of the block associated with an  entry to terminate the
2507 --  rendezvous action. It is placed in the statement sequence by
2508 --  MAKE_RENDEZVOUS (q.v.) . Entry is the entry, enteringtask the
2509 --  calling task, and orig_prio the priority of the task accepting the
2510 --  entry call (which may have been increased in order to run the rendezvous
2511 --  at the priority required by the RM 9.8).
2512 --
2513 --
2514  ('endrv_'):
2515    [-, entry, enteringtask, orig_prio] := STM;
2516    if enteringtask ∈ ABNORMAL_TASKS then   -- caller might have
2517                                            -- been aborted (RM 9.10)
2518      ABNORMAL_TASKS   less:= enteringtask;
2519      TERMINATED_TASKS with:= enteringtask;
2520    else
```

```
2521      [-, formals] := EMAP(entry); --
2522      BIND_EXIT(formals, VALSTACKT(enteringtask));
2523      disable;
2524        READY_TASKS with:= enteringtask;
2525        ENTERING lessf:= enteringtask;
2526      enable;
2527    end if;
2528    TASK_PRIO(CURTASK) := orig_prio;
2529    exec( [ ['end_'] ]);
2530    SCHEDULE;
2531
2532 -- ['accept', entryname, blk]
2533 --
2534 --  This statement is used only for accepts not part of a selective
2535 --  wait statement (q.v.) . Entryname is the name of the entry for
2536 --  which a rendezvous is to be accepted, and blk is the list of
2537 --  statements to be executed during the rendezvous.
2538 --
2539   ('accept'):
2540     [-, entryname, blk] := STM;
2541     exec( [ ['accept_', blk] ]);
2542     push_entryid(entryname);
2543 --
2544 --
2545   ('accept_'):
2546     [-, blk] := STM;
2547     pop(index);
2548     pop(entry);
2549     entryid := [entry, index];
2550     rendez_body := [blk, [ ]];
2551     disable;
2552 --
2553       e_call fromb WAITING_TASKS(CURTASK)(entryid);
2554       if present(e_call) then  -- immediate rendezvous
2555 --
2556         [caller, stmsq] := e_call;
2557         STSQT(caller) := stmsq + STSQT(caller);
2558         DELAYED_TASKS(caller) := Ω;
2559         TERMINATABLE less:= caller;
2560         enable;
2561         MAKE_RENDEZVOUS(caller, CURTASK, entryid, rendez_body);
2562 --
2563       else  -- must wait for request
2564 --
2565         OPEN_ENTRIES(CURTASK) := {[entryid, rendez_body]};
2566         READY_TASKS .lesstup:= CURTASK;
2567         enable;
2568         SCHEDULE;
2569       end if;
```

9.6 Delay statement

```
2571 -- ['delay', delayval]
2572 --
2573 --  The  delay  statement simply  unreadies the current task and makes the
2574 --  appropriate  entry in the delay  list. Note that  this processing only
2575 --  applies to  delay  statements outside a select,  for which it is never
2576 --  necessary to enter a  non-null  body in the delay list (select handles
2577 --  the other case itself). Delayval is the duration of the delay.
2578 --
2579  ('delay'):
2580    [-, delayval] := STM;
2581    exec( [['veval_', delayval],
2582          ['delay_'] ]);
2583
2584  ('delay_'):
2585    pop(delay);
2586    delayval := convert_duration(delay);
2587    if delayval > 0 then -- a delay <= 0 has no effect (RM 9.6)
2588      disable;
2589        DELAYED_TASKS(CURTASK) := [delayval, [ ]];
2590        READY_TASKS .lesstup:= CURTASK;
2591      enable;
2592      SCHEDULE;
2593    end if;
```

9.7.1 Selective wait

```
2595 -- ['selective_wait, select_list, else_body]
2596 --
2597 --  Select_list is a set of components of four types:
2598 --
2599 --    ['accept', entryname, rvous_block, after_block]
2600 --
2601 --    ['delay', delay_expr, delay_block]
2602 --
2603 --    ['guard', guard_expr, guarded_statement]
2604 --
2605 --    ['terminate']
2606 --
2607 --  The 'accept' and 'delay' components have the same form as the
2608 --  statements of those names. In the 'guard' component, guarded_statement
2609 --  is either an accept, a delay, or a terminate.
2610 --
2611 --  The second parameter to selective_wait is the else_body, which is
2612 --  omitted if it is absent in the source code.
2613 --
2614 --  The first  step in the processing of selective_wait is to evaluate
```

```
2615 --  guards and entryname or delay_expr for accept or delay statements.
2616 --
2617 --
2618  ('selective_wait'):
2619    [-, select_list, else_body] := STM;
2620    exec( [['selective_wait_', #select_list, else_body] ]);
2621    loop forall s ∈ select_list do
2622      if opcode(s) = 'guard' then  -- Evaluate guards first - RM 9.7.1
2623        [-, guard_expr, stmt] := s;
2624        exec( [['veval_', guard_expr],
2625              ['if_',
2626               [['select_true_guard_', stmt]],
2627               [['select_false_guard_']]      ]] );
2628      else
2629        exec([['select_true_guard_', s]]); -- open alternative if no guard
2630      end if;
2631    end loop forall s;
2632 --
2633  ('select_true_guard_'):
2634    [-, stmt] := STM;
2635    case opcode(stmt) of
2636        -- RM 9.7.1 requires entryname or delay to be
2637        -- evaluated immediatly after guard
2638 --
2639      ('accept'):
2640        [-, entryexpr, r_block, a_block] := stmt;
2641        exec( [['select_accept_', r_block, a_block]] );
2642        push_entryid(entryexpr);
2643 --
2644      ('delay'):
2645        [-, delay_expr, delay_block] := stmt;
2646        exec( [['veval_', delay_expr],
2647              ['select_delay_', delay_block]] );
2648 --
2649      ('terminate'):
2650        push( ['terminate'] );
2651 --
2652    end case;
2653 --
2654  ('select_false_guard_'):
2655    push( ['false_guard'] ); -- a place holder on VALSTACK
2656 --
2657  ('select_accept_'):
2658    [-, r_block, a_block] := STM;
2659    pop( index );
2660    pop( entry );
2661    entryid := [entry, index];
2662    push( ['accept', entryid, r_block, a_block] );
2663 --
```

```
2664  ('select_delay_'):
2665    [-, d_block] := STM;
2666    pop( delay );
2667    delayval := convert_duration(delay);
2668    push( ['delay', delayval, d_block] );
2669 --
2670  ('selective_wait_'):
2671 --
2672 -- When all the members of select_list have been evaluated
2673 -- we decide what action to take
2674 --
2675    [-, num_alts, else_body] := STM;
2676    can_terminate := false;
2677    delays := {};
2678    accepts := {};
2679    num_closed := 0;
2680    loop forall i ∈ {1..num_alts} do -- accumulate evaluated alternates
2681                                     -- of the various classes from VALSTACK
2682      pop( alt );
2683      case opcode(alt) of
2684 --
2685        ('false_guard'):
2686          num_closed +:= 1;
2687 --
2688        ('terminate'):
2689          can_terminate := true;
2690 --
2691        ('delay'):
2692          [-, delayval, d_block] := alt;
2693          delays with:= [delayval, d_block];
2694 --
2695        ('accept'):
2696          [-, entryid, r_block, a_block] := alt;
2697          accepts with:= [entryid, [r_block, a_block]];
2698 --
2699      end case;
2700    end loop forall;
2701 --
2702    disable;
2703      sched_flag := false;
2704      if num_closed = num_alts then -- no open alternates so
2705                                    -- RM 9.7.1 (9, 11) applies
2706        enable;
2707        if present(else_body) then
2708          STSQ := else_body + STSQ;
2709        else                        --
2710          exec( [['raise', 'PROGRAM_ERROR', 'no open alternative']] );
2711        end if;
2712 --
```

```
2713      elseif nonempty(possible_rvous := {ent ∈
2714                                    (domain WAITING_TASKS(CURTASK))
2715                                  * (domain accepts)
2716                          | WAITING_TASKS(CURTASK)(ent) /= [ ]})
2717      then -- immediate rendezvous RM 9.7.1 (6)
2718 --
2719        entryid := case SELECT_MODE of
2720                ('FAIR'): random possible_rvous,
2721                ('ARB') : arb    possible_rvous
2722                  else om
2723              end;
2724        [caller, stmsq] fromb WAITING_TASKS(CURTASK)(entryid);
2725       STSQT(caller) := stmsq + STSQT(caller);
2726        rendez_body := arb accepts{entryid};
2727       DELAYED_TASKS(caller) := Ω;
2728       TERMINATABLE less:= caller;
2729       enable;
2730       MAKE_RENDEZVOUS(caller, CURTASK, entryid, rendez_body);
2731 --
2732      elseif present(else_body) then -- 9.7.1 (e) applies due to fact
2733                                     -- that else part precludes presence
2734                                     -- of delay or terminate
2735        STSQ := else_body + STSQ;
2736 --
2737      else -- wait until open alternate can be selected
2738 --
2739        if nonempty(delays) then -- use least delay - RM 9.7.1 note
2740          mindelay := min/{d : [d, b] ∈ delays};
2741          find [d, b] ∈ delays | d = mindelay;
2742        if d > 0 then
2743          DELAYED_TASKS(CURTASK) := [d, b];
2744          READY_TASKS .lesstup:= CURTASK;
2745          OPEN_ENTRIES(CURTASK) := accepts; -- might be empty
2746          sched_flag := true;
2747        else -- non-positive delay can be chosen without task switch
2748          STSQ := b + STSQ;
2749        end if d;
2750        elseif can_terminate then
2751        READY_TASKS .lesstup:= CURTASK;
2752        OPEN_ENTRIES(CURTASK) := accepts; -- might be empty
2753        TERMINATABLE with:= CURTASK;
2754        CHECK_MASTER(CURTASK);
2755        sched_flag := true;
2756       else -- only accepts open
2757        READY_TASKS .lesstup:= CURTASK;
2758        OPEN_ENTRIES(CURTASK) := accepts;
2759        sched_flag := true;
2760        end if;
2761        enable;
```

```
2762        cond_schedule;
2763 --
2764      end if num_closed;
```

9.7.2 Conditional entry call

```
2766 -- ['conditional_entry_call', entry_call, stmsq, else_body]
2767 --
2768 --  Entry_call has the same form as an 'ecall' statement (q.v.),
2769 --  stmsq is a list of statements to be executed after the rendezvous
2770 --  requested in the entry_call, and else_body is a list of statements
2771 --  to be executed if that rendezvous can not immediatly take place.
2772 --
2773  ('conditional_entry_call'):
2774    [-, entry_call, stmsq, else_body] := STM;
2775
2776 --  Note that conditional entry call is the same as a timed entry call
2777 --  with a delay of zero.
2778
2779    exec( [['timed_entry_call', entry_call, stmsq,
2780                ['delay', ['ivalue', fix_fri(0)]],
2781          else_body] ] );
```

9.7.3 Timed entry calls

```
2783 -- ['timed_entry_call', entry_call, stmsq1, delay_stmt, stmsq2]
2784 --
2785 --  Entry_call has the same form as an 'ecall' statement (q.v.),
2786 --  stmsq1 is a list of statements to be executed should the rendezvous
2787 --  requested take place within the duration specified by the
2788 --  delay_stmt, and stmsq2 is a list of statements to be executed if
2789 --  the rendezvous does not take place.
2790 --
2791 ('timed_entry_call'):
2792    [-, entry_call, stmsq1, delay_stmt, stmsq2] := STM;
2793    [-, taskexpr, entryexpr, actuals] := entry_call;
2794    [-, delay_expr] := delay_stmt;
2795    exec( [['veval_', taskexpr],
2796          ['timed_entry_call_', actuals, stmsq1, delay_expr, stmsq2]]);
2797    push_entryid(entryexpr);
2798 --
2799  ('timed_entry_call_'):
2800    [-, actuals, stmsq1, delay_expr, stmsq2] := STM;
2801    pop( task );
2802    pop( index );
2803    pop( entry );
2804    [tag, taskid] := task;
2805    entryid := [entry, index];
2806    [-, formals] := EMAP(entry);
```

```
2807    PARAM_EVAL(actuals, formals,
2808      ['timed_entry_call__', delay_expr, entryid, taskid, stmsq1,
2809                          stmsq2] );
2810 --
2811 --
2812  ('timed_entry_call__'):
2813   [-, delay_expr, entryid, taskid, stmsq1, stmsq2] := STM;
2814   exec( [['veval_', delay_expr],
2815         ['timed_entry_call___', entryid, taskid, stmsq1, stmsq2]] );
2816
2817  ('timed_entry_call___'):
2818   [-, entryid, taskid, stmsq1, stmsq2] := STM;
2819   pop( delay );
2820   delayval := convert_duration(delay);
2821   disable;
2822 --
2823     if taskid ∈ TERMINATED_TASKS + COMPLETED_TASKS + ABNORMAL_TASKS
2824     then
2825 --
2826       enable;
2827       exec( [['raise', 'TASKING_ERROR', 'task terminated']] );
2828 --
2829     elseif present(entry_map := OPEN_ENTRIES(taskid))
2830           and nonempty(rendez_bodies := entry_map{entryid})
2831     then -- immediate rendezvous
2832
2833 -- Close the open entries in the called_task, cancel any delay, and take
2834 -- it out of the set of terminatable tasks.
2835
2836       OPEN_ENTRIES(taskid) := Ω;
2837       DELAYED_TASKS(taskid) := Ω;
2838       TERMINATABLE less:= taskid;
2839       enable;
2840       STSQ := stmsq1 + STSQ;
2841       MAKE_RENDEZVOUS(CURTASK, taskid, entryid, arb rendez_bodies);
2842 --
2843     elseif delayval > 0 then -- wait for rendezvous or for delay to time out
2844 --
2845       READY_TASKS .lesstup:= CURTASK;
2846       WAITING_TASKS(taskid)(entryid) with:= [CURTASK, stmsq1];
2847       DELAYED_TASKS(CURTASK) := [delayval, stmsq2];
2848       enable;
2849       SCHEDULE;
2850 --
2851     else -- delayval <= 0 - choose delay alternative immediatly
2852         -- (note that timed entry call with zero delay is equivalent
2853         -- to conditional entry call )
2854       enable;
2855       STSQ := stmsq2 + STSQ;
```

```
2856 --
2857       end if;
```

9.8 Priorities

```
2859 -- ['set_priority']
2860 --
2861 -- Produced by the front end as a result of pragma priority.
2862 -- Any in legitimate positions will be taken care of by the
2863 -- binder or the front end. Until necessary checks added,
2864 -- these will be treated as no_ops here.
2865 --
2866   ('set_priority'):
2867     pass;
```

9.10 Abort statement

```
2869 -- ['abort', tasknames]
2870 --
2871 --   Tasknames is a list (tuple) of tasks to be aborted.
2872 --
2873   ('abort'):
2874     [-, tasknames] := STM;
2875     exec( [ ['abort_', #tasknames] ]);
2876     loop forall exprt ∈ tasknames do
2877       exec( [ ['veval_', exprt] ]);
2878     end loop forall;
2879
2880   ('abort_'):
2881     [-, numabort] := STM;
2882     sched_flag := false;
2883     loop forall i ∈ [1..numabort] do
2884       pop( task );
2885       [tag, taskid] := task;
2886       abnormal_task_set := {taskid} + DEPENDANT_TASKS(taskid);
2887       loop forall task ∈ abnormal_task_set
2888             | task ∉ TERMINATED_TASKS + ABNORMAL_TASKS do --
2889         disable;
2890         if task ∉ ACTIVE_TASKS then -- not yet activated
2891
2892           TERMINATED_TASKS with:= task;
2893
2894         else                          -- already activated
2895
2896           if task ∉ READY_TASKS then  -- suspended for some reason
2897             DELAYED_TASKS(task) := Ω;
2898             CHECK_PENDING_RENDEZVOUS(task);
2899             CHECK_CURRENT_RENDEZVOUS(task, 'TASKING_ERROR', 'abort');
2900             HELD_TASKS less:= task;
```

```
2901          else
2902            READY_TASKS .lesstup:= task;
2903            sched_flag := true;
2904          end if;
2905
2906  --        must be done for all abnormal tasks
2907
2908          CHECK_UNSERVICED_RENDEZVOUS(task);
2909          ACTIVE_TASKS less:= task;
2910          if task ∉ ABNORMAL_TASKS then  --
2911            TERMINATED_TASKS with:= task;
2912          end if;
2913          CHECK_MASTER(task);
2914          UNCREATE(task);
2915
2916        end if task;
2917
2918        enable;
2919      end loop forall task;
2920    end loop forall;
2921    cond_schedule;
```

Chapter 11. EXCEPTIONS

11.1 exception declaration

```
2924    ('exception_decl'):          --
2925        pass;
2926
```

11.2 Exception handlers

```
2928 -- ['exception', whenlist]
2929 --
2930 -- The exception statement is used for exception handlers.
2931 -- Execution of this statement places the whenlist in the HANDLER
2932 -- slot in the top of ENVSTACK. The raise statement provides the
2933 -- only method of executing the body of an exception handler.
2934 -- The format of the whenlist is a tuple of pairs
2935 -- [exnames, handler], where exnames is a set of exception
2936 -- names and handler is the corresponding body. The name 'others'
2937 -- is used for the when others handler fielding any unfielded
2938 -- exceptions.
2939 --
2940  ('exception'):
2941    [-, whenlist] := STM;
2942    HANDLER := whenlist;
```

11.3 Raise statement

```
2944 -- ['raise', exname, cause, place]
2945 --
2946 -- The raise statement raises an exception and is used both for explicit
2947 -- raise statements which appear in the source program and for
2948 -- exceptions encountered during execution. Exname is the name of the
2949 -- exception. Place should be omitted (OM) in raise statements generated
2950 -- by the front end. It is used by the interpreter to pass information
2951 -- about the task name, procedure name, statement (and nesting level
2952 -- if the debug switch is on) at which an exception is raised
2953 -- for use in the case that a message must be issued when there is no
2954 -- handler supplied. Cause is the reason for the exception being raised.
2955 --
2956 -- Raise searches the HANDLER slot at the top of ENVSTACK for a handler
2957 -- matching exname. If one is found, it is substituted for the current
2958 -- statement sequence. Otherwise, the scope stack is popped, and the
2959 -- search is made again. If we are at the outermost scope, then a message
```

```
2960 -- is issued, and the task is terminated.
2961 --
2962  ('raise'):
2963    [-, exname, cause, place] := STM;
2964    if present(exname) then
2965      EMAP('$last_exception') := exname;
2966    else
2967      exname := EMAP('$last_exception');
2968    end if;
2969    cause := cause ? '';
2970    place := place ? ERR_PLACE();
2971    if is_simple_name(exname) then
2972      TERMINATE_UNACTIVATED;
2973      if HANDLER /= [ ] and exists [exnames, stm_list] ∈ HANDLER |
2974            exname ∈ exnames or 'others' ∈ exnames then
2975        STSQ := stm_list;
2976        HANDLER := [ ]; -- RM 11.4.1 (f)
2977      else -- no local handler
2978          --  - use handler in (dynamically) enclosing scope
2979
2980        ----- yp17
2981        -- The information collected here is used in DUMP_TASKING_INFO
2982        -- for debugging purposes explained in that procedure :
2983        TASKS_WITH_RAISED_EXC with:= [ CURTASK, exname, place ];
2984        CHECK_CURRENT_RENDEZVOUS(CURTASK, exname, cause);
2985        STSQ := [['raise_', exname, cause, place ]]; --
2986        WAIT_SUBTASKS;
2987      end if;
2988    else
2989      POP_ENVSTACK;
2990      SYSTEM_ERROR('raise');
2991    end if;
2992
2993  ('raise_'):
2994    [-, exname, cause, place] := STM;
2995    if #ENVSTACK <= 1 then
2996 --
2997      TO_LIST('*** Exception ' + Unqualed_name(exname) +
2998            if cause = 'static' then
2999              ' (detected at compile-time)'
3000            elseif cause = '' then
3001              ''
3002            else
3003              ' (' + cause + ')' end +
3004            ' raised in ' + place );
3005      TO_LIST('*** No handler, task is terminated');
3006      if CURTASK = MAINTASK then
3007        exec( [['finish_']] );
3008      else
```

```
3009        exec( [['terminate']]);
3010      end if CURTASK;
3011    else
3012      POP_ENVSTACK;
3013      exec( [['raise', exname, cause, place]] );
3014    end if;
```

Chapter 13. IMPLEMENTATION DEPENDENCIES

13.1 Representation clauses

```
3017 -- ['access_size_repr', access_type, access_size]
3018 --
3019  ('access_size_repr'):
3020    pass;
3021 --
3022 -- ['enum_size_repr', type_name, lower_bound, upper_bound, enum_map]
3023 --
3024 --  The enumeration representation clause is used to change the internal
3025 --  integer code values associated with the literal names of the
3026 --  enumeration type.
3027 --
3028  ('enum_size_repr'):
3029    [-, typename, lbd, ubd, enum_map] := STM;
3030    EMAP(typename) := ['enum', lbd, ubd, enum_map];
3031 --
3032 -- ['size_repr', typename, size_value]
3033 --
3034  ('size_repr'):
3035    pass;
3036 --
3037 -- ['task_size_repr', task_type, task_size]
3038 --
3039  ('task_size_repr'):
3040    pass;
```

13.8 Machine code insertions

```
3042 -- ['code', expr]
3043 --
3044 -- This is the Ada code statement.  It is ignored in this implementation.
3045 --
3046  ('code'):
3047    pass;
3048 --
3049 -- ['pragma', name, arg_list]
3050 --
3051 -- The pragma command allows interpreter support of pragmas.
3052 -- Name is the name of the pragma, and arg_list is a list of
3053 -- arguments (if any). If no run time support is provided for
3054 -- a given pragma, then this statement acts as a no-op.
```

```
3055 --
3056 ('pragma'):
3057     [-, name, arg_list] := STM;
3058     case name of
3059 --
3060 -- Pragmas supported are:
3061 --
3062 --
3063 --  ['pragma', 'EDUMP']
3064 --
3065 --  The edump (environment dump)  provides a standardized dump of
3066 --  STSQ, EMAP, and CONTENTS for debugging the Ada interpreter .
3067 --
3068        ('EDUMP'):
3069           TO_LIST(CURTASK + ' **** environment dump begin ****');
3070          do_dumps(['STSQ', 'EMAP', 'CONTENTS']);
3071           TO_LIST('**** environment dump end ****');
3072 --
3073 --  ['pragma', 'TDUMP']
3074 --
3075 --  The tdump provides a standardized dump of tables internal to the
3076 --  io packages.
3077 --
3078       ('TDUMP'):
3079          PREDEF('dump');
3080 --
3081 --  ['pragma', 'TRACE_ON']
3082 --
3083 --  Turns on full statement trace of the task within which it is found.
3084 --
3085       ('TRACE_ON'):
3086            TRACED_TASKS with:= CURTASK;
3087 --
3088 --  ['pragma', 'TRACE_OFF']
3089 --
3090 --  Turns off statement trace of the task in which it is found.
3091 --
3092 --  Note that neither of these pragmas effect traces controlled
3093 --  by the /trace parameter of the ADA command.
3094 --
3095       ('TRACE_OFF'):
3096            TRACED_TASKS less:= CURTASK;
3097 --
3098 --  ['pragma', 'DUMP', table_list]
3099 --
3100 --  Dumps the internal interpreter tables named in the table_list.
3101 --
3102       ('DUMP'):
3103          do_dumps(arg_list);
```

```
3104 --
3105 --  ['pragma', 'TRACE_TABLES_ON', table_list]
3106 --
3107 --  Causes the internal interpreter tables named in the table_list
3108 --  to be dumped after each instruction in the task in which it is
3109 --  found is executed.
3110 --
3111     ('TRACE_TABLES_ON'):
3112        TRACED_TABLES(CURTASK) := arg_list;
3113 --
3114 --  ['pragma', 'TRACE_TABLES_OFF']
3115 --
3116 --  Stops any table tracing that may have been selected for the task
3117 --  in which it is found.
3118 --
3119 --  Note that neither of the above two pragmas effect any table tracing
3120 --  that may have been selected as a result of the /trace parameter of
3121 --  the ADA command.
3122 --
3123     ('TRACE_TABLES_OFF'):
3124        TRACED_TABLES(CURTASK) := Ω;
3125 --
3126     ('DEBUG'):
3127        exec( [['pragma', '$DEBUG', #arg_list]] );
3128        loop forall arg ∈ arg_list do
3129          exec( [['veval_', arg]] );
3130        end loop forall;
3131 --
3132     ('$DEBUG'):
3133        loop forall i ∈ [1..arg_list] do
3134          pop( value );
3135          TO_LIST( str value );
3136        end loop forall;
3137 --
3138   end case;
3139 --
3140 -- ['stmt', first_token ]
3141 --
3142 --  Updates the current source statement number kept by the interpreter
3143 --  for use in error messages or trace output. If the relevant compiler
3144 --  option was not selected, then these statements will not be present
3145 --  in the AIS code.
3146 --
3147  ('stmt'):
3148   [-, line_number] := STM; --
3149   EMAP('line_number') := line_number; --
3150 --
3151 --
3152 -- trap for unknown AIS statements
```

```
3153 --
3154   else
3155       SYSTEM_ERROR('unknown stmt : ' + str STM);
3156 --
3157  -- End of main Case statement for statement types
3158 --
3159 --
3160 end case opcode(STM);
3161 --
3162 end proc INTERP;
```

INTERPRETER PROCEDURES

START_THE_CLOCK

```
3165 proc START_THE_CLOCK;
3166 --
3167 -- Initializes the system and task clocks.
3168 --
3169   CLOCK_TICK := 500;
3170   LAST_CLOCK_TICK := time;
3171   LAST_CLOCK_TICK_CURTASK := LAST_CLOCK_TICK;
3172   NEXT_INTERRUPT_TIME := LAST_CLOCK_TICK; -- force immediate interrupt
3173 --
3174 end proc START_THE_CLOCK;
```

INITIALIZE_DEBUG_INFO

```
3176 proc INITIALIZE_DEBUG_INFO;
3177 --
3178 -- Initialize the statement count and traceing.
3179 --
3180   STMTCOUNT := 0;
3181   CURRENT_STMT_COUNT := 0;
3182 --
3183   DEBUGGING := (STRACE > 0);
3184   TRACED_TASKS := {};
3185   TRACED_TABLES := {};
3186 --
3187 -- The information initialized here is used in DUMP_TASKING_INFO
3188 -- for debugging purposes explained in that procedure :
3189
3190   FULL_TASKNAME_MAP := {};
3191   TASKS_WITH_RAISED_EXC := {};
3192 --
3193 end proc INITIALIZE_DEBUG_INFO;
```

CHECK_CLOCK_INTERRUPT

```
3195 proc CHECK_CLOCK_INTERRUPT;
3196 --
3197 -- Called on each cycle thru the interpreter loop to
3198 -- simulate the effect of a hardware clock interrupt.
3199 --
3200   CURRENT_TIME := time;
```

```
3201 --
3202   if CURRENT_TIME >= NEXT_INTERRUPT_TIME then
3203 --
3204     SERVICE_CLOCK;
3205 --
3206   end if CURRENT_TIME;
3207 --
3208 end proc CHECK_CLOCK_INTERRUPT;
```

SERVICE_CLOCK

```
3210 proc SERVICE_CLOCK;
3211 --
3212 -- This routine would service the hardware clock interrupt if there
3213 -- were one. It updates the software clocks, ages the delay queue,
3214 -- and calls the scheduler if necessary.
3215 --
3216     actual_delay := CURRENT_TIME - LAST_CLOCK_TICK;
3217     NEXT_INTERRUPT_TIME := CURRENT_TIME + CLOCK_TICK;
3218     LAST_CLOCK_TICK := CURRENT_TIME;
3219 --
3220     SYSTEM_CLOCK +:= actual_delay;
3221 -----    UPDATE_TASK_CLOCK;
3222 --
3223     ADJUST_DELAYS(actual_delay);
3224 --  if SIMULATE_WAIT = 0 then NO_WAITING; end if;
3225     READY_DELAYED;
3226 --
3227     if TIME_SLICE /= 0 then -- don't allow tasks
3228                             -- to run until blocked
3229                             -- use time slices instead
3230 --
3231       DELAY_TIMER -:= actual_delay;
3232 --
3233  -- We make a scheduling decision if
3234  -- the current task has timed out.
3235 --
3236       if DELAY_TIMER <= 0 then
3237         DELAY_TIMER := TIME_SLICE;
3238         SCHEDULE;
3239       end if DELAY_TIMER;
3240 --
3241     end if TIME_SLICE;
3242 --
3243 end proc SERVICE_CLOCK;
```

DEBUGGING_CHECKS_1

```
3245 proc DEBUGGING_CHECKS_1;
3246 --
3247 -- Checks to be done before the execution of the current statement:
3248 --
3249 -- Check for empty statement.
3250 --
3251  if STM = [ ] then
3252    SYSTEM_ERROR('Missing code');
3253  end if;
3254
3255 --
3256 --
3257 -- Check for trace requested.
3258 --
3259  if STRACE >= 1 or CURTASK ∈ TRACED_TASKS then
3260   if (is_ais_stmt or STRACE >= 2) and opcode(STM) /= 'stmt' then
3261    if STRACE > 1 and TRACE_MODE /= 'free' and TRACE_COUNT <= 0 then
3262     TO_ERRFILE('?');
3263     reada(DATAFILE, t);
3264     if type (t ? [ ]) /= 'INTEGER' then
3265       t := 0;
3266     end if;
3267
3268     TRACE_MODE := if t = 0 then
3269                     'free'
3270                   elseif t > 0 then
3271                     'list'
3272                   else
3273                     'nolist'
3274                   end;
3275     TRACE_COUNT := abs t;
3276    end if;
3277    if STRACE = 1 or TRACE_MODE /= 'nolist' then
3278     PRETTY_PRINT(ERRFILE, [ERR_PLACE(), STM]);
3279    end if;
3280    if TRACE_MODE /= 'free' then
3281     TRACE_COUNT -:= 1;
3282    end if;
3283   end if;
3284  end if;
3285 --
3286 --
3287 end proc DEBUGGING_CHECKS_1;
```

DEBUGGING_CHECKS_2

```
3289 proc DEBUGGING_CHECKS_2;
3290 --
3291 -- Checks to be done after the execution of the current statement:
3292 --
3293 -- Check for dump of internal tables.
3294 --
3295  if present(tables := TRACED_TABLES(CURTASK)) then
3296   if is_ais_stmt or STRACE >= 2 then
3297    DO_DUMPS(tables);
3298   end if;
3299  elseif STRACE = 3 then
3300   DO_DUMPS(['VALSTACK']);
3301  elseif STRACE = 4 then
3302   DO_DUMPS(['VALSTACK', 'READY_TASKS', 'WAITING_TASKS',
3303         'OPEN_ENTRIES']);
3304  end if;
3305 --
3306 --
3307 --
3308 end proc DEBUGGING_CHECKS_2;
```

TERMINATE_EXECUTION

```
3310 proc TERMINATE_EXECUTION;
3311 --
3312 -- Actions to be taken just prior to interpreter returning to its caller.
3313 --
3314  UPDATE_TASK_CLOCK;
3315 --
3316  PREDEF_TERM;
3317 --
3318  -- Output task clocks and statement counts.
3319 --
3320  if CDEBUG5 /= 0 or STRACE > 0 then
3321   TO_LIST('TASK      ' + lpad('TASK CLOCKS', 20) +
3322         lpad('STATEMENT COUNTS', 20));
3323   loop forall [task, mytime] ∈ TASK_CLOCKS do
3324    mystmts := TASK_COUNTS(task);
3325    TO_LIST(rpad(task, 10) + lpad(str mytime, 20) +
3326      lpad(str mystmts, 20) );
3327   end loop forall;
3328  end if;
3329 --
3330 --
3331 --
3332  TO_LIST(' ');
```

```
3333  TO_LIST(' Execution complete');
3334 --
3335 end proc TERMINATE_EXECUTION;
```

INIT_ENV

```
3337 proc INIT_ENV(rd predef_procs);
3338 --
3339 -- Called during compiler initialization (in ADAMAIN).
3340 --
3341 -- Predef_procs are declarations for procedures or objects in the
3342 -- standard environment or text_io. Init_env initializes internal
3343 -- tables, and then calls the interpreter with predef_procs as the
3344 -- statement sequence, to allow the initialization of EMAP with
3345 -- this precompiled material. The interpreter tables are then cleaned
3346 -- up to allow user programs to be run on subsequent normal calls of
3347 -- the interpreter.
3348 --
3349  -- Note that MAINTASK, and IDLE are constants
3350  -- defined in the directory.
3351  UNINITIALIZED := newat;
3352
3353  CONTENTS := {}; -- initialized here because global to all tasks
3354
3355  TRACE_MODE := 'list';
3356  TRACE_COUNT := 0;
3357 --
3358  QUAL_MAP := { ['record', 'qual_discr_'],
3359               ['array', 'qual_index_'],
3360               ['integer', 'qual_range_'],
3361               ['float', 'qual_range_'],
3362               ['fixed', 'qual_range_'],
3363               ['enum', 'qual_range_']    };
3364 --
3365  TASKNAME_MAP := { [MAINTASK, MAINTASK], [IDLE, IDLE] };
3366 --
3367  INITIALIZE_TASKING;
3368 --
3369 --
3370 -- Create the initial environment stack
3371  empty_env := [ [{}, [[ ]], [ ], [ ], {}, ''']];
3372  ENVSTACKT := { [IDLE, empty_env], [MAINTASK, empty_env] };
3373  STSQT(IDLE) := [ ['loop', [['finish_check_']]] ];
3374 --
3375  GLOBAL_EMAP := {};              --
3376  EMAPT(MAINTASK) := INITIALIZE_STANDARD();
3377 --
3378  SELECT_MODE := 'FAIR';
3379  TIME_SLICE := 0;
```

```
3380  STRACE := 0;
3381  AIS_MAIN := '';
3382  AIS_CODE := (predef_procs ? [ ]) + [['finish_']];
3383  AIS_PRIO := [ ];
3384  INTERP();
3385 --
3386  AIS_CODE := [ ];
3387  STSQT(MAINTASK) := [ ]; -- code entered from AIS_CODE by interpreter
3388  GLOBAL_EMAP := EMAPT(MAINTASK); -- to increase speed, better have a
3389                                  -- map containing things that never
3390                                  -- change
3391 --
3392 --
3393 end proc INIT_ENV;
```

TASKING PROCEDURES (Chap. 9)

ACTIVATE_TASK

```
3396 proc ACTIVATE_TASK(task);
3397 --
3398 -- Elaboration of the declarations is complete. RM 11.4.2 (d) requires
3399 -- exceptions raised during elaboration of a task's declarations to
3400 -- propagate to.the activating unit. This insures it, and allows
3401 -- termination of not yet activated tasks .
3402 -- (RM 9.3 - proc TERMINATE_UNACTIVATED)
3403 --
3404     [taskid, taskbody] := task;
3405     body := EMAP(taskbody);
3406     if absent(body) then
3407       exec([['raise', 'PROGRAM_ERROR',
3408             'Access to task body before elaboration']]); --
3409       return;
3410     end if;
3411     [decls, stmts, hndlr] := body;
3412     special_hndlr :=
3413           [['exception',
3414             [[{'others'},
3415               [['signal_', CURTASK, 'error']
3416           ]]]]];
3417     STSQT(taskid) :=
3418       [['block',
3419         [ special_hndlr + decls + [['signal_', CURTASK, 'ok']],
3420           stmts,
3421           hndlr
3422         ]
3423       ]];
3424     EMAPT(taskid) +:= (EMAP lessf 'current_task');          --
3425     ACTIVE_TASKS with:= taskid;
3426     READY_TASKS with:= taskid;
3427     ACTIVATING_SUBTASKS(CURTASK) with:= taskid;
3428
3429 end proc ACTIVATE_TASK;
```

ADJUST_DELAYS

```
3431 proc ADJUST_DELAYS(actual_delay);
3432 --
3433 -- Adjusts the delay times remaining in delayed tasks
```

```
3434 -- by subtracting the amount of time since the last
3435 -- time the clock was serviced.
3436 --
3437  disable;
3438 --
3439  DELAYED_TASKS := {[t, [d - actual_delay, b]] :
3440                          [d, b] = DELAYED_TASKS(t)};
3441 --
3442  enable;
3443 --
3444 end proc ADJUST_DELAYS;
```

CHECK_MASTER

```
3446 proc CHECK_MASTER(taskid);
3447 --
3448 -- Called when a task terminates, is aborted, or is waiting at
3449 -- a selective wait with an open terminate alternative.
3450 -- This may allow the master to complete termination,
3451 -- leave a scope, or its decendants to be terminated.
3452 -- (RM 9.4 and 9.7.1 (f) )
3453 --
3454  if STRACE >= 5 then
3455   DO_DUMPS(['ACTIVE_TASKS', 'TERMINATED_TASKS',
3456             'COMPLETED_TASKS', 'ABNORMAL_TASKS',  --
3457             'TERMINATABLE', 'HELD_TASKS']);
3458  end if;
3459  my_master := MASTER(taskid);
3460  disable;
3461  if my_master ∈ HELD_TASKS or my_master ∈ TERMINATABLE then
3462   d_set := DEPENDANT_TASKS(my_master);
3463   if my_master ∈ HELD_TASKS and empty(d_set) then
3464 --    master may complete its termination
3465     READY_TASKS with:= my_master;
3466     HELD_TASKS less:= my_master;
3467   elseif my_master ∈ TERMINATABLE
3468     and (empty(d_set) or d_set subset TERMINATABLE) then
3469     CHECK_MASTER(my_master);
3470   elseif nonempty(d_set) and d_set subset TERMINATABLE then
3471 --    all tasks in d_set may be terminated
3472     loop forall d ∈ d_set do
3473       STSQT(d) := [['terminate']];
3474       OPEN_ENTRIES(d) := Ω;
3475       READY_TASKS with:= d;
3476     end loop forall;
3477     TERMINATABLE -:= d_set;
3478   end if;
3479   if STRACE >= 5 then
3480     DO_DUMPS(['ACTIVE_TASKS', 'TERMINATED_TASKS',
```

```
3481                'COMPLETED_TASKS', 'ABNORMAL_TASKS', --
3482                'TERMINATABLE', 'HELD_TASKS']);
3483    end if STRACE;
3484  end if my_master;
3485  enable;
3486 end proc CHECK_MASTER;
```

CHECK_CURRENT_RENDEZVOUS

```
3488 proc CHECK_CURRENT_RENDEZVOUS(taskid, exname, cause);    --
3489 --
3490 -- Here we test for the case of an exception which is raised (or
3491 -- received) by the called task during a rendezvous and propagate either
3492 -- a copy of the exception to the entering task, first testing that the
3493 -- entering task is still attempting to enter.
3494 --
3495      last_stmt_taskid := STSQT(taskid)(#STSQT(taskid)); --
3496      if present(last_stmt_taskid) and
3497        opcode(last_stmt_taskid) = 'endrv_' then
3498        [-, -, enteringtask] := last_stmt_taskid; --
3499        if ENTERING(enteringtask) = taskid then
3500          STSQT(enteringtask) :=
3501            [['raise', exname, cause ]] + STSQT(enteringtask);--
3502          ENTERING lessf:= enteringtask;
3503          READY_TASKS with:= enteringtask;
3504        end if;
3505      end if;
3506 end proc CHECK_CURRENT_RENDEZVOUS;
```

CHECK_PENDING_RENDEZVOUS

```
3508 proc CHECK_PENDING_RENDEZVOUS(taskid);
3509 --
3510 -- Used by task being aborted to check
3511 -- if it is waiting for a rendezvous and cancel it.
3512 --
3513 -- if it is engaged into a rendezvous, put it into ABNORMAL_TASKS
3514 --
3515  if exists ent = WAITING_TASKS(t) | (nonempty(ent)
3516    and (exists q = ent(e) | (q /= [ ]
3517      and (exists i ∈ {1..#q} | q(i)(1) = taskid )))) then
3518    WAITING_TASKS(t)(e)(i..i) := [ ];
3519  end if;
3520  if present(ENTERING(taskid)) then   -- Engaged in a rendezvous
3521     ABNORMAL_TASKS with:= taskid;
3522  end if;
3523 end proc CHECK_PENDING_RENDEZVOUS;
```

CHECK_UNSERVICED_RENDEZVOUS

```
3525 proc CHECK_UNSERVICED_RENDEZVOUS(taskid);
3526 --
3527 -- Used by a task that is terminating or being aborted
3528 -- to check for tasks waiting on its entry queues which
3529 -- have not been serviced. If any are found, TASKING_ERROR
3530 -- is raised in the calling task.
3531 --
3532   e := WAITING_TASKS(taskid) ? {};
3533   if nonempty(e) then
3534     loop forall elist = e(ename) | elist /= [ ] do
3535       loop forall [t, -] ∈ elist do
3536         STSQT(t) := [['raise', 'TASKING_ERROR', 'called task aborted']];
3537         READY_TASKS with:= t;
3538         DELAYED_TASKS(t) := Ω;
3539       end loop forall;
3540     end loop forall;
3541     WAITING_TASKS(taskid) := {};
3542   end if;
3543   loop forall called = ENTERING(caller) | called = taskid do
3544    STSQT(caller) := [['raise', 'TASKING_ERROR', 'called task aborted']];
3545     READY_TASKS with:= caller;
3546   end loop;
3547 end proc CHECK_UNSERVICED_RENDEZVOUS;
```

CREATE_TASK

```
3549 proc CREATE_TASK(taskid, priority, entryids, t_master, t_taskenv);
3550 --
3551 -- Create_task is used to create a task when a task object
3552 -- declaration has been elaborated.
3553 --
3554   ENVSTACKT(taskid) := [ [{}, [ ], [ ], [ ], {}, ''] ];--
3555   MASTER(taskid) := t_master;
3556   TASKENV(taskid) := t_taskenv;
3557   WAITING_TASKS(taskid) := {[entryid, [ ]] : entryid ∈ entryids};
3558   TASK_PRIO(taskid) := priority;
3559   TASK_CLOCKS(taskid) := 0;
3560   TASK_COUNTS(taskid) := 0;
3561 end proc CREATE_TASK;
```

DEPENDANT_TASKS

```
3563 proc DEPENDANT_TASKS(root);
3564 --
3565 -- Does transitive closure of MASTER relationship
3566 -- for active tasks.
```

```
3567 --
3568  d_set := {t ∈ ACTIVE_TASKS |
3569            MASTER(t) = root and TASKENV(t) = #ENVSTACKT(root)};
3570  loop forall d ∈ d_set do
3571    d_set +:= DEPENDANT_TASKS(d);
3572  end loop forall;
3573  return d_set;
3574 end proc DEPENDANT_TASKS;
```

INITIALIZE_TASKING

```
3576 proc INITIALIZE_TASKING;
3577 --
3578 -- Called by Init_env during compiler initialization. Creates the
3579 -- initial tasking environment.
3580 --
3581  TASK_PRIO := { [IDLE, -1], [MAINTASK, 9] };
3582  ACTIVE_TASKS := {IDLE, MAINTASK};
3583  READY_TASKS := [IDLE, MAINTASK];
3584  OPEN_ENTRIES := {};
3585  MASTER := {};
3586  DELAYED_TASKS := {};
3587  HELD_TASKS := {};
3588  ENTERING := {};
3589  WAITING_TASKS := { [IDLE, {}], [MAINTASK, {}] };
3590  ACTIVATING_SUBTASKS := {};          --
3591  TASKENV := {};
3592  TERMINATABLE := {};
3593  TERMINATED_TASKS := {};
3594  COMPLETED_TASKS := {};              --
3595  ABNORMAL_TASKS := {};               --
3596  DELAY_TIMER := 0;
3597  CURTASK := MAINTASK;
3598  TASK_CLOCKS := { [IDLE, 0], [MAINTASK, 0] };
3599  TASK_COUNTS := { [IDLE, 0], [MAINTASK, 0] };
3600  SYSTEM_CLOCK := 0;
3601 --
3602 --
3603 end proc INITIALIZE_TASKING;
```

␣

```
3605 op .lesstup(tname, entry);
3606 --
3607  -- Produces a tuple without entry
3608 --
3609  assert is_tuple(tname);
3610  if exists i ∈ {1..#tname} | tname(i) = entry then
3611    tname(i..i) := [ ];
```

```
3612  end if;
3613  return tname;
3614 --
3615 end op .lesstup;
```

MAKE_RENDEZVOUS

```
3617 proc MAKE_RENDEZVOUS(calling_task, called_task, entryid, rendez_body);
3618 --
3619 -- Initiates a rendezvous between two tasks. Used symmetrically by
3620 -- entering task or accepting task when they discover a rendezvous
3621 -- to be possible.
3622 --
3623  disable;
3624  ENTERING(calling_task) := called_task;
3625  READY_TASKS .lesstup:= calling_task;
3626  enable;
3627  [entryname, -] := entryid;
3628  [br, b] := rendez_body;
3629  STSQT(called_task) := b + STSQT(called_task);
3630  ENVSTACKT(called_task) with:=
3631     [EMAPT(called_task), [ ], [ ], [ ], {}, ''];
3632  STSQT(called_task) := br +
3633     [['endrv_', entryname, calling_task, TASK_PRIO(called_task)]];
3634  TASK_PRIO(called_task) max:= TASK_PRIO(calling_task); -- RM 9.8
3635  entry_object := EMAPT(calling_task)(entryname);
3636  [-, formals] := entry_object;
3637  BIND_ENTER(formals, EMAPT(called_task), VALSTACKT(calling_task));
3638  disable;
3639  READY_TASKS := [called_task] + (READY_TASKS .lesstup called_task);
3640  enable;
3641  SCHEDULE;
3642 end proc MAKE_RENDEZVOUS;
```

NO_WAITING

```
3644 proc NO_WAITING;
3645 --
3646 -- If there are no tasks to be scheduled, then we can reduce all the
3647 -- delay times (to avoid useless simulated waiting!)
3648 --
3649  disable;
3650    if READY_TASKS = [IDLE] and nonempyt(DELAYED_TASKS) then
3651 --  compute the minimum positive delay until a delayed task is ready
3652      mindelay := 0 max (min/{d : [d, b] = DELAYED_TASKS(t)});
3653      DELAYED_TASKS := {[t, [d - mindelay, b]]:
3654                                  [d, b] = DELAYED_TASKS(t)};
3655      SYSTEM_CLOCK +:= mindelay;
3656      TASK_CLOCKS(IDLE) +:= mindelay;
```

```
3657     end if;
3658   enable;
3659 --
3660 end proc NO_WAITING;
```

READY_DELAYED

```
3662 proc READY_DELAYED;
3663 --
3664 -- Ready any tasks whose delay period is completed.
3665 --
3666   disable;
3667   loop forall [d, b] = DELAYED_TASKS(t) | d <= 0 do
3668     OPEN_ENTRIES(t) := Ω;
3669     STSQT(t) := b + STSQT(t);
3670     READY_TASKS with:= t;
3671     DELAYED_TASKS(t) := Ω;
3672     if exists e_queue = WAITING_TASKS(c_task)
3673         | exists t_list = e_queue(e_name)
3674           | exists l_elmt ∈ t_list
3675             | l_elmt(1) = t then
3676       WAITING_TASKS(c_task)(e_name) .lesstup:= l_elmt;
3677     end if;
3678   end loop forall;
3679   enable;
3680 end proc READY_DELAYED;
```

SCHEDULE

```
3682 proc SCHEDULE;
3683 --
3684 -- SCHEDULE
3685 -- --------
3686 --
3687 -- Makes a scheduling decision.
3688 --
3689 --
3690   UPDATE_TASK_CLOCK;
3691   disable;
3692 --
3693  -- IDLE is always in READY_TASKS and of lowest priority
3694 --
3695     maxprio := max/[TASK_PRIO(t) : t ∈ READY_TASKS];
3696     find t ∈ READY_TASKS | TASK_PRIO(t) = maxprio;
3697     CURTASK := t;
3698     if TIME_SLICE /= 0 then
3699
3700 --     round-robin
3701       READY_TASKS .lesstup:= CURTASK;
```

```
3702        READY_TASKS with:= CURTASK;
3703      else
3704
3705   --   fifo
3706        pass; -- find does a sequential search
3707      end if;
3708    enable;
3709   --
3710   --
3711  end proc SCHEDULE;
```

TERMINATE_UNACTIVATED

```
3713  proc TERMINATE_UNACTIVATED;
3714  --
3715  -- Called when exception raised to terminate
3716  -- any unactivated tasks declared in current scope.
3717  --
3718    loop forall t = MASTER(taskid)
3719      |    t = CURTASK
3720        and TASKENV(taskid) = #ENVSTACK
3721        and taskid ∉ ACTIVE_TASKS
3722    do
3723      TERMINATED_TASKS with:= taskid;
3724      CHECK_UNSERVICED_RENDEZVOUS(taskid); -- (RM 11.5)
3725      UNCREATE(taskid);
3726    end loop forall;
3727  end proc TERMINATE_UNACTIVATED;
```

UPDATE_TASK_CLOCK

```
3729  proc UPDATE_TASK_CLOCK;
3730  --
3731  -- Used to insure accurate task clocks. Adds accumulated time
3732  -- (and count of statements executed) to software clock entry
3733  -- for current task. Called before any possible task switch.
3734  --
3735    actual_delay := CURRENT_TIME - LAST_CLOCK_TICK_CURTASK;
3736  --
3737    TASK_CLOCKS(CURTASK) +:= actual_delay;
3738    TASK_COUNTS(CURTASK) +:= CURRENT_STMT_COUNT;
3739  --
3740    LAST_CLOCK_TICK_CURTASK := CURRENT_TIME;
3741    CURRENT_STMT_COUNT := 0;
3742  end proc UPDATE_TASK_CLOCK;
```

UNCREATE

```
3744 proc UNCREATE(task);
3745 --
3746 -- Reclaims table space used by task being terminated.
3747 --
3748  WAITING_TASKS(task) := Ω;
3749  OPEN_ENTRIES(task) := Ω;
3750  TASK_PRIO(task) := Ω;
3751  loop forall t = MASTER(child)
3752           | t = task and TASKENV(child) = #ENVSTACKT(task)
3753  do
3754   MASTER(child) := Ω;
3755   TASKENV(child) := Ω;
3756  end loop forall;
3757  ENVSTACKT(task) := Ω;
3758 end proc UNCREATE;
```

WAIT_SUBTASK

```
3760 proc WAIT_SUBTASKS;
3761 --
3762 -- Checks to see if there are any tasks which are subtasks of the
3763 -- current environment which are still active. If so, the current
3764 -- task is placed in hold status to await termination of these subtasks.
3765 -- The decendants of the current environment are checked to see if they
3766 -- may be terminated.The call to wait_subtasks must always be the last
3767 -- action performed in interpreting a single statement form, since it
3768 -- can cause loss of control ( removal of the task from the ready list).
3769 -- Usually the call to wait_subtasks is preceded by an exec statement
3770 -- which specifies the action to be taken after the wait (if any) is
3771 -- completed.
3772 --
3773  if STRACE >= 5 then
3774   DO_DUMPS(['MASTER', 'TASKENV', 'ACTIVE_TASKS', 'TERMINATED_TASKS',
3775            'TERMINATABLE', 'HELD_TASKS']);
3776  end if;
3777  disable;
3778  d_set := DEPENDANT_TASKS(CURTASK);
3779  if nonempty(d_set) then
3780   READY_TASKS .lesstup:= CURTASK;
3781   HELD_TASKS with:= CURTASK;
3782   if d_set subset TERMINATABLE then
3783    loop forall d ∈ d_set do
3784     STSQT(d) := [['terminate']];
3785     OPEN_ENTRIES(d) := Ω;
3786     READY_TASKS with:= d;
3787    end loop forall;
```

```
3788      TERMINATABLE -:= d_set;
3789    end if;
3790    if STRACE >= 5 then
3791      DO_DUMPS(['ACTIVE_TASKS', 'TERMINATED_TASKS', 'TERMINATABLE',
3792              'HELD_TASKS']);
3793    end if;
3794    SCHEDULE;
3795  end if;
3796  enable;
3797 end proc WAIT_SUBTASKS;
```

SCOPE PROCEDURES (Chap. 6)

BIND_ENTER

```
3800 proc BIND_ENTER(rd formals, rw called_emap, rd caller_valstack);
3801 --
3802 -- Used to bind actual arguments, which have been evaluated and stored
3803 -- on the VALSTACK of the caller, to the formals of a called procedure
3804 -- or entry. Parameter formals is the formal parameter list obtained
3805 -- from the procedure's IVALUE, caller_valstack is the VALSTACK of the
3806 -- caller and called_emap is the environment map of the called procedure
3807 -- or entry.
3808 --
3809 --
3810  loop forall i ∈ [1..#formals] do
3811    [formalname, -, -] := formals(i);
3812    arg_obj := caller_valstack(#caller_valstack - i + 1);
3813    called_emap(formalname) := arg_obj;
3814  end loop forall;
3815 end proc BIND_ENTER;
```

BIND_EXIT

```
3817 proc BIND_EXIT(rd formals, rw caller_valstk);
3818 --
3819 -- Called on exit from an entry or procedure to
3820 -- unstack the no longer needed arguments from VALSTACK.
3821 --
3822  caller_valstk(#caller_valstk - #formals + 1 ..) := [ ];
3823 end proc BIND_EXIT;
```

PARAM_EVAL (Chap.6)

```
3825 proc PARAM_EVAL(actuals, formals, operation);
3826 --
3827 -- Sets up evaluation of actual parameter lists. Actuals is a list of
3828 -- o_exprs for the actual parameters. Formals is the signature of the
3829 -- procedure being called. Operation is an interpreter statement to be
3830 -- executed when the parameters are evalated onto the VALSTACK.
3831 --
3832 --
3833 -- Semantics of parameter passing in the interpreter:
3834 --
3835 --       At the interface between the calling procedure and the called
3836 --   procedure (or entry) several distinct operations may be required
3837 --   by the sematics of ADA beyond the simple identification of the
3838 --   formal with the actual parameter (here called 'binding'). These
3839 --   include constraint checking, type conversions, and for records,
3840 --   access types and arrays, the imposition of the constraints of the
3841 --   actual parameter on the formal parameter.
3842 --
3843 --       In order to conveniently perform the various combinations of
3844 --   operations required, the operation supplied as a parameter,
3845 --   ( e.g. a 'call__' or a 'ecall_' statement, which use BIND_ENTER
3846 --   to do the binding) will be surrounded by various auxilliary
3847 --   statements. Binding is done by reference for records and arrays,
3848 --   and by value for scalars and access parameters. In those cases where
3849 --   semantics require call by value/return, the auxilliary statements
3850 --   will create temporary variables into which the value of the actual
3851 --   parameter will be copied, and the call will be by reference to these
3852 --   temporarys. Each of the temporarys or actuals (as applicable) is
3853 --   evaluated and placed on VALSTACK. After the evaluation sequence,
3854 --   the first parameter will be on top of the stack, and the other
3855 --   parameters will be below it, in order.
3856
3857 --
3858 -- Skeleton statement sequences for the various cases are expressed
3859 -- below: (Note Tx stands for temporary x, and similarly for Ty)
3860 --
3861 --   Scalar in, unconstrained formal:
3862 --
3863 --     ['object', Tx, arg]
3864 --     ['oeval_', Tx]
3865 --     operation
3866 --
3867 --   Scalar in, constrained formal:
```

```
3868 --
3869 --    ['object', Tx, ['qual_range', formal_type, arg]]
3870 --    ['oeval_', Tx]
3871 --    operation
3872 --
3873 --  Scalar out :
3874 --
3875 --    ['bind', Ty, arg]
3876 --    ['object', Tx, formal_type]
3877 --    ['oeval_', Tx]
3878 --    operation
3879 --    [':=', Ty, ['qual_range', actual_type, Tx]]
3880 --
3881 --  Scalar in out :
3882 --
3883 --    ['bind', Ty, ['qual_range', formal_type, arg]]
3884 --    ['object', Tx, Ty]
3885 --    ['oeval_', Tx]
3886 --    operation
3887 --    [':=', Ty, ['qual_range', actual_type, Tx]]
3888 --
3889 --  Note: for scalar parameters where a type conversion is required,
3890 --    the actual parameter will appear as a 'convert' operation in
3891 --    actuals. In these cases, the above sequences will be modified to
3892 --    contain the needed 'convert' operations on entry and/or exit from
3893 --    the called procedure.
3894 --
3895 --  Arrays and records in, unconstrained formal:
3896 --
3897 --    ['oeval_', arg]
3898 --    operation
3899 --
3900 --  Arrays in, constrained formal:
3901 --
3902 --    ['oeval_', [['qual_index', formal_type, arg]]]
3903 --    operation
3904 --
3905 --  Records in, constrained formal:
3906 --
3907 --    ['oeval_', [['qual_discr', formal_type, arg]]]
3908 --    operation
3909 --
3910 --  Arrays or records out or in out, unconstrained formal, constrained
3911 --  actual:
3912 --
3913 --    ['oeval_', arg]
3914 --    operation
3915 --
3916 --  Records out or in out, unconstrained formal, unconstrained actual:
```

```
3917 --  (An optimization - can safely use the next pattern)
3918 --
3919 --    ['oeval_', arg]
3920 --    operation
3921 --
3922 --  Records out or in out, constrained formal, unconstrained actual:
3923 --
3924 --    ['oeval_', [['make_constrained', ['qual_discr',
3925 --        formal_type, arg], formal_type]]]
3926 --    operation
3927 --
3928 --  Arrays out or in out, constrained formal, constrained actual:
3929 --
3930 --    ['oeval_', [['qual_index', formal_type, arg]]]
3931 --    operation
3932 --
3933 --  Records out or in out, constrained formal, constrained actual:
3934 --
3935 --    ['oeval_', [['qual_discr', formal_type, arg]]]
3936 --    operation
3937 --
3938 --  Access types:
3939 --        On entry to a procedure, an access type parameter of mode in
3940 --     or inout appears as a convert operation of the form:
3941 --
3942 --     ['convert', actual_object_type, formal_object_type, access_argument]
3943 --
3944 --     to facilitate constraint checks. This convert will generate
3945 --     a 'qual_access', which will in turn execute the appropriate
3946 --     'qual_discr' for a record, 'qual_index' for an array, or
3947 --     'qual_range' for a scalar.
3948 --
3949 --
3950   param_list := [ ];
3951   pre_stmt_list := [ ];
3952   post_stmt_list := [ ];
3953
3954   loop for fp ∈ formals do
3955     [fname, fmode, ftype] := fp;
3956
3957 --   type name must be supplied for text_io - no corresponding actual
3958
3959   if ftype = '$type' then
3960         param_list with:= fname;
3961
3962   else
3963       arg fromb actuals;
3964       f_itype := GET_ITYPE(ftype);
3965
```

```
3966      case type_mark(f_itype) of

          Scalar types

3968      ('integer', 'float', 'fixed', 'enum'):
3969
3970   -- temporary variables :
3971
3972        temp_in := '$temp' + str newat;
3973        param_list with := temp_in;
3974        if fmode /= 'in' then
3975            temp_out := '$temp' + str newat;
3976        end if;
3977
3978   -- Constrained scalar :
3979
3980        if is_tuple arg and opcode(arg) = 'convert' then
3981
3982          case fmode of
3983
3984          ('in'):
3985            pre_stmt_list with:= ['object', [temp_in], ftype, arg];
3986          ('out'):
3987            [-, atype, -, scalar_arg] := arg;
3988            pre_stmt_list +:=
3989              [ ['bind', temp_out, scalar_arg],
3990                ['object', [temp_in], ftype] ];
3991            post_stmt_list with :=
3992              [':=', temp_out, ['convert', ftype, atype, temp_in] ];
3993         ('inout'):
3994            [-, atype, -, scalar_arg] := arg;
3995            pre_stmt_list +:=
3996              [ ['bind', temp_out, scalar_arg],
3997                ['object', [temp_in], ftype, arg ] ];
3998            post_stmt_list with :=
3999              [':=', temp_out, ['convert', ftype, atype, temp_in] ];
4000          else
4001            SYSTEM_ERROR('PARAM_EVAL');
4002          end case fmode;
4003
4004        else
4005
4006   -- Unconstrained scalar
4007
4008          case fmode of
4009
4010          ('in'):
4011            pre_stmt_list with :=
4012              ['object', [temp_in], ftype, ['qual_range', ftype, arg]];
```

```
4013          ('out'):
4014            pre_stmt_list +:=
4015              [ ['bind', temp_out, arg],
4016                ['object', [temp_in], ftype] ];
4017            post_stmt_list with:= [':=', temp_out, temp_in];
4018          ('inout'):
4019            pre_stmt_list +:=
4020              [['bind', temp_out, arg],
4021               ['object', [temp_in], ftype,
4022                                  ['qual_range', ftype, temp_out]]];
4023            post_stmt_list with:= [':=', temp_out, temp_in];
4024          else
4025            SYSTEM_ERROR('PARAM_EVAL');
4026          end case fmode;
4027
4028        end if is_tuple arg;
4029
```

Task

```
4031      ('task'):
4032        temp_in := '$temp' + str newat;
4033        param_list with := temp_in;
4034        pre_stmt_list with:= ['object', [temp_in], ftype, arg];
```

Array

```
4036      ('array'):
4037
4038  [-, -, -, is_constrained] := f_itype;
4039        if is_oexpr(arg) then        -- it is a valid left-hand side
4040          if is_constrained then
4041            param_list with:= ['qual_index', ftype, arg];
4042          else
4043            param_list with:= arg;
4044          end if;
4045        else
4046
4047  -- The argument is an array literal, or array valued function
4048  -- call or attribute. We construct an object for it.
4049
4050          temp_obj := '$temp' + str newat;
4051          pre_stmt_list with:= ['constant', [temp_obj], ftype, arg];
4052          if is_constrained then
4053            param_list with:= ['qual_index', ftype, temp_obj];
4054          else
4055            param_list with:= temp_obj;
4056          end if;
4057        end if;
```

Access

```
4059      ('access'):
4060
4061          [-, formal_obj_type] := f_itype;
4062          formal_obj_itype := GET_ITYPE(formal_obj_type);
4063
4064          temp_in := '$temp' + str newat;
4065          param_list with := temp_in;
4066          if fmode /= 'in' then
4067              temp_out := '$temp' + str newat;
4068          end if;
4069
4070          case fmode of
4071
4072          ('in'):
4073            pre_stmt_list with:= ['object', [temp_in], ftype, arg];
4074
4075          ('out'):
4076            if is_tuple arg and opcode(arg) = 'convert' then
4077
4078 --   Constrained access
4079
4080              [-, atype, -, acc_arg] := arg;
4081              actual_itype := GET_ITYPE(atype);
4082              [-, actual_obj_type] := actual_itype;
4083              actual_qual :=
4084                  QUAL_MAP(type_mark(formal_obj_itype));
4085              if present(actual_qual) then
4086                post_stmt_list with:=
4087                  ['veval_',
4088                  ['qual_access', actual_obj_type, temp_in,
4089                     actual_qual] ];
4090              end if;
4091
4092 --   Unconstrained access
4093
4094            else
4095              acc_arg := arg;
4096            end if is_tuple;
4097
4098            pre_stmt_list +:=
4099              [ ['bind', temp_out, acc_arg],
4100                ['object', [temp_in], ftype] ];
4101            post_stmt_list with:= [':=', temp_out, temp_in];
4102
4103          ('inout'):
4104
```

```
4105          if is_tuple arg and opcode(arg) = 'convert' then
4106             [-, atype, -, acc_arg] := arg;
4107             actual_itype := GET_ITYPE(atype);
4108             [-, actual_obj_type] := actual_itype;
4109             actual_qual :=
4110                QUAL_MAP(type_mark(formal_obj_itype));
4111             if present(actual_qual) then
4112               pre_stmt_list with :=
4113                 ['veval_',
4114                 ['qual_access', formal_obj_type, acc_arg,
4115                    actual_qual] ];
4116               post_stmt_list with:=
4117                 ['veval_',
4118                 ['qual_access', actual_obj_type, temp_in,
4119                    actual_qual] ];
4120             end if;
4121          else
4122             acc_arg := arg;
4123          end if is_tuple;
4124
4125          pre_stmt_list +:=
4126             [ ['bind', temp_out, acc_arg],
4127               ['object', [temp_in], ftype, temp_out] ];
4128          post_stmt_list with:= [':=', temp_out, temp_in]; --
4129        else
4130          SYSTEM_ERROR('PARAM_EVAL');
4131        end case fmode;
```

Record

```
4133      ('record'):
4134
4135        case fmode of
4136
4137        ('in'):
4138          [-, -, constr_disc_list] := f_itype;
4139          if is_oexpr(arg) then          -- it is a valid left-hand side
4140            if present(constr_disc_list) then
4141              param_list with:= ['qual_discr', ftype, arg];
4142            else
4143              param_list with:= arg;
4144            end if;
4145          else
4146
4147 -- The argument is a record literal, or record valued function
4148 --  call or attribute. We construct an object for it.
4149
4150            temp_obj := '$temp' + str newat;
4151            pre_stmt_list with:= ['constant', [temp_obj], ftype, arg];
```

```
4152            if present(constr_disc_list) then
4153              param_list with:= ['qual_discr', ftype, temp_obj];
4154            else
4155              param_list with:= temp_obj;
4156            end if;
4157          end if;
4158
4159        ('out', 'inout'):
4160          [-, -, constr_disc_list] := f_itype;
4161          if present(constr_disc_list) then
4162              param_list with :=
4163                ['make_constrained', ['qual_discr', ftype, arg], ftype];
4164          else
4165              param_list with:= arg;
4166          end if;
4167        else
4168          SYSTEM_ERROR('PARAM_EVAL');
4169        end case fmode;
4170
4171      else
4172
4173        SYSTEM_ERROR('PARAM_EVAL');
4174
4175      end case type_mark(f_itype);
4176
4177    end if;
4178
4179  end loop for;
4180
4181 --
4182  -- Now emit the statement sequence
4183 --
4184       loop while param_list /= [ ] do
4185         pre_stmt_list with := ['oeval_', param frome param_list];
4186       end loop while;
4187       exec(   pre_stmt_list
4188             + [ operation ]
4189             + post_stmt_list );
4190
4191 end proc PARAM_EVAL;
```

is_oexpr

```
4193 proc is_oexpr(xpr);
4194
4195 -- Verify that an expression is a valid left hand side
4196 -- Used  in PARAM_EVAL
4197
4198   if is_simple_name(xpr) then
```

```
4199      return true;
4200    elseif not is_tuple(xpr)  then
4201      return false;
4202    else
4203      case opcode(xpr) of
4204      ('[ ]', '[..]', '.') :
4205        return is_oexpr(xpr(2));
4206      ('@'):
4207        return true;
4208      else
4209        return false;
4210      end case;
4211    end if;
4212
4213 end proc is_oexpr;
```

CREATE_COPY

```
4215 proc CREATE_COPY(rd arg_val, itype, index_position);
4216 --
4217 -- Returns a new IOBJECT whose IVALUE is the same as arg_val.
4218 -- If arg_val is of an array or record type, then itype is
4219 -- its type.  This is used to constrain record iobjects created
4220 -- by this routine. The parameter itype may be Ω for other types.
4221
4222  if is_literal_ivalue(arg_val) then
4223    loc := newat;
4224    CONTENTS(loc) := arg_val;
4225    return loc;
4226  elseif is_fixed_ivalue(arg_val) then
4227    loc := newat;
4228    CONTENTS(loc) := arg_val;
4229    return ['fixed_iobject', loc];
4230  else
4231    case type_mark(arg_val) of
4232 --
4233      ('task_ivalue'):
4234        loc := newat;
4235        CONTENTS(loc) := arg_val(2);
4236        TASKNAME_MAP(arg_val(2)) := '?';
4237        return ['task_iobject', loc];
4238 --
4239      ('label_ivalue'):
4240        loc := newat;
4241        CONTENTS(loc) := arg_val(2..);
4242        return ['label_iobject', loc];
4243 --
4244      ('proc_ivalue'):
4245        task_loc := newat;
4246        index_loc := newat;
4247        formals_loc := newat;
4248        body_loc := newat;
4249        [-, task, index, formals, body] := arg_val;
4250        CONTENTS(task_loc) := task;
4251        CONTENTS(index_loc) := index;
4252        CONTENTS(formals_loc) := formals;
4253        CONTENTS(body_loc) := body;
4254        return ['proc_iobject',
4255          task_loc, index_loc, formals_loc, body_loc];
4256 --
4257      ('access_ivalue'):
```

```
4258        loc := newat;
4259        CONTENTS(loc) := arg_val(2);
4260        return ['access_iobject', loc];
4261 --
4262      ('array_ivalue'):                                    --
4263        [-, val_seq, lowbound, highbound, nullval] := arg_val;
4264        index_position ?:= 1;
4265          [-, index_type_list, elem_itype] := GET_ITYPE(itype);
4266        if index_position = #index_type_list then
4267          return
4268            ['array_iobject',
4269             [ CREATE_COPY(elem, elem_itype, Ω): elem ∈ val_seq ],
4270             CREATE_COPY(lowbound, Ω, Ω),
4271             CREATE_COPY(highbound, Ω, Ω), Ω, Ω, nullval];
4272        else
4273          return
4274            ['array_iobject',
4275             [ CREATE_COPY(sub_array, itype, index_position + 1):
4276                 sub_array ∈ val_seq],
4277             CREATE_COPY(lowbound, Ω, Ω),
4278             CREATE_COPY(highbound, Ω, Ω), Ω, Ω, nullval];
4279        end if;
4280 --
4281      ('record_ivalue'):
4282
4283        [-, arg_field_values] := arg_val;
4284
4285          itype := GET_ITYPE(itype);
4286
4287          if not is_tuple itype or type_mark(itype) /= 'record' then
4288            SYSTEM_ERROR('CREATE_COPY');
4289            return ['record_iobject', {}];
4290          end if;
4291
4292          -- get the discriminants and form the discr map
4293          [-, [-, variant_part], -, disc_list] := itype;
4294          dmap := {[d, arg_field_values(d)] : d ∈ disc_list};
4295          if nonempty(dmap) then
4296             itype := APPLY_DISCR(dmap, itype);
4297                                          -- apply the discrim constr
4298          end if;
4299
4300        -- note that new_objdeclist is a map, but objdeclist is a tuple
4301        new_objdeclist := {};
4302          [-, [objdeclist, -]] := itype;
4303        loop for [field_name, [field_itype, -]] ∈ objdeclist |
4304           field_name ∈ domain arg_field_values and
4305           field_name /= 'fields_present' do
4306           new_objdeclist(field_name) :=
```

```
4307              CREATE_COPY(arg_field_values(field_name), field_itype,
4308                          Ω);
4309          end loop for;
4310
4311          if variant_part /= [ ] then      -- allocate 'fields_present'
4312            loc := newat;
4313            CONTENTS(loc) := domain new_objdeclist;
4314            new_objdeclist('fields_present') := loc;
4315          end if;
4316
4317          constr_disc_list :=
4318            if empty(disc_list) then Ω else disc_list end;
4319          return ['record_iobject', new_objdeclist, constr_disc_list];
4320 --
4321       ('$predef_op'):
4322          return arg_val;
4323 --
4324       else
4325          SYSTEM_ERROR('CREATE_COPY');
4326            return ['record_iobject', {}];
4327 --
4328       end case type_mark(arg_val);
4329    end if;
4330 end proc CREATE_COPY;
4331
```

CREATE_OBJ (Chap.3)

```
4333 proc CREATE_OBJ( itype_name, default_init,
4334                   t_master, t_taskenv, obj_name);
4335 --
4336 -- The CREATE_OBJ procedure is invoked by the create_obj_ statement of
4337 -- the interpreter when an object must be created from a type.
4338 -- The parameters are as follows:
4339 --
4340 --   itype_name     the name of the ITYPE giving the type of the object
4341 --
4342 --   default_init   set true if default initializations of record field
4343 --                  values are required.
4344 --
4345 --   t_master       the master task, used for creating task objects.
4346 --                  This is usually, but not always CURTASK.
4347 --
4348 --   t_taskenv      the task environment to be used for creating task
4349 --                  objects. This is usually, but not always, #ENVSTACK.
4350 --
4351 --   obj_name       for task only, for debugging purposes
4352 --
4353 -- On completion of execution of the create_obj_ statement, the IOBJECT
4354 -- which is created is left on VALSTACK, with all necessary entries
4355 -- made in the CONTENTS map.
4356
4357    itype := GET_ITYPE(itype_name);
4358
4359    case type_mark(itype) of
4360
```

Scalar types

```
4362    -- For all scalar types, all we do is create a new location and
4363    -- set its initial value to "UNINITIALIZED" (no value set).
4364
4365    ('integer', 'enum', 'float'):
4366      loc := newat;
4367      CONTENTS(loc) := UNINITIALIZED;
4368      push (loc);
4369
4370    ('fixed'):
4371      loc := newat;
4372      CONTENTS(loc) := UNINITIALIZED;
4373      push( ['fixed_iobject', loc] );
```

Access types

```
4375  -- For access types, initialize to NULL value as per RM 3.8 para. 3
4376
4377    ('access'):
4378      loc := newat;
4379      CONTENTS(loc) := NULL; -- RM 3.8 para. 3
4380      push( ['access_iobject', loc] );
```

Task

```
4382  -- Task case. Note that task objects are constants, and most of the
4383  -- information comprising them are in tables external to EMAP.
4384
4385    ('task'):
4386      loc := newat;
4387      taskid := '$task' + str loc;
4388      TASKNAME_MAP(taskid) := UNQUALED_NAME(obj_name);
4389      CONTENTS(loc) := taskid;
4390      if default_init then
4391        [-, entryids, prio, taskbody] := itype;
4392  --
4393        TASKS_DECLARED with:= [taskid, taskbody];
4394        CREATE_TASK(taskid, prio, entryids, t_master, t_taskenv);
4395        full_obj_name := break(obj_name, '#' );
4396        FULL_TASKNAME_MAP(taskid) := full_obj_name;
4397      -- and to allow the body of a task type to refer to itself through the
4398      -- type name (see -
4399      -- need to use the following entry, because there is no other way to
4400      -- get from CURTASK to the iobject for CURTASK):
4401        EMAPT(taskid)('current_task') := ['task_iobject', loc];
4402      end if;
4403      push( ['task_iobject', loc] );
```

Entry case

```
4405    ('entry'):
4406      [-, formals] := itype;
4407      push( ['entry_iobject', formals] );
```

Record

```
4409  -- Record case. The complex case arises with discriminants present.
4410  -- What we do in this case is to construct a new environment (using the
4411  -- environment stack in the usual manner). This environment is a copy
4412  -- of the surrounding environment, except that entries are made for all
4413  -- the discriminants of the record, referencing their values (the value
4414  -- in EMAP is directly the value of the discriminant). In addition,
```

```
4415  -- there is a standard entry, under the name 'constr_disc_list' which
4416  -- is a copy of the value of the constr_disc_list field in the itype.
4417  -- These entries are used in dealing with component arrays and
4418  -- structures which are dynamic (i.e. depend on discriminants).
4419
4420    ('record'):
4421      [-, [objdec_list, variant_part], constr_disc_list, disc_list]
4422                          := itype;
4423
4424  -- The first step is to deal with the case of a dynamic record, i.e.
4425  -- one which is constrained by one or more of the discriminants of
4426  -- an enclosing record, as indicated by the dynamic indication
4427  -- in constr_disc_list.
4428
4429      if is_discr_ref (constr_disc_list) then
4430        [-, dmap] := constr_disc_list;
4431
4432  -- First set the value of all the dynamic discriminants from the values
4433  -- specified for the discriminants of the enclosing record.
4434
4435        loop for [fname, fvalue] ∈ dmap | is_discr_ref (fvalue) do
4436          if IS_OK_DISCR(fvalue) then --
4437            [-, dname] := fvalue;
4438            dmap(fname) := EMAP(dname);
4439          else
4440            return;
4441          end if;
4442        end loop for;
4443
4444  -- If the enclosing record is constrained, then the inner one is also,
4445  -- so this is where we apply the discriminants to obtain the
4446  -- constrained type.
4447
4448        if present(EMAP('constr_disc_list')) then
4449          itype := APPLY_DISCR (dmap,
4450            ['record', [objdec_list, variant_part], Ω, disc_list]);
4451
4452  -- If the outer level is unconstrained, then so is the inner one, so
4453  -- here is where we override the initial default values of the
4454  -- discriminants with the values supplied in the discrimininant
4455  -- constraint.
4456
4457        else
4458          loop for [discrim_name, discrim_value] ∈ dmap do
4459            [field_itype, field_vexpr] := objdec_list(discrim_name);
4460            field_vexpr := ['ivalue', discrim_value];
4461            objdec_list(discrim_name) := [field_itype, field_vexpr];
4462          end loop for;
4463        end if;
```

```
4464
4465  -- Acquire modified itype, and dynamic record preprocessing is complete
4466
4467        [-, [objdec_list, variant_part], constr_disc_list, disc_list]
4468                              := itype;
4469      end if;
4470
4471  -- If there are discriminants, then we establish the new environment
4472
4473      if nonempty(disc_list) then
4474        PUSH_ENVSTACK;
4475        EMAP('constr_disc_list') := constr_disc_list;
4476      end if;
4477
4478  -- If there is no default initialization, then we can flatten out the
4479  -- variant part (if there is one), so that we include all its fields
4480  -- (which in this case will not be initialized in any case).
4481
4482      if (not default_init) and (variant_part /= [ ]) then
4483        [variant_name, altern_list] := variant_part;
4484        loop for [choices, component_list] ∈ altern_list do
4485          objdec_list +:= FLATTEN_RECORD (component_list);
4486        end loop for;
4487        variant_part := [ ];
4488      end if;
4489
4490  -- If we have no default initialization, then we just need to create
4491  -- the component objects and assemble the result.
4492
4493      if not default_init then
4494        exec ([['create_obj_', field_itype, false, t_master, t_taskenv,
4495                              obj_name]
4496              : [field_name, [field_itype, field_vexpr]]
4497                    ∈ objdec_list] +
4498            [['create_obj_r1_', objdec_list, disc_list,
4499                    constr_disc_list]]);
4500
4501  -- Otherwise if we have default initialization in the non-discriminant
4502  -- case, then we first create the component objects, then evaluate the
4503  -- default initialization expressions, then perform the initializations
4504  -- and finally assemble the result.
4505
4506      elseif empty(disc_list) then
4507        exec ([['create_obj_', field_itype,
4508                is_uninitialized(field_vexpr),   --
4509                t_master, t_taskenv, obj_name]
4510              : [field_name, [field_itype, field_vexpr]]
4511                    ∈ objdec_list] +
4512            [['veval_', field_vexpr]
```

```
4513              : [field_name, [field_itype, field_vexpr]]
4514                   ∈ objdec_list] +
4515             [['create_obj_r2_', #objdec_list]] +
4516             [['create_obj_r1_', objdec_list, disc_list,
4517                   constr_disc_list]]);
4518
4519  -- Here we have the hard case where we are performing default
4520  -- initializations, and there are discriminants present. The
4521  -- first step is to evaluate the discriminants and make the
4522  -- entries in EMAP to reflect the discriminant values. Note
4523  -- that we must make these one by one, since one discriminant
4524  -- could reference the value of a previously initialized one.
4525
4526     else
4527       exec ([['create_obj_r3_', field_name, field_vexpr]
4528             : [field_name, [field_itype, field_vexpr]]
4529                   ∈ objdec_list | field_name ∈ disc_list] +
4530            [['create_obj_r5_', objdec_list, variant_part,
4531                                  disc_list, obj_name]]);
4532     end if;
```

Array

```
4534  -- Case of array
4535
4536    ('array'):
4537
4538    [-, index_type_list, elmt_type] := itype;
4539
4540  -- Even in the case of a multidimensional array, an array_ivalue of
4541  -- array_ivalues is built.
4542
4543   total_elements := 1;
4544   total_live_elements := 1;
4545   array_structure := [ ];
4546
4547   loop for index_type ∈ index_type_list do
4548
4549    [-, lowbound, highbound] := GET_ITYPE(index_type);
4550
4551  -- Deal with the lower bound. We first create the integer location
4552  -- which will be used to hold its value. If the bound is static,
4553  -- then we set its value in CONTENTS and use this value as the
4554  -- lower bound for the created object sequence.
4555
4556  -- If the bound is dynamic, then we use the dmap parameter to
4557  -- obtain the type and value of the corresponding discriminant.
4558  -- The array object we construct takes its lower bound from the
4559  -- mimimum value of the index type (low end of its subtype range),
```

```
4560  -- since we must ensure that the constructed object is large enough to
4561  -- hold the maximum possible entry. The CONTENTS entry for the
4562  -- lower bound is copied from the actual value of the discriminant
4563  -- (and is Ω if there is no initial value).
4564
4565    if is_discr_ref (lowbound) then
4566        [-, lowdiscr, discr_itype, index_base_type] := lowbound;
4567        [-, discr_lowbound, -] := GET_ITYPE(discr_itype);
4568        [-, index_lowbound, -] := GET_ITYPE(index_base_type);
4569        actual_lowbound := index_lowbound max discr_lowbound;
4570        lowvalue := EMAP(lowdiscr) ? actual_lowbound; --
4571    else
4572      lowvalue := actual_lowbound := lowbound;
4573      lowdiscr := Ω;
4574    end if;
4575
4576  -- Deal with upper bound in a similar manner, except that we use
4577  -- the largest possible value of the index type as the high
4578  -- bound for creation of the sequence of objects.
4579
4580   if is_discr_ref (highbound) then
4581      [-, highdiscr, discr_itype, index_base_type] := highbound;
4582      [-, -, discr_highbound] := GET_ITYPE(discr_itype);
4583      [-, -, index_highbound] := GET_ITYPE(index_base_type);
4584      actual_highbound := index_highbound min discr_highbound;
4585      highvalue := EMAP(highdiscr) ? actual_highbound; --
4586    else
4587      highvalue := actual_highbound := highbound;
4588      highdiscr := Ω;
4589    end if;
4590
4591  -- If the array is a null array, it is compatible with any type,
4592  -- if it is not null, and if the bounds are discriminants, they must be
4593  -- of the index subtype, or else raise constraint_error (in IS_OK_DISCR)
4594
4595    if lowvalue <= highvalue then
4596      if is_discr_ref(highbound) and not(IS_OK_DISCR(highbound)) then
4597        return;
4598      end if;
4599      if is_discr_ref(lowbound) and not(IS_OK_DISCR(lowbound)) then
4600        return;
4601      end if;
4602    end if;
4603
4604  -- We make sure that the number of elements does not exceed the
4605  -- maximum we allow.
4606
4607    numelements := (actual_highbound - actual_lowbound + 1) max 0;
4608    live_elements := (highvalue - lowvalue + 1 ) max 0;
```

```
4609    total_live_elements *:= live_elements;
4610    total_elements *:= numelements;
4611
4612    array_structure with :=
4613      [actual_lowbound, actual_highbound, lowvalue, highvalue,
4614       lowdiscr, highdiscr];
4615
4616   end loop for;
4617
4618    if total_elements > 2**15 - 1 then
4619       exec([['raise', 'STORAGE_ERROR', 'array > 2**15 - 1 elements']]);
4620       return;
4621    end if;
4622
4623 -- The structure of the array is known, so we can construct the
4624 -- sequence of statements that will actually create the elements, then
4625 -- the final array_iobject. Care must be taken not to compute any
4626 -- initial value for elements that exist, but are not "live", i.e.
4627 -- do not belong to the array we are building presently.
4628 -- There is a special case: a null array for which a ghost is
4629 -- created immediately.
4630
4631   if total_elements = 0 then
4632      aggr := Ω;
4633      loop while array_structure /= [ ] do
4634         [-, -, lowvalue, highvalue, -, -] frome array_structure;
4635         lowobj := newat;
4636         CONTENTS(lowobj) := lowvalue;
4637         highobj := newat;
4638         CONTENTS(highobj) := highvalue;
4639         aggr := ['array_iobject', [aggr], lowobj, highobj];
4640      end loop;
4641      push (aggr + [Ω, Ω, 'null']);
4642
4643   else -- not a null array
4644
4645   stmt_list := [['create_obj_', elmt_type, default_init,
4646                  t_master, t_taskenv, obj_name ]];
4647   stmt_list_without_init := [['create_obj_', elmt_type, false,
4648                  t_master, t_taskenv, obj_name ]];
4649
4650
4651   loop while array_structure /= [ ] do
4652      [lowbound, highbound, lowvalue, highvalue,
4653               lowdiscr, highdiscr] frome array_structure;
4654
4655      lowobj := newat;
4656      CONTENTS(lowobj) := lowvalue;
4657      highobj := newat;
```

```
4658      CONTENTS(highobj) := highvalue;
4659
4660      new_list := [ ];
4661
4662  -- nlive is the number of "live" elements, ndead is the number of 'dead'
4663  -- elements, i.e. elements that may exists later on if the record
4664  -- which contains this particular array is globally re-assigned.
4665  -- In any case a 'dead' element receive any initial value, even if
4666  -- it is a task.
4667
4668      nlive := (highvalue - lowvalue + 1) max 0;
4669      if nlive /= 0 then
4670        new_list with:= ['repeat_', nlive, stmt_list];
4671      end if;
4672      ndead := ((lowbound - lowvalue) max 0)
4673            + ((highbound - highvalue) max 0);
4674      if ndead /= 0 then
4675        new_list with:= ['repeat_', ndead, stmt_list_without_init];
4676      end if;
4677      numelements := nlive + ndead;
4678      stmt_list := new_list with ['create_obj_a_', lowobj, highobj,
4679                          lowdiscr, highdiscr, numelements];
4680      stmt_list_without_init :=
4681        [ ['repeat_', numelements, stmt_list_without_init] ]
4682       with ['create_obj_a_', lowobj, highobj,
4683                          lowdiscr, highdiscr, numelements];
4684
4685    end loop while;
4686    exec( stmt_list );
4687
4688    end if total_elements = 0;
```

Others

```
4690    -- Case of delayed type definition, which is an error, since the
4691    -- definition should have been completed by the time we need it!
4692
4693     ('delayed'):
4694       SYSTEM_ERROR('Access before elaboration (private type)');
4695
4696    -- Any other itype passed means something has gone wrong!
4697
4698     else
4699       SYSTEM_ERROR('unknown type ' + str itype);
4700
4701     end case type_mark(itype);
4702
4703 end proc CREATE_OBJ;
```

IS_OK_DISCR

```
4705 proc IS_OK_DISCR(discr_ref);
4706     [-, discr, discr_itype, target_type] := discr_ref;
4707     value := EMAP(discr);
4708     if present(value) and present(target_type) then
4709       [-, lo, hi] := GET_ITYPE(target_type);
4710       if (value < lo) or (value > hi) then
4711         exec([['raise', 'CONSTRAINT_ERROR',
4712                         'discrim out of bounds']]);
4713         return false;
4714       end if;
4715     end if;
4716     return true;
4717 end proc IS_OK_DISCR;
```

FLATTEN_RECORD

```
4719 proc FLATTEN_RECORD (component_list);
4720 --
4721 -- This procedure takes a component_list and works its way
4722 -- through flattening out all variants.
4723
4724     [objdec_list, variant_part] := component_list;
4725
4726     if variant_part /= [ ] then
4727
4728       [discrim_name, altern_list] := variant_part;
4729
4730       loop for [choices, component_list] ∈ altern_list do
4731         objdec_list +:= FLATTEN_RECORD (component_list);
4732       end loop for;
4733     end if;
4734
4735     return objdec_list;
4736
4737 end proc FLATTEN_RECORD;
```

INITIALIZE_STANDARD

```
4739 proc INITIALIZE_STANDARD;
4740 --
4741 -- This routine creates the structures used to initialize
4742 -- the standard environment. Called by INIT_ENV.
4743 --
4744 --
4745 macro fix_0;  fix_fri(0)  endm;
4746 macro fix_day;  fix_fri(86400)  endm;
```

```
4747 macro fix_hundredth; [1, 100] endm;
4748 macro fix_infinity;  [1, 0]   endm;
4749
4750  NULL := '$null';
4751  null_loc := newat;
4752  CONTENTS +:= { [null_loc, NULL] };
4753
4754  return {[ 'INTEGER', ['integer', ADA_MIN_INTEGER, ADA_MAX_INTEGER]],
4755          ['NATURAL', ['integer', 0, ADA_MAX_INTEGER ]],
4756          ['POSITIVE', ['integer', 1, ADA_MAX_INTEGER ]],
4757 --
4758 --
4759          ['FLOAT', ['float',
4760                   ADA_MIN_REAL, ADA_MAX_REAL, ADA_REAL_DIGITS]],
4761          ['$FIXED', ['fixed', fix_umin(fix_infinity), --
4762                      fix_infinity,
4763                      fix_0 ] ],
4764          ['universal_fixed', ['fixed', fix_umin(fix_infinity),
4765                                        fix_infinity,
4766                                        fix_0 ] ],
4767 --
4768           ['PRIORITY', ['integer', 0, 9 ]],
4769           ['PRIORITY_RANGE', ['integer', 0, 9 ]],
4770          ['BOOLEAN', ['enum', boolean_false, boolean_true,
4771                       {[boolean_false, 'FALSE'],
4772                       [boolean_true, 'TRUE'] }]],
4773
4774          ['DURATION', ['fixed', fix_umin(fix_day), --
4775                       fix_day,
4776                       fix_hundredth ]],
4777 --
4778          ['TASK', ['task'] ],
4779 --
4780  -- character as an enumeration type
4781 --
4782          ['CHARACTER', ['enum', 0, 127,
4783             {[i, "'" + char i + "'"] : i ∈ {0..127}}]],
4784 --
4785 --
4786 --
4787          ['STRING', ['array', 'POSITIVE', 'CHARACTER', false] ],
4788 --
4789 -- Access variable with NULL pointer for setting access
4790 -- objects to null.
4791 --
4792          ['NULL', ['access_iobject', null_loc]]
4793 --
4794 --
4795                              }
```

```
4796 --
4797 -- names of predefined operators (used by renames)
4798 --
4799       + {[op_name, ['$operator', op_name]] : op_name ∈
4800          {
4801          ':=', '=', '/=', '<', '<=',
4802          '>', '>=', '&',
4803          'and', 'or', 'xor',
4804          'not',
4805          'row', 'andthen',
4806          'orelse',
4807          '+i', '+fl', '+fx', '-i', '-fl', '-fx',
4808          '*i', '*fl', '*fx', '/i', '/fl', '/fx',
4809          '*ifx', '*fxi', '/fxi', '**i', '**fl',
4810          '+ui', '+ufl', '+ufx', '-ui', '-ufl', '-ufx',
4811          'absi', 'absfl', 'absfx',
4812          'modi', 'remi'
4813          } };
4814
4815 drop fix_0, fix_day, fix_hundredth, fix_infinity;      --
4816
4817 end proc INITIALIZE_STANDARD;
```

CONTROL PROCEDURES

EXEC

```
4820 proc EXEC(rd new_stmts);
4821 --
4822 -- Procedure EXEC is given a tuple of statements
4823 -- which are to be executed in the order given. This is achieved by
4824 -- appending the sequence of statements to the front of the current
4825 -- statement sequence, STSQ.
4826 --
4827   STSQ := new_stmts + STSQ;
4828 end proc EXEC;
```

SYSTEM_ERROR

```
4830 proc SYSTEM_ERROR(message);
4831     TO_LIST(ERR_LINE() + 'Bad code -- ' + message);
4832     exec( [['raise', Except_unames('SYSTEM_ERROR'),
4833                 'please report to NYU']] );
4834 end proc SYSTEM_ERROR;
```

COMPUTE PROCEDURES

Setval

```
4837 proc SETVAL (object, value, drec);
4838 --
4839 -- Performs an assignment of the IVALUE in value
4840 -- to the IOBJECT in object. Note that it is not object itself which
4841 -- is modified, but the values (mapped by CONTENTS). This is
4842 -- why object is a read only argument to the procedure.
4843 --
4844 -- The third parameter, drec, is set to om, except in the case
4845 -- of a component assignment in a complete record assignment.
4846 -- In this case, drec contains the value of the complete record
4847 -- being assigned, and is used to obtain discriminant values
4848 -- which specify bounds for dynamic arrays.
4849 --
4850 -- All length checks and discriminant checks are made in SETVAL.
4851 -- If a CONSTRAINT_ERROR should result, a raise statement is appended
4852 -- to the front of the statement list and SETVAL returns the value
4853 -- false. Otherwise it returns true.
4854
4855  -- First deal with the case of a scalar object
4856
4857  if is_location (object) then
4858
4859    CONTENTS(object) := value;
4860 --
4861  -- Now some special objects
4862 --
4863  elseif not is_tuple object then
4864
4865    return false;                -- Result of a prior array length error
4866
4867  else
4868
4869    case type_mark(object) of
4870      ('task_iobject'):
4871        CONTENTS(location(object)) := value(2); --
4872      ('proc_iobject'):
4873        loop for i ∈ [2..5] do
4874          CONTENTS(object(i)) := value(i);
4875        end loop for;
4876      ('label_iobject'):
```

```
4877        CONTENTS(location(object)) := value(2..);
4878      ('access_iobject'):
4879        CONTENTS(location(object)) := value(2);
4880      ('fixed_iobject'):
4881        CONTENTS(location(object)) := value;
4882  --
4883  -- Now deal with the case of a record. Just assign the fields
4884  -- of the record value to the corresponding objects.
4885  -- But first check discriminants if necessary.
4886
4887      ('record_iobject'):
4888        [-, objdeclist, constr_disc_list] := object;
4889        [-, fieldmap] := value;
4890
4891          if present(constr_disc_list) and          -- check discriminants
4892            exists f ∈ constr_disc_list |
4893                  CONTENTS(objdeclist(f)) /= fieldmap(f) then
4894             exec( [['raise', 'CONSTRAINT_ERROR', 'discriminant']] );
4895             return false;                          -- to stop the recursion
4896          end if;
4897
4898        if 'fields_present' ∈ domain fieldmap then
4899          fields := fieldmap('fields_present');
4900        else
4901          fields := domain fieldmap;
4902        end if;
4903        loop for f ∈ fields do
4904          if not SETVAL (objdeclist(f), fieldmap(f), fieldmap) then
4905             return false;
4906           end if;
4907        end loop for;
4908
4909  -- Now we must check for the special case where the variant of
4910  -- a dynamic record has been modified and its fields_present
4911  -- indication must be updated.
4912
4913        if 'fields_present' ∈ domain objdeclist then
4914          CONTENTS(objdeclist('fields_present')) := fields;
4915        end if;
4916
4917  -- Now deal with the case of an array. First perform the length check.
4918
4919      ('array_iobject'):                          --
4920        [-, lseq, lb, ub, lbdiscr, ubdiscr, nullobj] := object;
4921        [-, rseq, l_val, u_val, nullval] := value;
4922
4923  -- First check for a case of a dynamic array being assigned in
4924  -- a complete record assignment (the fact that we have a dynamic
4925  -- array is indicated by the presence of either the lbdiscr or
```

```
4926 -- ubdiscr fields, and the complete record assignment case is
4927 -- indicated by the presence of the drec argument which gives
4928 -- the value of the complete record being assigned). If we have
4929 -- this case, then the bounds corresponding to unconstrained
4930 -- discriminants are set to their proper values (from drec).
4931
4932       if present(lbdiscr) and present(drec) then
4933         CONTENTS(lb) := drec(lbdiscr);
4934       end if;
4935
4936       if present(ubdiscr) and present(drec) then
4937         CONTENTS(ub) := drec(ubdiscr);
4938       end if;
4939
4940         lbval := CONTENTS(lb);
4941         ubval := CONTENTS(ub);
4942
4943 -- Now we perform the length check. Note that an empty array
4944 -- can be assigned to any empty object, regardless of its bounds
4945
4946       nullobj := (nullobj = 'null') or (ubval < lbval);    --
4947       nullval := (nullval = 'null') or (u_val < l_val);
4948       if nullobj and nullval then
4949         return true;
4950       elseif  -- not both empty
4951          u_val - l_val /= ubval - lbval          -- not same length
4952       then
4953          exec([['raise', 'CONSTRAINT_ERROR',
4954                          'arrays not same length']]);
4955         return false;
4956       end if;
4957
4958 -- Now the bounds are set correctly, and we simply assign the
4959 -- component values (using the right hand side length since it
4960 -- may be the case that the left side is longer in the dynamic
4961 -- array case, since it was allocated for the maximum length).
4962 -- Note that we pass down the drec parameter to take care of
4963 -- the case of a two dimensional array where the inner array
4964 -- is the dynamic one.
4965
4966       loop for i ∈ [1..#rseq] do
4967          if not SETVAL (lseq(i), rseq(i), drec) then
4968            return false;
4969          end if;
4970       end loop for;
4971
4972 -- Logic error check (one of the above cases should hold!)
4973
4974     else
```

```
4975        TO_LIST(ERR_LINE() + 'Bad code -- in assignment statement');
4976        TO_LIST('LHS = ', str object, ' RHS = ', str value);
4977          return false;
4978      end case type_mark(object);
4979
4980    end if is_location(object);
4981
4982    return true;
4983
4984  end proc SETVAL;
```

L_VALUE

```
4986  proc L_VALUE(object);
4987  --
4988  -- L_VALUE is given  an iobject  and  returns  the corresponding
4989  -- ivalue. It simply returns uninitialized when appropriate without
4990  -- signalling any error, the caller must check as required for this.
4991  --
4992    if is_location(object) then
4993      return CONTENTS(object);
4994    else
4995      case type_mark(object) of
4996
4997        ('task_iobject'):
4998          return ['task_ivalue', CONTENTS(location(object)) ];
4999        ('proc_iobject'):
5000          [-, task_loc, index_loc, formals_loc, body_loc] := object;
5001          return ['proc_ivalue', CONTENTS(task_loc),
5002                                 CONTENTS(index_loc),
5003                                 CONTENTS(formals_loc),
5004                                 CONTENTS(body_loc)     ];
5005        ('label_iobject'):
5006          return ['label_ivalue'] + CONTENTS(location(object));
5007        ('access_iobject'):
5008          return ['access_ivalue', CONTENTS(location(object)) ];
5009        ('fixed_iobject'):
5010          return CONTENTS(location(object));
5011
5012        ('array_iobject'):
5013          [-, seq, lowbound, highbound, -, -, nullobj] := object;  --
5014          seq := [L_VALUE(e) : e ∈ seq];
5015          lowbound := L_VALUE(lowbound);
5016          highbound := L_VALUE(highbound);
5017  --
5018            s := if highbound < lowbound then [ ]
5019                 elseif seq = [ ] then [ ]                          --
5020                 else seq(1..highbound - lowbound + 1) end;
5021          return ['array_ivalue', s, lowbound, highbound, nullobj];
```

```
5022
5023      ('record_iobject'):
5024        [-, objdeclist, -] := object;
5025        fields_present := domain objdeclist;
5026        if 'fields_present' ∈ fields_present then
5027          fields_present := CONTENTS(objdeclist('fields_present'));
5028        end if;
5029        return ['record_ivalue',
5030              {[f, L_VALUE(objdeclist(f))] : f ∈ fields_present }];
5031
5032      else
5033        SYSTEM_ERROR('ivalue of ' + str object);
5034      end case type_mark(object);
5035    end if is_location(object);
5036
5037 end proc L_VALUE;
```

APPLY_DISCR

```
5039 proc APPLY_DISCR (dmap, rectype);
5040 --
5041 -- This procedure applies discriminant constraints to a record
5042 -- type which may have a variant part .
5043 --
5044 -- Dmap is the ivalue for the discriminant aggregate, which
5045 -- has the form of a map from discriminants to their values.
5046 -- Rectype is the itype for the record type
5047
5048 -- The result returned is an itype for the constrained result.
5049
5050  [-, [objdeclist, variantpart], constr_disc_list, disc_list] :=
5051                                                      rectype;
5052
5053 -- If the constraint includes a dynamic discriminant, than it cannot
5054 -- be applied now. This itype is the itype of a record which is a
5055 -- component of an enclosing record, and the constraint depends on
5056 -- one or more of the discriminants of that enclosing record. Return
5057 -- a record itype marked dynamic. The constraint will be applied by
5058 -- create_obj_ when the enclosing record object is created.
5059
5060  if exists a ∈ range dmap | is_discr_ref(a) then                --
5061    return
5062    ['record', [objdeclist, variantpart],
5063              ['discr_ref', dmap], disc_list];                  --
5064  end if;
5065
5066 -- First step is to identify any discriminants which appear in the
5067 -- objdeclist and supply the values of the constraints as initial
5068 -- values. The values supplied for a discriminant override any
```

```
5069  -- initial values supplied in the record type definition.
5070
5071    loop for i ∈ [1..#objdeclist] do
5072      [field_name, [field_itype, field_vexpr]] := objdeclist(i);
5073      if field_name ∈ domain dmap then
5074        objdeclist(i) := [field_name, [field_itype,
5075                          ['ivalue', dmap(field_name)]]];
5076      end if;
5077    end loop for;
5078
5079  -- Now see if we have a variant part depending on a discriminant
5080  -- which is one of the ones being constrained. If so, then we
5081  -- select the corresponding componentlist and discard the rest.
5082  -- The objdeclist of this componentlist is then appended to the
5083  -- original objdeclist (since these now become fixed fields of
5084  -- the resulting type) and the variantpart of this componentlist
5085  -- (resulting from possible nested variant parts) replaces the
5086  -- original variant part. This test is then repeated in a loop
5087  -- to apply the constraints to the nested variant part. Note that
5088  -- this loop must terminate, since all discriminants must be given
5089  -- values in dmap, and all variants must depend on one of them.
5090
5091   loop while variantpart /= [ ] do
5092    [variant_name, altern_list] := variantpart;
5093    if variant_name ∈ domain dmap then
5094
5095     -- Find the variant to be used
5096      if (exists [choices, component_list ] ∈ altern_list
5097        | CONTAINS(choices, dmap(variant_name)))
5098      or
5099        present(component_list := altern_list(choices := {'others'}))
5100      then
5101       [vobjdeclist, variantpart] := component_list;
5102       objdeclist +:= vobjdeclist;
5103      else
5104       SYSTEM_ERROR('APPLY_DISCR');
5105      end if;
5106
5107    else
5108      SYSTEM_ERROR('missing discriminant for variant ' + variant_name);
5109      return [ ];
5110    end if;
5111   end loop while;
5112
5113
5114  -- See if we have any dynamic arrays which reference discriminants
5115  -- which are being constrained. If so, replace the index type with
5116  -- a corresponding version in which the discriminant reference has
5117  -- been replaced by its constrained value.
```

```
5118
5119  loop for [fname, [ftype, initval]] = objdeclist(i) do          --
5120
5121    [fkind, index_type_list, elementtype] := GET_ITYPE(ftype);
5122    if fkind /= 'array' then
5123      continue;
5124    end if;
5125
5126  new_index_type_list := [ ];
5127  loop while index_type_list /= [ ] do
5128     index_type fromb index_type_list;
5129    [index_typename, lower, upper, enum_map] := GET_ITYPE (index_type);
5130
5131    if is_discr_ref(lower) then                                   --
5132      [-, discrimname] := lower;
5133      if discrimname ∈ domain dmap then
5134         lower := dmap(discrimname);
5135      end if;
5136    end if;
5137
5138    if is_discr_ref(upper) then                                   --
5139      [-, discrimname] := upper;
5140      if discrimname ∈ domain dmap then
5141         upper := dmap(discrimname);
5142      end if;
5143    end if;
5144
5145    index_type := [index_typename, lower, upper, enum_map];
5146    new_index_type_list with := index_type;
5147
5148   end loop while;
5149
5150    ftype := ['array', new_index_type_list, elementtype];
5151    objdeclist(i) := [ fname, [ftype, initval] ];              --
5152
5153  end loop for;
5154
5155 -- Return result, setting the list of constrained discriminants to
5156 -- the field names in dmap (which must be all the names of the
5157 -- discriminants).
5158
5159  return ['record', [objdeclist, [ ]], domain dmap, disc_list];
5160
5161 end proc APPLY_DISCR;
```

CONTAINS

```
5163 proc CONTAINS (choicelist, value);
5164 --
5165 -- This is a general procedure used to determine whether a value
5166 -- appears within the set of values specified by choicelist. In
5167 -- fact if choicelist were represented as a set of values, then
5168 -- it would be exactly a membership test. As it is, the format
5169 -- of a choicelist is:
5170
5171 --   choicelist => {choice}
5172 --   choice => value | ['range', lbd, ubd] | 'others'
5173
5174 -- Note that everything has been evaluated, so the values are
5175 -- discrete integer values.
5176
5177 -- CONTAINS is a boolean procedure which returns true if value
5178 -- is contained in choicelist and false otherwise.
5179
5180  return exists choice ∈ choicelist |
5181    if is_range(choice) then              --
5182      expr
5183        get_bounds_of(choice);
5184        yield (lbd <= value and value <= ubd);
5185      end
5186    elseif is_literal_ivalue(choice) then
5187      value = choice
5188    elseif choice = 'others' then
5189      true
5190    else
5191      expr
5192        SYSTEM_ERROR('CONTAINS');
5193        yield false;                      --
5194      end
5195    end;
5196
5197 end proc CONTAINS;
```

GET_ITYPE

```
5199 proc GET_ITYPE (typeref);
5200 --
5201 -- This procedure is passed an itype reference and returns
5202 -- the itype obtained by dereferencing an itype name reference.
5203 -- It is essentially a conditional dereference operator for
5204 -- itype references (but it only goes down to the first non-
5205 -- named level).
5206
```

```
5207  loop while is_simple_name(typeref) do
5208    new_typeref := GLOBAL_EMAP(typeref) ? EMAP(typeref);
5209    if absent(new_typeref) then
5210      exec([['raise', 'PROGRAM_ERROR', 'Access to ' + str typeref
5211              + ' before elaboration']]);
5212      return om;
5213    end if;
5214    typeref := new_typeref;
5215  end loop while;
5216  return typeref;
5217 end proc GET_ITYPE;
```

VEVAL: Expression evaluation

```
5219 proc VEVAL_PROC(rd evalexpr);
5220 --
5221 -- Processes most of the operator forms avaliable to 'veval_' .
5222 -- Evalexpr is the expression to be evaluated. Note that forms
5223 -- generated by the interpreter in course of evalution are
5224 -- indicated by names starting with - .
5225 --
5226 const
5227    unary_ops = {'+ui', '+ufl', '+ufx', '-ui', '-ufl', '-ufx',
5228                 'absi', 'absfl', 'absfx', 'not', 'row'       }; --
5229
5230 -- Forms processed by VEVAL_PROC:
5231 -- ------------------------------
5232 --
5233  case opcode(evalexpr) of
5234 --
5235 --
5236 -- An IVALUE can appear as a constant in a VEXPR using the form:
5237 --
5238 --  ['ivalue', ivalue]
5239 --
5240 -- Here the 'ivalue' tag acts like the LISP QUOTE function and the
5241 -- result of evaluating the VEXPR is merely to return the IVALUE
5242 -- specified by the ivalue entry which is the second component.
5243 --
5244        ('ivalue') :
5245            [-, ivalue] := evalexpr;
5246            push( ivalue );
5247 --
5248 --
5249 -- Expression
5250 --
5251 --   This operator '()' has been introduced to distinguish a name
5252 --   from an expression, when the expression is reduced to a name.
5253 --
5254       ('()') :
5255           [-, evalexpr] := evalexpr;
5256           exec( [[ 'veval_', evalexpr ]] );
5257 --
5258 --
5259 -- Discriminant reference :
5260 --
5261 --   This can only legitimately appear when we are within the dummy
```

```
5262 --   scope created by create_obj_ when creating the object for a
5263 --   record with discriminants. In this case, EMAP contains the
5264 --   required discriminant value.
5265 --
5266       ('discr_ref') :
5267           [-, discr_name, discr_itype] := evalexpr; --
5268           push( EMAP(discr_name) ? GLOBAL_EMAP(discr_name)
5269                                  ? evalexpr );
```

4.1.1 Indexed Components

```
5271 --   ['[ ]', array_vexpr, subscript_vexpr]
5272 --
5273 --   The array_vexpr is a VEXPR which evaluates to the array IVALUE,
5274 --   and subscript_vexpr evaluates to an integer or enum IVALUEs which
5275 --   are the subscript values. The result is the selected component
5276 --   IVALUE.
5277 --
5278
5279       ('[ ]'):
5280         [-, array, subscript_list] := evalexpr;
5281         loop while subscript_list /= [ ] do
5282           subscript frome subscript_list; --
5283           exec ( [['veval_', subscript],
5284                   ['veval_', ['$subscript']] ]);
5285         end loop;
5286         exec( [['veval_', array]] );
5287
5288       ('$subscript'):
5289         pop( subscript );
5290         pop( array );
5291         [-, seq, lowval, highval] := array;
5292
5293         if absent(lowval) or absent(highval) then
5294           SYSTEM_ERROR('$subscript');
5295         elseif lowval = UNINITIALIZED or highval = UNINITIALIZED then
5296           SYSTEM_ERROR('$subscript');
5297         elseif subscript < lowval or subscript > highval then
5298           exec([['raise', 'CONSTRAINT_ERROR',
5299                           'out-of-range subscript']]);
5300         else
5301 --
5302           value := seq (subscript - lowval + 1);
5303           if value = UNINITIALIZED
5304             or (is_tuple value and value(2) = UNINITIALIZED) then
5305             exec([['raise', 'PROGRAM_ERROR',
5306                           'uninitialized element']]);
5307           else
5308             push( value );
```

```
5309          end if;
5310        end if;
```

4.1.2 Slices

```
5312 --  ['[..]', array_vexpr, slice_texpr]
5313 --
5314 --  The array_vexpr is a VEXPR which evaluates to the array IVALUE.
5315 --  The slice_texpr entry is a TEXPR which evaluates to an ITYPE for
5316 --  discrete type whose range specifies the slice to be taken. If
5317 --  slice_texpr is a constraint operation, then it must be an
5318 --  'index_range' constraint.
5319 --
5320
5321      ('[..]'):
5322        [-, array, slice] := evalexpr;
5323        exec ( [['veval_', array],
5324                ['teval_', slice],
5325                ['veval_', ['$slice']]] );
5326
5327      ('$slice'):
5328        pop( slice );
5329        pop( array );
5330        [-, seq, lowval, highval] := array;
5331        [-, slicelo, slicehi] := slice;
5332
5333        if absent(lowval) or absent(highval) then
5334
5335          SYSTEM_ERROR('$slice');
5336
5337        elseif lowval = UNINITIALIZED or highval = UNINITIALIZED then
5338
5339          SYSTEM_ERROR('$slice');
5340
5341        elseif slicehi < slicelo then -- null slice(special bounds check)
5342                                      -- RM 4.1.2 para. 2, 3
5343          push( ['array_ivalue', [ ], slicelo, slicehi] );
5344
5345        elseif slicelo < lowval or highval < slicehi then
5346
5347          exec ( [['raise', 'CONSTRAINT_ERROR', 'out-of-bounds']]);
5348
5349        else
5350
5351          push( ['array_ivalue',
5352            seq(slicelo - lowval + 1 .. slicehi - lowval + 1),
5353                                                  slicelo, slicehi] );
5354
5355        end if;
```

4.1.3 Selected Components (record)

```
5357 --  ['.', record_vexpr, fieldname]
5358 --
5359 --  The record_vexpr is a VEXPR which evaluates to the record IVALUE.
5360 --  The fieldname is the name (a SETL string) of the field whose
5361 --  corresponding IOBJECT will be the returned result. Note that we must
5362 --  check for the field actually existing (may have wrong variant
5363 --  present).
5364
5365      ('.'):
5366        [-, record_vexpr, fieldname] := evalexpr;
5367        exec ( [['veval_', record_vexpr],
5368                ['veval_', ['$select', fieldname]] ]);
5369
5370      ('$select'):
5371        [-, fieldname] := evalexpr;
5372        pop( record );
5373        [-, objdeclist] := record;
5374        if present(objdeclist(fieldname)) then
5375
5376          fields := objdeclist('fields_present');
5377          if present(fields) then
5378                                              -- record with
5379                                    -- unconstrained variants
5380 --
5381            if fieldname ∉ fields then
5382              exec([['raise', 'CONSTRAINT_ERROR', 'field absent']]);
5383            else
5384              value := objdeclist(fieldname); --
5385            end if;
5386
5387          else -- no variants or constrained discriminants
5388
5389            value := objdeclist(fieldname);  --
5390
5391          end if;
5392
5393        else -- fieldname absent from this object
5394
5395          exec( [['raise', 'CONSTRAINT_ERROR', 'field absent']] );
5396
5397        end if;
5398 --
5399        if value = UNINITIALIZED
5400          or (is_tuple value and value(2) = UNINITIALIZED) then
5401          exec([['raise', 'PROGRAM_ERROR', 'uninitialized field']]);
5402        else
```

```
5403          push( value );
5404        end if;
```

4.1.3 Selected components (dereference)

```
5406 --  ['@', access_vexpr]
5407 --
5408 --  The access_vexpr entry is a VEXPR which when evaluated returns
5409 --  an IVALUE of access type (or of a renamed object). The result
5410 --  returned is the IVALUE designated by the access value.
5411 --
5412        ('@'):
5413          [-, name] := evalexpr;
5414          exec( [['veval_', name],
5415                ['veval_', ['$@']]] );
5416 --
5417        ('$@'):
5418          pop( access_value );
5419          if is_access_ivalue(access_value) then    --
5420            [-, iobject] := access_value;
5421            if iobject /= NULL then
5422              value := I_VALUE( iobject );
5423              if value = UNINITIALIZED
5424                or is_uninitialized (value) then
5425                exec( [['raise', 'PROGRAM_ERROR',
5426                      'access to uninitialized object']] ); --
5427              else
5428                push( value );
5429              end if;
5430            else
5431              exec( [['raise', 'CONSTRAINT_ERROR', 'null access']] );
5432            end if;
5433          else
5434            SYSTEM_ERROR('veval (dereferencing)');
5435          end if;
```

4.3.1 Record aggregate

```
5437 --      ['record_aggregate', type_name,
5438 --                      [[field_names], element_vexpr], ....]
5439 --
5440 --      The field_names are the names of the corresponding fields,
5441 --      and the range elements are VEXPR's for the field values.
5442 --      As indicated by the structure, more than one field can map
5443 --      to the same expression value. Note that others cannot appear
5444 --      explicitly, the front end is expected to expand others as
5445 --      a (possibly empty?) list of the corresponding field names.
5446 --
5447      ('record_aggregate'):
```

```
5448        [-, type_name, field_list] := evalexpr;
5449        [-, -, -, discr_list] := GET_ITYPE(type_name);
5450        discr_list ?:= {};
5451        discr_exec := field_exec := [ ];
5452
5453 -- the discriminants are evaluated first, in case they are to
5454 -- determine the constraint applicable to some of the components
5455 -- cf. LRM 4.3.1(2)  4.3.2(9)
5456 -- To achieve this, the field_list given is first sorted:
5457
5458        loop for [field_names, field_vexpr] ∈ field_list do
5459          loop for field_name ∈ field_names do
5460             if field_name ∈ discr_list then
5461                discr_exec with:= [field_name, field_vexpr];
5462             else
5463                field_exec with:= [field_name, field_vexpr];
5464             end if;
5465          end loop for;
5466        end loop for;
5467
5468        exec([['veval_',
5469              ['$$record_aggregate', type_name, #field_exec]]]);
5470        loop for [field_name, field_vexpr] ∈ field_exec do
5471           exec( [['veval_', field_vexpr], ['push_', field_name]] );
5472        end loop;
5473
5474        if discr_exec /= [ ] then
5475          exec([['veval_', ['$record_aggregate', #discr_exec]]]);
5476          loop for [field_name, field_vexpr] ∈ discr_exec do
5477            exec( [['veval_', field_vexpr], ['push_', field_name]] );
5478          end loop;
5479        end if;
5480
5481     ('$record_aggregate'):
5482       [-, nb_discr] := evalexpr;
5483       discr_map := {};
5484       loop for i ∈ [1..nb_discr] do
5485          pop( discr_name );
5486          pop( discr_value );
5487          discr_map(discr_name) := discr_value;
5488          -- to make them available for index subtype of other components:
5489          EMAP(discr_name) := discr_value;
5490       end loop for i;
5491       push ( discr_map );
5492
5493 --
5494     ('$$record_aggregate'):
5495        [-, type_name, num_field_values] := evalexpr;
5496
```

```
5497        itype := GET_ITYPE(type_name);
5498        disc_list := itype(4) ? {};        -- And its discrims
5499
5500        field_map := {};                   -- Construct the field map
5501        loop for i ∈ [1..num_field_values] do
5502           pop( field_name );
5503           pop( field_value );
5504           field_map(field_name) := field_value; --
5505        end loop for i;
5506
5507        if nonempty(disc_list) then    -- Check if constrained
5508          pop ( discr_map );
5509          field_map +:= discr_map;
5510          itype := APPLY_DISCR(discr_map, itype);--Now it is constrained
5511        end if;
5512
5513        push( ['record_ivalue', field_map] );
5514
```

4.3.2 Array aggregate

```
5516 --     ['array_aggregate', texpr, pos_entries, named_entries, ghost]
5517 --
5518 --     pos_entries    ::=   [vexpr, vexpr, ...] | [ ]
5519 --     named_entries  ::=   [[choice_list, element_vexpr], ...] | [ ]
5520 --     choice_list    ::=   'others' | [choice, ...]
5521 --     choice         ::=   choice_vexpr | ['range', lbd, ubd]
5522 --
5523 --     The second component is a texpr for index constraint on the
5524 --     value of the aggregate. This is used for assigning index
5525 --     values to positional entries and for determining the values
5526 --     supplied by an others clause.
5527 --
5528 --     The pos_entries component is a sequence of VEXPR's which give
5529 --     values of the positional entries given, in the sequence in
5530 --     which they appear in the aggregate.
5531 --
5532 --     The named_entries component maps sets of index values, specified
5533 --     by a choice_list entry, to corresponding values, as given by
5534 --     the VEXPR element_vexpr. A choice_list entry has two forms,
5535 --     it is either the string 'others', or it is a set of index
5536 --     choices, where an index choice is either a VEXPR giving a
5537 --     particular index value, or a TEXPR for a discrete type whose
5538 --     bounds specify a range of index values.
5539 --
5540 --     The LRM 4.3.2(10) specify the following order of elaboration:
5541 --       - first all the choices (including choices in subaggregates)
5542 --       - if not a null array, check that the bounds of choices
5543 --         belong to the index subtype (constraint_error)
```

```
5544 --          - in all cases, if multidimensional array, check that all
5545 --            subaggregates have same bounds (constraint_error)
5546 --          - evaluate component expressions exactly as many times as
5547 --            required by the choices
5548
5549
5550 --  1.1 compute choices:
5551
5552 ('array_aggregate'):
5553    [-, index_type_list, pos_entries, named_entries] := evalexpr;
5554    [exec_list, new_aggr] :=
5555      aggr_step_1_1(pos_entries, named_entries, index_type_list);
5556    exec( exec_list with
5557          ['veval_', ['$array_aggregate', index_type_list, new_aggr]]);
5558
5559 --  1.2 compute size of array, and check that bounds of subaggregates
5560 --      are identical:
5561
5562 ('$array_aggregate'):
5563    [-, index_type_list, [pos_entries, named_entries]] := evalexpr;
5564    [array_size, actual_index_list, c_error] :=
5565      aggr_step_1_2(pos_entries, named_entries, index_type_list);
5566    if c_error then
5567      exec([['raise', 'CONSTRAINT_ERROR', 'incompatible bounds']]);
5568      return; -- from VEVAL_PROC
5569    end if;
5570
5571 --  1.3 if not a null array, then check that bounds of (sub)aggregates
5572 --      belong to the index_subtype:
5573 --      Note: we already know that all subaggregates have same bounds.
5574
5575    if array_size /= 0 then
5576      loop for i ∈ [1..#index_type_list] do
5577
5578        [-, aggr_low, aggr_high] := actual_index_list(i);
5579        [-, type_low, type_high] := GET_ITYPE(index_type_list(i));
5580        if is_discr_ref(type_low) then
5581          [-, d_name, -, index_subtype] := type_low;
5582          type_low := EMAP(d_name) ? GET_ITYPE(index_subtype)(2);
5583        end if;
5584        if is_discr_ref(type_high) then
5585          [-, d_name, -, index_subtype] := type_high;
5586          type_high := EMAP(d_name) ? GET_ITYPE(index_subtype)(3);
5587        end if;
5588
5589
5590        -- note: we know that aggr_low <= aggr_high
5591        --       otherwise size would have been 0
5592
```

```
5593          if aggr_low  < type_low
5594          or aggr_high > type_high then
5595            exec([['raise', 'CONSTRAINT_ERROR', 'bounds out of range']]);
5596            return; -- from VEVAL_PROC
5597          end if;
5598        end loop;
5599
5600 --  2.1 if null array, no evaluation of any component, we just build
5601 --      an appropriate 'array_ivalue', with proper bounds:
5602
5603     else  -- array_size = 0
5604       aggr := Ω;
5605       loop while actual_index_list /= [ ] do
5606          [-, low, high] frome actual_index_list;
5607          aggr := ['array_ivalue', [aggr], low, high];
5608       end loop;
5609       push (aggr with 'null');                    --
5610       return; -- from VEVAL_PROC
5611     end if;
5612
5613 --  2.2 now compute all elements:
5614
5615     exec([['veval_', ['$$array_aggregate', actual_index_list]]]);
5616     aggr_step_2_2(pos_entries, named_entries, actual_index_list);
5617
5618 --  2.3 gather components to build array_ivalue:
5619
5620 ('$$array_aggregate'):
5621     [-, actual_index_list] := evalexpr;
5622     aggr := aggr_step_2_3(actual_index_list);
5623     push ( aggr );
5624
```

4.4 Attribute

```
5626 --      ['''', attribute, .....]
5627 --
5628 --      This form handles attributes. The bulk of the processing
5629 --      is done in proc ATTRIBUTE_EVAL (q.v.).
5630 --
5631        (''''):
5632          ATTRIBUTE_EVAL(evalexpr);
```

4.5 Operators

```
5634 --      ['andthen', op1, op2]
5635 --
5636 --      The andthen form takes two parameters which are the boolean
5637 --      expressions for the operands. This case is handled specially
```

```
5638 --    because of the short circuit evaluation required.
5639 --
5640       ('andthen'):
5641         [-, op1, op2] := evalexpr;
5642         exec( [['veval_', op1],
5643               ['if_', [['veval_', op2]],
5644                     [['push_', boolean_false ]]] ]);
5645 --
5646 --    ['in', exprv, type_expr]
5647 --
5648 --    The in form is used  for an in  operator.The first parameter is
5649 --    the expression whose value is to be tested, and the second is
5650 --    an expression for the type.
5651 --
5652       ('in'):
5653 --
5654         [-, exprv, type_expr] := evalexpr;
5655         exec( [['veval_', exprv],
5656               ['teval_', type_expr],
5657               ['veval_', ['$in']] ]);
5658 --
5659       ('$in'):
5660         pop( typeentry );
5661         pop( value );
5662         case type_mark(typeentry) of
5663         ('integer', 'enum', 'float'):
5664           get_bounds_of(typeentry);
5665           push( test(value >= lbd and value <= ubd) );
5666         ('fixed'):
5667           get_bounds_of(typeentry);
5668           push( test(fix_leq(lbd, value) and fix_leq(value, ubd)) );
5669         ('array'):
5670           result := true;
5671           loop while type_mark(typeentry) = 'array' do
5672
5673             [-, index_itype, typename] := typeentry;
5674             typeentry := GET_ITYPE(typename);
5675
5676             [-, l_val, u_val] := GET_ITYPE(index_itype);
5677             [-, lseq, lbval, ubval] := value;
5678
5679 -- Now we perform the bounds check. Note that an empty array
5680 -- matches any empty array object, regardless of its bounds.
5681
5682             if (ubval >= lbval or u_val >= l_val)
5683                                                 -- not both empty
5684               and (u_val /= ubval or l_val /= lbval) then
5685                                                 -- not same bounds
5686               result := false;
```

```
5687              quit;
5688            end if;
5689            if lseq = [ ] then
5690              SYSTEM_ERROR('membership');
5691              quit;
5692            end if;
5693            value := lseq(1);
5694 --
5695          end loop while;
5696          push( test( result ) );
5697 --
5698        ('record'):
5699          [-, objdeclist] := value;
5700          [-, component_list, constr_disc_list] := typeentry;
5701          [types_objdec_list, -] := component_list;
5702          result := true;
5703          if present(constr_disc_list) then
5704 --
5705            if exists f ∈ constr_disc_list,
5706                       [d, [t, v]] ∈ types_objdec_list |
5707                    d = f and objdeclist(f) /= v(2) then
5708              result := false;
5709            end if;
5710          end if;
5711          push( test( result ) );
5712 --
5713        end case;
5714 --
5715 --
5716 --      ['notin', exprv, type_expr]
5717 --
5718 --      The notin form is  used for a notin  operator. The first
5719 --      parameter is the expression whose value is to be tested, and
5720 --      the second is an expression for the type.
5721 --
5722       ('notin'):
5723         [-, exprv, type_expr] := evalexpr;
5724         exec( [['veval_', ['not', ['in', exprv, type_expr]]]]);
5725 --
5726 --      ['orelse', op1, op2]
5727 --
5728 --      The orelse form takes two parameters which are the boolean
5729 --      expressions for the operands. This case is handled specially
5730 --      because of the short circuit evaluation required.
5731 --
5732       ('orelse'):
5733         [-, op1, op2] := evalexpr;
5734         exec( [['veval_', op1],
5735               ['if_', [['push_', boolean_true]],
```

```
5736                    [['veval_', op2]]] );
5737 --
```

4.6 Conversions

```
5739 --     ['convert', from_type, to_type, cexpr]
5740 --
5741 --     Convert is used for those scalar type conversions in the source
5742 --     that do not resolve to a range qualification check between
5743 --     subtypes. In those cases where a conversion from one numeric
5744 --     type to another is wanted, this operator must be used. This
5745 --     operator also emits a range check of the result against the
5746 --     target type.
5747 --
5748    ('convert'):
5749     [-, from_type, to_type, cexpr] := evalexpr;
5750     if from_type = to_type then
5751       exec( [['veval_', cexpr]] );
5752     else
5753       exec([['teval_', from_type],
5754             ['teval_', to_type],
5755             ['veval_', cexpr],
5756             ['veval_', ['$convert']],
5757             ['veval_', ['qual_range_']] ]);
5758     end if;
5759
5760    ('$convert'):
5761     pop( value );
5762     pop( to_type );
5763     pop( from_type );
5764     case type_mark(from_type) of
5765
5766       ('enum'):
5767         case type_mark(to_type) of
5768           ('enum'):
5769             push( value );
5770         else
5771           SYSTEM_ERROR('convert');
5772         end case type_mark(to_type);
5773
5774       ('integer'):
5775         case type_mark(to_type) of
5776
5777           ('integer'):
5778             push( value );
5779
5780           ('float'):
5781             push( float value );
5782
```

```
5783          ('fixed'):
5784            push( fix_fri( value ) );
5785
5786          else
5787            SYSTEM_ERROR('convert');
5788
5789        end case type_mark(to_type);
5790
5791      ('float'):
5792        case type_mark(to_type) of
5793
5794          ('integer'):
5795            if abs value > float ADA_MAX_INTEGER then
5796              exec( [['raise', 'NUMERIC_ERROR', 'overflow']] );
5797            else
5798              push( sign(value) * fix (abs value + 0.5) );     --
5799            end if;
5800
5801          ('float'):
5802            push( value );
5803
5804          ('fixed'):
5805            push( fix_frr(value) );
5806
5807          else
5808            SYSTEM_ERROR('convert');
5809
5810        end case type_mark(to_type);
5811
5812      ('fixed'):
5813        case type_mark(to_type) of
5814
5815          ('integer'):
5816            if (ivalue := fix_toi(value)) /= 'OVERFLOW' then
5817              push( ivalue );
5818            else
5819              exec( [['raise', 'NUMERIC_ERROR', 'overflow']] );
5820            end if;
5821
5822          ('float'):
5823            if (r_value := fix_tor(value)) /= 'OVERFLOW' then
5824              push( r_value );
5825            else
5826              exec( [['raise', 'NUMERIC_ERROR', 'overflow']] );
5827            end if;
5828
5829          ('fixed'):
5830            push( value );
5831
```

```
5832            else
5833              SYSTEM_ERROR('convert');
5834
5835            end case type_mark(to_type);
5836
5837          else
5838            SYSTEM_ERROR('convert');
5839
5840        end case type_mark(from_type);
5841        push( to_type ); -- needed by 'qual_range_'
```

4.7 Qualifiers

```
5843 --   ['qual_range', typename, value]
5844 --
5845 --   The qual_range form is used to check the value of an expression
5846 --   for inclusion in a scalar subtype. The first parameter is the name
5847 --   of the type, and the second is the expression to be qualified.
5848 --   Qual_range performs a range check appropriate to the type. Note
5849 --   that the intermediate language requires all range checks to be
5850 --   explicit. Thus the result of most operators must be appropriately
5851 --   qualified using this form.
5852 --
5853 --
5854        ('qual_range'):
5855         [-, typename, value] := evalexpr;
5856 --     In case of a qual_range enclosing a discr_ref, the evaluation is
5857 --     delayed, as it is evaluated only when building an object.
5858
5859         if is_tuple(value) and is_discr_ref(value) then
5860           push( value with typename );
5861         else
5862           exec( [['veval_', value],
5863                  ['teval_', typename],
5864                  ['veval_', ['qual_range_']]] );
5865         end if;
5866 --
5867        ('qual_range_'):
5868         pop( typedesc );
5869         value := TOP_VALSTACK;
5870         get_bounds_of(typedesc);
5871         if type_mark(typedesc) /= 'fixed' then
5872           if value < lbd or value > ubd then
5873             exec( [['raise', 'CONSTRAINT_ERROR', 'out of range']] );
5874           end if;
5875         else
5876           if fix_lss(value, lbd) or fix_gtr(value, ubd) then
5877             exec( [['raise', 'CONSTRAINT_ERROR', 'out of range']] );
5878           end if;
```

```
5879        end if;
5880 --
5881 --
5882 --    ['qual_length', typename, value]
5883 --
5884 --    The  qual_length form is used to check the length of an array
5885 --    for conformity to the length specified by an array type.The first
5886 --    parameter is the name of the type, and the second is an expression
5887 --    for the array to be qualified.
5888 --
5889       ('qual_length'):
5890         [-, typename, value] := evalexpr;
5891         exec( [['veval_', value],
5892                ['teval_', typename],
5893                ['veval_', ['qual_length_']]] );
5894
5895 --
5896       ('qual_length_'):
5897         pop( itype );
5898         [-, index_list, -] := itype;
5899         array := [TOP_VALSTACK];    --
5900         loop while index_list /= [ ] do
5901          index fromb index_list;
5902
5903         [-, l_val, u_val] := GET_ITYPE(index);
5904         [-, array, lbval, ubval, nullval] := array(1);
5905         nullval := present(nullval) or (ubval < lbval);
5906 -- At this time we cannot handle bounds given by a discriminant
5907
5908         if not (is_discr_ref(l_val) or is_discr_ref(u_val)) then
5909
5910 -- Now we perform the length check. Note that an empty array
5911 -- matches any empty array object, regardless of its bounds.
5912
5913           if nullval and (u_val < l_val) then
5914             quit; -- both empty
5915           elseif                                -- not both empty
5916              u_val - l_val /= ubval - lbval          -- not same length
5917           then
5918             exec( [['raise', 'CONSTRAINT_ERROR', 'not same length']] );
5919           end if;
5920
5921         end if;
5922       end loop;
5923
5924 --
5925 --
5926 --    ['qual_discr', typename, value]
5927 --
```

```
5928 --    The  qual_discr form is used to check a record for conformity to
5929 --    the discriminants specified by a record type. The first parameter
5930 --    is the name of the type, and the second is an expression for the
5931 --    record to be qualified.
5932 --
5933      ('qual_discr'):
5934        [-, typename, value] := evalexpr;
5935        exec( [['veval_', value],
5936               ['teval_', typename],
5937               ['veval_', ['qual_discr_']]] );
5938 --
5939      ('qual_discr_'):
5940        pop( itype );
5941        ivalue := TOP_VALSTACK;
5942        [-, objdeclist] := ivalue;
5943        [-, component_list, constr_disc_list] := itype;
5944
5945        if is_set(constr_disc_list) then
5946 -- Constrained record type, ensure that record value
5947 -- has same values for  discriminants.
5948          [types_objdec_list, -] := component_list;
5949
5950            if exists f ∈ constr_disc_list,
5951                        [d, [t, v]] ∈ types_objdec_list |
5952                    d = f and objdeclist(f) /= v(2) then
5953          exec( [['raise', 'CONSTRAINT_ERROR', 'discriminant']] );
5954         end if;
5955
5956        end if;
5957
5958 --
5959 --    ['qual_index', typename, value]
5960 --
5961 --    The  qual_index form is used to check a record for sliding array
5962 --    assignments in the initialization.  The first parameter is the
5963 --    name of the type, and the second is an expression.
5964 --
5965      ('qual_index'):
5966        [-, typename, value] := evalexpr;
5967        exec( [['veval_', value],
5968               ['teval_', typename],
5969               ['veval_', ['qual_index_']]] );
5970 --
5971      ('qual_index_'):
5972        pop( itype );
5973        ivalue := [TOP_VALSTACK]; --
5974        [-, index_list, element_itype] := itype;
5975        loop while index_list /= [ ] do
5976          index_itype fromb index_list;
```

```
5977            [-, l_val, u_val] := GET_ITYPE(index_itype);
5978            [-, ivalue, lbval, ubval] := ivalue(1);
5979 -- Note : NO, the fact that it may be a null array is irrelevant,
5980 --        the subtypes of null arrays must be identical ! PK 21-Feb-84.
5981 --
5982           if is_discr_ref(l_val) then
5983             [-, d_name, d_type] := l_val;
5984             l_val := EMAP(d_name) ? GET_ITYPE(d_type)(2);
5985           end if;
5986           if is_discr_ref(u_val) then
5987             [-, d_name, d_type] := u_val;
5988             u_val := EMAP(d_name) ? GET_ITYPE(d_type)(3);
5989           end if;
5990           if (u_val /= ubval or l_val /= lbval) then -- not same bounds
5991             exec( [['raise', 'CONSTRAINT_ERROR', 'not same bounds']] );
5992             quit;
5993             end if;
5994         end loop while;
5995 --
5996 -- [ 'qual_access', type_name, access_vexpr, actual_qual ]
5997 --
5998 --   This form is used to check the value designated by an access
5999 --   value. After dereferencing the value, it transfers control to
6000 --   one of the actual 'qual' sub-instruction : 'qual_index_',
6001 --   'qual_length_', 'qual_discr_' or 'qual_range_'.
6002 --   typename is the subtype, and access_vexpr designates the value
6003 --   to qualify.
6004 --
6005    ('qual_access'):
6006      [-, type_name, access_vexpr, actual_qual ] := evalexpr;
6007      exec( [['veval_', access_vexpr],
6008             ['veval_', ['qual_access_', type_name, actual_qual]]] );
6009 --
6010    ('qual_access_'):
6011      [-, type_name, actual_qual] := evalexpr;
6012      ivalue := TOP_VALSTACK;
6013      [-, iobject] := ivalue;
6014      if iobject /= NULL then
6015          push( ivalue );   -- duplicate TOS
6016          exec([
6017               ['veval_', ['$@']],
6018               ['teval_', type_name],
6019               ['veval_', [actual_qual]],
6020               ['pop_']                  ]);-- discard TOS
6021      end if;
6022
6023 --
6024 --  [ 'qual_sub', index_list, array_vexpr ]
6025 --
```

```
6026 --   This form is used to qualify the index ranges of an array value,
6027 --   given by array_vexpr, by a subtype, namely texpr.
6028 --   The array_ivalue is left on top of stack.
6029 --
6030   ('qual_sub'):
6031     [-, index_list, array_expr] := evalexpr;
6032     exec ( [['veval_', array_expr],
6033             ['push_', index_list],
6034             ['veval_', ['qual_sub_']]] );
6035 --
6036   ('qual_sub_'):
6037     pop ( index_list );
6038     array := [TOP_VALSTACK];   --
6039     loop while index_list /= [ ] do
6040       index fromb index_list;
6041       [-, lowb, highb] := GET_ITYPE(index);
6042       [-, array, low, high, nullval] := array(1);
6043       nullval := present(nullval) or (high < low); --
6044       if nullval then quit; end if;
6045       if ( low < lowb ) or ( high > highb ) then
6046         exec( [['raise', 'CONSTRAINT_ERROR', 'out of bounds']]);
6047       end if;
6048     end loop;
6049 --
6050
6051
6052 --   Qualifications on access types are also produced in the front end
6053 --   and are converted here to the generic qual_access.
6054
6055     ('qual_adiscr', 'qual_aindex', 'qual_alength', 'qual_arange'):
6056     [opc, type_name, access_vexpr] := evalexpr;
6057     actual_qual := 'qual_' + opc(7..) + '_';
6058
6059     exec( [['veval_', access_vexpr],
6060           ['veval_', ['qual_access_', type_name, actual_qual]] ]);
```

4.8 Allocators

```
6062 --   ['new', type_name, object_texpr, aexpr]
6063 --
6064 --   The new operation allocates an access object. Here type_name is
6065 --   the name of the access type.
6066 --   Object_texpr is a texpr giving the type of the accessed object,
6067 --   and aexpr (if present) is a vexpr that evalutes to
6068 --   a value of the type accessed.
6069 --
6070     ('new'):
6071       [-, type_name, object_texpr, aexpr] := evalexpr;
6072
```

```
6073        if present(aexpr) and aexpr /= [ ] then
6074          exec( [['teval_', object_texpr],
6075                ['veval_', aexpr],
6076                   ['veval_', ['$new']] ]);
6077          else
6078             obj_type := if is_simple_name(object_texpr) then
6079                                          object_texpr else '' end;
6080             exec([['teval_', object_texpr],
6081                  ['veval_', ['new_', type_name, obj_type]] ]);
6082          end if;
6083
6084       ('new_'):
6085          [-, type_name, obj_type] := evalexpr;
6086            [-, -, t_master, t_taskenv] :=
6087                  EMAP(type_name) ? GLOBAL_EMAP(type_name);
6088          pop( object_type );
6089          if type_mark(object_type) = 'record' then
6090            [-, -, constr_discr_list, discr_list] := object_type;
6091            object_type(3) := constr_discr_list ? discr_list;
6092          end if;
6093          push( TASKS_DECLARED );
6094          TASKS_DECLARED := {};
6095          exec ([
6096             ['create_obj_', object_type, true, t_master, t_taskenv,
6097               obj_type],
6098             ['activate_'],
6099             ['veval_', ['$$new']] ]);
6100 --
6101       ('$new'):
6102          pop( value );
6103          pop( object_type );
6104          new_loc := CREATE_COPY(value, object_type, Ω);
6105          push( ['access_ivalue', new_loc] );
6106 --
6107 --
6108       ('$$new'):
6109          pop( new_loc );
6110          pop( TASKS_DECLARED );
6111          push( ['access_ivalue', new_loc] );
```

call and raise

```
6113 --    ['call', ..... ]
6114 --
6115 --    The call form is used for a call to a function within an
6116 --    expression. It is identical to a normal call statement (q.v.),
6117 --    and is pushed onto the main statement sequence for execution.
6118 --
6119       ('call'):
```

```
6120        exec( [evalexpr] );
6121 --
6122 --    ['raise', exception_name]
6123 --
6124 --    An exception may be raised within an expression by use of this
6125 --    form. The raise is simply pushed onto the statement sequence.
6126 --
6127       ('raise'):
6128        exec( [evalexpr] );
```

others operators (4.5)

```
6130 -- Remaining forms are  operator forms  where the first  element is  the
6131 -- operator  name.  In the  case  of a  binary  operator, there are two
6132 -- parameters which  are the expressions for the left and right operands.
6133 -- In  the  unary  case,  there  is a  single  parameter which  is the
6134 -- expression for the operand. The arguments to these forms are evaluated
6135 -- and an opbinary_ or opunary_ statement is pushed onto STM as needed.
6136 --
6137       else
6138        [opname, op1, op2, type_name] := evalexpr; --
6139
6140        if opname ∈ unary_ops then
6141         exec( [['veval_', op1],
6142               ['opunary_', opname] ]);
6143        else
6144         exec( [['veval_', op1],
6145               ['veval_', op2],
6146               ['opbinary_', opname] ]);
6147        end if opname;
6148
6149       end case opcode(evalexpr);
6150 end proc VEVAL_PROC;
```

Subsidiary procedures for array aggregate evaluation (4.3.2)

aggr 1.1: evaluate non static choices

```
6153 procedure aggr_step_1_1(pos_entries, named_entries, index_type_list);
6154 --
6155 --  Step 1.1 of array aggregate evaluation:
6156 --    - evaluate all non static choices, building anonymous range
6157 --        subtype for each one;
6158 --    - returns a tuple consisting of
6159 --        . the list of choices to evaluate
6160 --        . a new aggregate, where the choices have been replaced
6161 --          by the appropriate subtype.
6162 --
6163 --  Note: there are a few rules that restrict the possibilities:
6164 --    - apart from a final 'others' choice, the rest of the aggregate
6165 --      must be either all positional or all named
6166 --    - a named association is only allowed to have a choice that is
6167 --      not static (or a choice that is a null range) if the aggregate
6168 --      includes a single component association, and this component
6169 --      association has a single choice. 'others' is static if the
6170 --      index subtype is static.
6171
6172 macro get_value(x);
6173   if is_integer(x) then x
6174   elseif x(1) = 'ivalue' then x(2)
6175   else 'error in aggr 1.1'
6176   end
6177 endm;
6178
6179   exec_list := [ ];
6180   index_type fromb index_type_list;
6181
6182 -- first try to find some non-static choice, and replace it by a subtype
6183 -- name:
6184
6185   if pos_entries = [ ]
6186     and #named_entries = 1
6187   then
6188     [[choice_list, element_vexpr]] := named_entries;
6189     if choice_list = 'others' then
6190       new_type := index_type;
6191     elseif is_simple_name(choice_list) then
6192       new_type := choice_list;
```

```
6193      elseif #choice_list = 1 then
6194        [choice] := choice_list;
6195        if is_range(choice) then
6196          [-, lbd, ubd] := choice;
6197          actual_index := ['subtype', 'INTEGER', ['range', lbd, ubd]];
6198        elseif choice(1) = 'subtype' or choice(1) = 'tname' then
6199          actual_index := choice;
6200        else
6201          actual_index :=
6202              ['subtype', 'INTEGER', ['range', choice, choice]];
6203        end if;
6204        new_type := 'aggr_index' + str(newat);
6205        exec_list with := ['type', new_type, actual_index];
6206 --
6207      else -- single choice list but more than one choice: then static
6208        new_type := index_type;
6209      end if;
6210      named_entries := [[ new_type, element_vexpr]];
6211    end if;
6212
6213
6214 --  then, if we have not exhausted all dimensions, we (recursively) go
6215 --  down one level
6216 --  also do some clean up in the named part
6217
6218    iteration := named_entries;
6219    named_entries := [ ];
6220    loop forall [choice_list, element_vexpr] ∈ iteration do
6221      if is_simple_name(choice_list) then
6222        new_choice_list := choice_list;
6223      else
6224        new_choice_list := [ ];
6225        loop for choice ∈ choice_list do
6226          if is_range(choice) then
6227            [-, lbd, ubd] := choice;
6228          else
6229            lbd := ubd := choice;
6230          end if;
6231          new_choice_list with:=
6232            ['range', get_value(lbd), get_value(ubd)];
6233        end loop;
6234      end if;
6235      if index_type_list /= [ ] then
6236        if element_vexpr(1) = 'ivalue' then
6237          -- undo multidimensional array partially folded by front-end
6238          [-, seq, low, high, nullval] := element_vexpr(2);
6239          p := n := [ ];
6240          if present(nullval) or high < low then
6241            n := [[[['range', low, high]], seq(1)]];
```

```
6242          else
6243            p := seq;
6244          end if;
6245        else      -- necessarily an array_aggregate
6246          [-, -, p, n] := element_vexpr;
6247        end if;
6248        [sub_exec_list, element_vexpr] :=
6249              aggr_step_1_1(p, n, index_type_list);
6250        exec_list +:= sub_exec_list;
6251      end if;
6252      named_entries with:= [new_choice_list, element_vexpr];
6253    end loop;
6254    if index_type_list /= [ ] then
6255      iteration := pos_entries;
6256      pos_entries := [ ];
6257      loop forall element_vexpr ∈ iteration do
6258        if element_vexpr(1) = 'ivalue' then
6259          -- too hastily folded by front-end into an 'array_ivalue'
6260          p := element_vexpr(2)(2);
6261            n := [ ];
6262        else      -- necessarily an array_aggregate
6263          [-, -, p, n] := element_vexpr;
6264        end if;
6265        [sub_exec_list, sub_aggr] :=
6266                          aggr_step_1_1(p, n, index_type_list);
6267        exec_list +:= sub_exec_list;
6268        pos_entries with:= sub_aggr;
6269      end loop;
6270    end if;
6271
6272    return [exec_list, [pos_entries, named_entries]];
6273  end procedure aggr_step_1_1;
```

aggr 1.2: compute and check bounds

```
6275  procedure aggr_step_1_2(pos_entries, named_entries, index_type_list);
6276  --
6277  --   Step 1.2 of array aggregate evaluation:
6278  --     - compute the actual structure of the aggregate
6279  --     - checks that all subaggregate have the same bounds
6280  --     - returns a triple consisting of:
6281  --         . the size of the array
6282  --         . the actual index type list
6283  --         . a flag indicating a constraint_error
6284
6285    index_type fromb index_type_list;
6286
6287  -- Go down recursively in the inner subaggregates (if any)
6288
```

```
6289   if index_type_list /= [ ] then
6290     actual_index_list := Ω;
6291     loop forall [choice_list, element_vexpr] ∈ named_entries do
6292       [p, n] := element_vexpr; -- necessarily an array_aggregate
6293       [sub_size, sub_index_list, c_error] :=
6294           aggr_step_1_2(p, n, index_type_list);
6295       actual_index_list ?:= sub_index_list; -- first occurence
6296       if c_error or (actual_index_list /= sub_index_list) then
6297         return [0, [ ], true];
6298       end if;
6299     end loop;
6300     loop forall element_vexpr ∈ pos_entries do
6301       [p, n] := element_vexpr; -- necessarily an array_aggregate
6302       [sub_size, sub_index_list, c_error] :=
6303           aggr_step_1_2(p, n, index_type_list);
6304       actual_index_list ?:= sub_index_list; -- first occurence
6305       if c_error or (actual_index_list /= sub_index_list) then
6306         return [0, [ ], true];
6307       end if;
6308     end loop;
6309   else
6310     sub_size := 1;
6311     actual_index_list := [ ];
6312   end if;
6313
6314 -- Now compute the structure of the present aggregate: the current
6315 --   actual index has the form ['range', low, high]
6316
6317   [-, low, high] := GET_ITYPE(index_type);
6318   if is_discr_ref(low) then
6319     [-, d_name, -, index_subtype] := low;
6320     low := EMAP(d_name) ? GET_ITYPE(index_subtype)(2);
6321   end if;
6322   if is_discr_ref(high) then
6323     [-, d_name, -, index_subtype] := high;
6324     high := EMAP(d_name) ? GET_ITYPE(index_subtype)(3);
6325   end if;
6326
6327   if pos_entries /= [ ] then
6328     if named_entries = [ ] then
6329       high := low - 1 + #pos_entries;
6330       -- else there must be an 'others' choice
6331     end if;
6332
6333 -- only named entries:
6334
6335   elseif named_entries /= [ ] then
6336
6337     -- detect the non static case (which is single choice)
```

```
6338      if #named_entries = 1
6339      and is_simple_name(n := named_entries(1)(1)) then
6340        [-, low, high] := GET_ITYPE(n);
6341        if is_discr_ref(low) then
6342          [-, d_name, -, index_subtype] := low;
6343          low := EMAP(d_name) ? GET_ITYPE(index_subtype)(2);
6344        end if;
6345        if is_discr_ref(high) then
6346          [-, d_name, -, index_subtype] := high;
6347          high := EMAP(d_name) ? GET_ITYPE(index_subtype)(3);
6348        end if;
6349
6350      -- purely static, but may have an 'others' choice
6351      elseif exists [u, -] ∈ named_entries | u = 'others' then
6352        pass; -- [-, low, high] := GET_ITYPE(index_type);
6353
6354      -- purely static, no others choice
6355      else
6356        high := low := Ω;
6357        loop forall [choice_list, -] ∈ named_entries do
6358          loop forall choice ∈ choice_list do
6359            [-, lbd, ubd] := choice; -- only 'range' left
6360            high ?:= ubd;
6361            high max:= ubd;
6362            low ?:= lbd;
6363            low min:= lbd;
6364          end loop;
6365        end loop;
6366        present_index := ['range', low, high];
6367      end if;
6368
6369  -- no named entries and no positional entries : null string
6370    else
6371      if low = ADA_MIN_INTEGER then -- careful !
6372        low +:= 1;
6373      end if;
6374      high := low - 1;
6375    end if;
6376
6377  -- Now that we have the current index subtype and the size of a
6378  -- subaggregate, we can compute the size:
6379     size := sub_size * ( 0 max (high - low + 1) );
6380     actual_index_list := [['range', low, high]] + actual_index_list;
6381
6382     return [size, actual_index_list, false];
6383
6384  end procedure aggr_step_1_2;
```

aggr 2.2: evaluate components

```
6386 procedure aggr_step_2_2(pos_entries, named_entries, actual_index_list);
6387 --
6388 --  Step 2.2 of array aggregate evaluation:
6389 --    - evaluation of the components, actual_index_list being used
6390 --      to determine how many are missing in the case of an 'others'
6391 --      choice.
6392
6393   [-, low, high] fromb actual_index_list;
6394   low_index := low;
6395   mapping := {};
6396   loop for element_vexpr ∈ pos_entries do
6397     mapping(low_index) := element_vexpr;
6398     low_index +:= 1;
6399   end loop;
6400   loop for [choice_list, element_vexpr] ∈ named_entries do
6401     if is_simple_name(choice_list) then
6402       -- this covers both the non static range and the 'others' choice
6403       loop forall i ∈ [low_index..high]| absent(mapping(i)) do
6404           mapping(i) := element_vexpr;
6405       end loop;
6406     else
6407       loop for choice ∈ choice_list do
6408         [-, lbd, ubd] := choice;
6409         loop for i ∈ [lbd..ubd] do
6410           mapping(i) := element_vexpr;
6411         end loop;
6412       end loop;
6413     end if;
6414   end loop;
6415
6416 --  exact sequence of evaluation:
6417
6418   if actual_index_list = [ ] then
6419     exec( [['veval_', mapping(i)] : i ∈ [low..high] ] );
6420   else
6421     loop for i ∈ [low..high] do
6422       [p, n] := mapping(i);
6423       aggr_step_2_2(p, n, actual_index_list);
6424     end loop;
6425   end if;
6426
6427 end procedure aggr_step_2_2;
```

aggr 2.3: build array_ivalue

```
6429 procedure aggr_step_2_3(actual_index_list);
6430 --
6431 --  Step 2.3 of array aggregate evaluation:
6432 --    - build and return the array_ivalue whose structure is given
6433 --      by the actual index list and whose components are on the
6434 --      stack
6435
6436   [-, low, high] fromb actual_index_list;
6437   if actual_index_list /= [ ] then
6438     aggr := [ ];
6439     loop for i ∈ [low..high] do
6440       aggr with:= aggr_step_2_3(actual_index_list);
6441     end loop;
6442   else
6443     aggr := VALSTACK(#VALSTACK - high + low ..);
6444     VALSTACK(#VALSTACK - high + low ..) := [ ];
6445   end if;
6446   return ['array_ivalue', aggr, low, high];
6447
6448 end procedure aggr_step_2_3;
```

ATTRIBUTE_EVAL

```
6450 proc ATTRIBUTE_EVAL(attrib_expr);
6451 --
6452 -- Processes non-static attribute evaluation. Attrib_expr is
6453 -- a tuple passed by 'veval_' consisting of'''' and the name
6454 -- of the attribute followed by its arguments. The value of the
6455 -- attribute is left on top of VALSTACK.
6456 --
6457 -- The following attributes are static and are evaluated in the
6458 -- front end:
6459 --   base, machine_rounds, machine_radix, machine_emax, machine_emin,
6460 --   machine_overflows.
6461 -- Other attributes are also evaluated statically in the front-end
6462 -- but nevertheless appear here for completeness.
6463 --
6464 --
6465  macro fix_tenth; [1, 10] endm;
6466  macro fix_10; fix_fri(10) endm;
6467  macro fix_1; fix_fri(1) endm;
6468  macro fix_2; fix_fri(2) endm;
6469  macro fx_max;
6470     if fix_geq(fix_abs(ubd), fix_abs(lbd)) then
6471       fix_abs(ubd)
6472     else
6473       fix_abs(lbd)
6474     end
6475   endm;
6476  macro fx_small;
6477    fix_exp(fix_2,
6478      floor(log(fix_tor(delta)) / log(2.0)))
6479  endm;
6480  macro fx_mantissa;
6481    ceil(log(fix_tor(fix_div(fx_max, fx_small))) / log(2.0))
6482  endm;
6483  macro fl_mantissa;
6484     ceil((float(digits) * (log 10.0))/(log 2.0) + 1.0)
6485  endm;
6486  macro fl_emax;
6487     (4 * fl_mantissa)
6488  endm;
6489 --
6490 --
6491  case LOWER_CASE_OF(attrib_expr(2)) of
6492 --
```

```
6493 -- ["'", 'address', object_expr]
6494 --
6495 -- An address is a value of the type SYSTEM.ADDRESS defined as
6496 --      type ADDRESS is new INTEGER;
6497 -- The address of an object is:
6498 --     - for a scalar, a access object, a task, a subprogram, or a label
6499 --       some unique integer associated once for all with that entity.
6500 --     - for a non null record, the address is that of the
6501 --         first component.
6502 --     - for a null record, the address is 0.
6503 --     - for an array, the address is that of the lower bound
6504 --     - for a package or an entry, it raises SYSTEM_ERROR
6505
6506    ('address'):
6507      [-, -, object_expr] := attrib_expr;
6508      exec( [['oeval_', object_expr],
6509            ['veval_', ["'", 'address_']] ]);
6510
6511    ('address_'):
6512     pop(iobject);
6513     if not is_atom(iobject) then
6514       if is_tuple(iobject) then
6515         case type_mark(iobject) of
6516         ('fixed_iobject', 'label_iobject',
6517          'task_iobject', 'access_iobject'):
6518           [-, iobject] := iobject;
6519         ('array_iobject'):
6520           [-, -, iobject] := iobject;
6521         ('record_iobject'):
6522           [-, objdeclist] := iobject;
6523           if empty(objdeclist) then
6524             push(0);
6525           else
6526             iobject from objdeclist;
6527             push(iobject(2));
6528             exec([['veval_', ["'", 'address_']]]);
6529           end if;
6530           return;
6531         ('proc_iobject'):
6532           [-, -, -, -, iobject] := iobject;
6533         else
6534           SYSTEM_ERROR('no address for:' + str(iobject));
6535         end case;
6536       else
6537         SYSTEM_ERROR('no address for:' + str(iobject));
6538       end if;
6539
6540     end if;
6541
```

```
6542    -- here iobject is a setl atom, that we convert into an integer;
6543
6544        s := str(iobject);
6545        s := rbreak(s, '#');
6546        v := 0;
6547        loop while s /= '' do
6548            c := len(s, 1);
6549            v := 10*v + abs(c) - abs('0');
6550        end loop;
6551
6552        push(v);
6553
6554 --
6555 -- ["'", 'size', type_expr]
6556 --
6557 -- Type_expr is ignored and 'size' always returns 0, since 'size'
6558 -- is not meaningful in this implimentation.
6559 --
6560     ('size'):
6561         push( 0 );
6562 --
6563 -- ["'", 't_first', type_expr]
6564 --
6565 -- Type_expr evaluates to an ITYPE. For t'first where t is a scalar object
6566 -- the front end supplies the appropriate type expression. For t'first(i)
6567 -- where t is an array object or type, the front end supplies an
6568 -- expression for the appropriate index type.
6569 --
6570     ('t_first'):
6571         [-, -, type_expr] := attrib_expr;
6572         exec( [['teval_', type_expr],
6573                 ['veval_', ['', 't_first_']] ]);
6574 --
6575     ('t_first_'):
6576         pop( itype );
6577         get_bounds_of(itype);
6578         push( lbd );
6579 --
6580 -- ["'", 't_last', type_expr]
6581 --
6582 -- See comments above for 't_first'.
6583 --
6584     ('t_last'):
6585         [-, -, type_expr] := attrib_expr;
6586         exec( [['teval_', type_expr],
6587                 ['veval_', ['', 't_last_']] ]);
6588 --
6589     ('t_last_'):
6590         pop( itype );
```

```
6591      get_bounds_of(itype);
6592      push( ubd );
6593 --
6594 --
6595 -- ["'", 'o_first', object_expr, index_value]
6596 --
6597 -- Object_expr evaluates to an array IOBJECT. Index_value is an integer
6598 -- ordinal specifing an index.
6599 --
6600    ('o_first'):
6601      [-, -, object_expr, index_value] := attrib_expr;
6602      exec( [['oeval_', object_expr],
6603            ['veval_', index_value],
6604            ['veval_', ["'", 'o_first_']] ]);
6605 --
6606    ('o_first_'):
6607      pop( index_value );
6608      pop( obj );
6609      [tag, seq, lb, ub] := obj;
6610      loop for i ∈ [2..index_value] do
6611        if seq = [ ] then              --
6612          SYSTEM_ERROR('o_first');
6613          quit;
6614        end if;
6615        [-, seq, lb, ub] := seq(1);
6616      end loop for;
6617      push( CONTENTS(lb) );
6618 --
6619 -- ["'", 'o_last', object_expr, index_value]
6620 --
6621 -- See comments above for 'o_first'.
6622 --
6623    ('o_last'):
6624      [-, -, object_expr, index_value] := attrib_expr;
6625      exec( [['oeval_', object_expr],
6626            ['veval_', index_value],
6627            ['veval_', ["'", 'o_last_']] ]);
6628 --
6629    ('o_last_'):
6630      pop( index_value );
6631      pop( obj );
6632      [tag, seq, lb, ub] := obj;
6633      loop for i ∈ [2..index_value] do
6634        if seq = [ ] then              --
6635          SYSTEM_ERROR('o_last');
6636          quit;
6637        end if;
6638        [-, seq, lb, ub] := seq(1);
6639      end loop for;
```

```
6640      push( CONTENTS(ub) );
6641 --
6642 -- ["'", 'image', type_expr, val_expr]
6643 --
6644 -- Type_expr evaluates to an ITYPE, and val_expr to a value of the same
6645 -- (scalar) type.
6646 --
6647    ('image'):
6648      [-, -, type_expr, val_expr] := attrib_expr;
6649      exec( [['teval_', type_expr],
6650             ['veval_', val_expr],
6651             ['veval_', ["'", 'image_']] ]);
6652 --
6653    ('image_'):
6654      pop( ivalue );
6655      pop( itype );
6656      case type_mark(itype) of
6657        ('integer'):
6658          s := str ivalue;
6659          if ivalue >= 0 then --
6660            s := ' ' + s;
6661          end if;
6662        ('enum'):
6663          [-, -, -, enum_map] := itype;
6664          s := ENUM_LIT_OF(enum_map, ivalue);
6665        else
6666          SYSTEM_ERROR('image');
6667        end case;
6668        push( ['array_ivalue', [abs c : c ∈ s], 1, #s] );
6669 --
6670 -- ["'", 'value', type_expr, string_expr]
6671 --
6672 -- Type_name is a scalar ITYPE, string_expr to a value of
6673 -- type string.
6674 --
6675    ('value'):
6676      [-, -, type_name, string_expr] := attrib_expr;
6677      exec( [['push_', type_name],
6678             ['veval_', string_expr],
6679             ['veval_', ["'", 'value_']] ]);
6680 --
6681    ('value_'):
6682      pop( string_val );
6683      pop( itype_name );
6684      itype := GET_ITYPE(itype_name);
6685      [-, seq, l_val, u_val] := string_val;
6686      setl_string := +/[char seq(i) : i ∈ [l_val..u_val]];
6687      leading_blanks := span(setl_string, ' ');      --
6688      trailing_blanks := rspan(setl_string, ' ');
```

```
6689      case type_mark(itype) of
6690        ('integer'):
6691          value := ADAVAL('integer', setl_string);
6692          if value /= 'OVERFLOW' then
6693            exec( [['veval_', ['qual_range', itype_name, value]] ]);
6694          else
6695            exec( [['raise', 'CONSTRAINT_ERROR', 'invalid integer'] ]);
6696          end if;
6697 --
6698        ('enum'):
6699          [-, -, -, enum_map] := itype;
6700          value := ENUM_ORD_OF(enum_map, setl_string);
6701          if present(value) then
6702            exec( [['veval_', ['qual_range', itype_name, value]] ]);
6703          else
6704            exec( [['raise', 'CONSTRAINT_ERROR', 'invalid enum.id'] ]);
6705          end if;
6706        else
6707          SYSTEM_ERROR('value_');
6708        end case;
6709 --
6710 -- ["'", 'pos', type_expr, val_expr]
6711 --
6712 -- Type_expr evaluates to a discrete ITYPE and val_expr to a value of
6713 -- that type.
6714 --
6715     ('pos'):
6716       [-, -, type_expr, val_expr] := attrib_expr;
6717       exec( [['teval_', type_expr],
6718              ['veval_', val_expr],
6719              ['veval_', ["'", 'pos_']] ]);
6720 --
6721     ('pos_'):
6722       pop( ivalue );
6723       pop( itype );
6724       get_bounds_of(itype);
6725       if lbd <= ivalue and ivalue <= ubd then
6726         case type_mark(itype) of
6727           ('enum'):
6728             push( ivalue - lbd );
6729           ('integer'):
6730             push( ivalue );
6731           else
6732             SYSTEM_ERROR('pos_');
6733         end case;
6734       else
6735         exec( [['raise', 'CONSTRAINT_ERROR', 'out of range'] ]);
6736       end if lbd;
6737 --
```

```
6738 -- ["'", 'val', type_expr, pos_expr]
6739 --    .
6740 -- Type_expr evaluates to a discrete ITYPE and pos_expr to a value of
6741 -- universal integer type.
6742 --
6743     ('val'):
6744       [-, -, type_expr, pos_expr] := attrib_expr;
6745       exec( [['teval_', type_expr],
6746              ['veval_', pos_expr],
6747              ['veval_', ["'", 'val_']] ]);
6748 --
6749     ('val_'):
6750       pop( ivalue );
6751       pop( itype );
6752       get_bounds_of(itype);
6753       if lbd <= ivalue and ivalue <= ubd then
6754         case type_mark(itype) of
6755           ('enum'):
6756             push( ivalue + lbd );
6757           ('integer'):
6758             push( ivalue );
6759           else
6760             SYSTEM_ERROR('val_');
6761         end case;
6762       else
6763         exec( [['raise', 'CONSTRAINT_ERROR', 'out of range']] );
6764       end if;
6765 --
6766 -- ["'", 'pred', type_expr, val_expr]
6767 --
6768 -- Type_expr evaluates to a discrete ITYPE and val_expr to a value
6769 -- of that type.
6770 --
6771     ('pred'):
6772       [-, -, type_expr, val_expr] := attrib_expr;
6773       exec( [['teval_', type_expr],
6774              ['veval_', val_expr],
6775              ['veval_', ["'", 'pred_']] ]);
6776 --
6777     ('pred_'):
6778       pop( ivalue );
6779       pop( itype );
6780       get_bounds_of(itype);
6781       if lbd < ivalue and ivalue <= ubd then
6782         push( ivalue - 1 );
6783       else
6784         exec( [['raise', 'CONSTRAINT_ERROR', 'out of range']] );--
6785       end if;
6786 --
```

```
6787 -- ["'", 'succ', type_expr, val_expr]
6788 --
6789 -- Type_expr evaluates to a discrete ITYPE and val_expr to a value
6790 -- of that type.
6791 --
6792     ('succ'):
6793       [-, -, type_expr, val_expr] := attrib_expr;
6794       exec( [['teval_', type_expr],
6795              ['veval_', val_expr],
6796              ['veval_', ["'", 'succ_']] ]);
6797 --
6798     ('succ_'):
6799       pop( ivalue );
6800       pop( itype );
6801       get_bounds_of(itype);
6802       if lbd <= ivalue and ivalue < ubd then
6803         push( ivalue + 1 );
6804       else
6805         exec( [['raise', 'CONSTRAINT_ERROR', 'out of range']] );
6806       end if;
6807 --
6808 -- ["'", 'aft', type_expr]
6809 --
6810     ('aft'):
6811       [-, -, type_expr] := attrib_expr;
6812       exec( [['teval_', type_expr],
6813              ['veval_', ["'", 'aft_']]] );
6814 --
6815     ('aft_'):
6816       pop( itype );
6817       get_bounds_of(itype);
6818       aft := 1;
6819       loop while fix_lss(delta, fix_tenth) do --
6820         aft +:= 1;
6821         delta := fix_mul(delta, fix_10); --
6822       end loop while;
6823
6824       push( aft );
6825
6826 --
6827 -- ["'", 'digits', type_expr]
6828 --
6829     ('digits'):
6830       [-, -, type_expr] := attrib_expr;
6831       exec( [['teval_', type_expr],
6832              ['veval_', ["'", 'digits_']] ]);
6833 --
6834     ('digits_'):
6835       pop( itype );
```

```
6836        get_bounds_of(itype);
6837        push( digits );
6838 --
6839 -- ["'", 'delta', type_expr]
6840 --
6841      ('delta'):
6842        [-, -, type_expr] := attrib_expr;
6843        exec( [['teval_', type_expr],
6844              ['veval_', ["'", 'delta_']] ]);
6845 --
6846      ('delta_'):
6847        pop( itype );
6848        get_bounds_of(itype);
6849        push( rat_tor(delta, ada_max_digits) );
6850 --
6851 -- ["'", 'fore', type_expr]
6852 --
6853      ('fore'):
6854        [-, -, type_expr] := attrib_expr;
6855        exec( [['teval_', type_expr],
6856              ['veval_', ["'", 'fore_']] ]);
6857 --
6858      ('fore_'):
6859        pop( itype );
6860        get_bounds_of(itype);
6861        max_val := (abs fix_tor(lbd))
6862             max (abs fix_tor(ubd));
6863        push( floor (log max_val / log 10.0) + 2 );
6864 --
6865 -- ["'", 'width', type_expr]
6866 --
6867      ('width'):
6868        [-, -, type_expr] := attrib_expr;
6869        exec( [['teval_', type_expr],
6870              ['veval_', ["'", 'width_']] ]);
6871
6872 --
6873      ('width_'):
6874          pop( itype );
6875          if type_mark(itype) = 'integer' then
6876               [-, lbd, ubd, -] := itype;
6877                  max_val := abs lbd max abs ubd;
6878                  push( # str max_val + 1 );
6879          else
6880               [-, -, -, enum_map] := itype;
6881                  -- Must find longest name in enumeration type.
6882                  push( max/[ #lit: lit ∈ range enum_map] );
6883          end if;
6884
```

```
6885 --
6886 -- ["'', 't_length', type_expr]
6887 --
6888 -- Type_expr evaluates to the ITYPE of an array index.
6889 --
6890    ('t_length'):
6891      [-, -, type_expr] := attrib_expr;
6892      exec( [['teval_', type_expr],
6893            ['veval_', ["", 't_length_']] ]);
6894 --
6895    ('t_length_'):
6896      pop( itype );
6897      get_bounds_of(itype);
6898      push(( ubd - lbd + 1) max 0 );
6899 --
6900 --
6901 -- ["'', 'o_length', object_expr, index_value]
6902 --
6903 -- Object_expr evaluates to an array IOBJECT. Index_value is the
6904 -- integer ordinal of an index.
6905 --
6906    ('o_length'):
6907      [-, -, object_expr, index_value] := attrib_expr;
6908      exec( [['oeval_', object_expr],
6909            ['veval_', index_value],
6910            ['veval_', ["", 'o_length_']] ]);
6911 --
6912    ('o_length_'):
6913      pop( index_value );
6914      pop( obj );
6915      [tag, seq, lb, ub] := obj;
6916      loop for i ∈ [2..index_value] do
6917        if seq = [ ] then                --
6918          SYSTEM_ERROR('o_length');
6919          quit;
6920        end if;
6921        [-, seq, lb, ub] := seq(1);
6922      end loop for;
6923      lbd := CONTENTS(lb);
6924      ubd := CONTENTS(ub);
6925      push( (ubd - lbd + 1) max 0 );
6926 --
6927 -- ["'', 't_range', type_expr]
6928 --
6929 -- Type_expr evaluates to the scalar ITYPE of an array index.
6930 --
6931    ('t_range'):
6932      [-, -, type_expr] := attrib_expr;
6933      exec( [['teval_', type_expr],
```

```
6934            ['veval_', ['"', 't_range_']] ]);
6935 --
6936    ('t_range_'):
6937      pop( itype );
6938      get_bounds_of(itype);
6939      push( ['integer', lbd, ubd] );
6940 --
6941 -- ['"', 'o_range', object_expr, index_value]
6942 --
6943 -- Object_expr evaluates to an array object. Index_value is an integer
6944 -- ordinal of an array index.
6945 --
6946    ('o_range'):
6947      [-, -, object_expr, index_value] := attrib_expr;
6948      exec( [['oeval_', object_expr],
6949            ['veval_', index_value],
6950            ['veval_', ['"', 'o_range_']] ]);
6951 --
6952    ('o_range_'):
6953      pop( index_value );
6954      pop( obj );
6955      [tag, seq, lb, ub] := obj;
6956      loop for i ∈ [2..index_value] do
6957        if seq = [ ] then              --
6958          SYSTEM_ERROR('o_range');
6959          quit;
6960        end if;
6961        [-, seq, lb, ub] := seq(1);
6962      end loop for;
6963      push( ['integer', CONTENTS(lb), CONTENTS(ub)] );
6964 --
6965 -- ['"', 'o_constrained', obj_expr]
6966 --
6967 -- obj_expr evaluates to a record IOBJECT.
6968 --
6969    ('o_constrained'):
6970      [-, -, obj_expr] := attrib_expr;
6971      exec( [['oeval_', obj_expr],
6972            ['veval_', ['"', 'o_constrained_']] ]);
6973 --
6974    ('o_constrained_'):
6975      pop( iobj );
6976      [-, -, constr_disc_list] := iobj;
6977      push( test( present(constr_disc_list) ) );
6978
6979 -- ['"', 't_constrained', type_expr]
6980 --
6981 -- type_expr evaluates to a record Itype
6982 --
```

```
6983    ('t_constrained'):
6984      [-, -, type_expr] := attrib_expr;
6985      exec( [['teval_', type_expr],
6986            ['veval_', ["'", 't_constrained_']] ]);
6987 --
6988    ('t_constrained_'):
6989      pop( itype );
6990      [-, -, constr_disc_list] := itype;
6991      push( test( present(constr_disc_list) ) );
6992
6993 --
6994 -- ["'", 'position', record_component]
6995 -- ["'", 'first_bit', record_component]
6996 -- ["'", 'last_bit', record_component]
6997 --
6998 -- These attributes are meaningless in this implimentation, so
6999 -- an arbitrary value of 0 is returned for them. The argument
7000 -- record_component is ignored.
7001 --
7002    ('position', 'first_bit', 'last_bit'):
7003      push( 0 );
7004 --
7005 -- ["'", 'storage_size', type_expr]
7006 --
7007 -- This attribute is relatively meaningless in this implimentation,
7008 -- since the limit is effectively the size of the virtual name space.
7009 -- The largest ada integer is returned as its value.
7010 --
7011    ('storage_size'):
7012      push( ADA_MAX_INTEGER );
7013 --
7014 -- ["'", 'terminated', task_expr]
7015 --
7016 -- Task_expr evaluates to a task_ivalue.
7017 --
7018    ('terminated'):
7019      [-, -, task_expr] := attrib_expr;
7020      exec( [['veval_', task_expr],
7021            ['veval_', ["'", 'terminated_']] ]);
7022 --
7023    ('terminated_'):
7024      pop( task );
7025      [tag, taskid] := task;
7026      push( test( taskid ∈ TERMINATED_TASKS ) );
7027 --
7028 -- ["'", 'count', entry_expr]
7029 --
7030 -- Entry_expr evaluates to an entryid.
7031 --
```

```
7032    ('count'):
7033      [-, -, entry_expr] := attrib_expr;
7034      exec( [['veval_', ["'", 'count_']] ]);
7035      push_entryid(entry_expr);
7036 --
7037     ('count_'):
7038       pop( index );
7039       pop( entry );
7040       entryid := [entry, index];
7041       disable;
7042       push( #(WAITING_TASKS(CURTASK)(entryid) ? {}) );
7043       enable;
7044
7045 -- ["'", 'callable', task_expr]
7046 --
7047 -- Task_expr evaluates to a task_ivalue.
7048 --
7049    ('callable'):
7050      [-, -, task_expr] := attrib_expr;
7051      exec( [['veval_', task_expr],
7052            ['veval_', ["'", 'callable_']] ]);
7053 --
7054    ('callable_'):
7055      pop( task );
7056      [tag, taskid] := task;
7057      push(test (taskid ∉ TERMINATED_TASKS + COMPLETED_TASKS
7058                          + ABNORMAL_TASKS) ); --
7059 --
7060 -- ["'", 'emax', type_expr]
7061 --
7062   ('emax'):
7063     [-, -, type_expr] := attrib_expr;
7064     exec( [['teval_', type_expr],
7065           ['veval_', ["'", 'emax_']] ]);
7066 --
7067   ('emax_'):
7068     pop( itype );
7069     get_bounds_of(itype);
7070     push (fl_emax);
7071 --
7072 -- ["'", 'small', type_expr]
7073 --
7074   ('small'):
7075     [-, -, type_expr] := attrib_expr;
7076     exec( [['teval_', type_expr],
7077           ['veval_', ["'", 'small_']] ]);
7078 --
7079   ('small_'):
7080     pop( itype );
```

```
7081      get_bounds_of(itype);
7082      case type_mark(itype) of
7083      ('fixed'):
7084        push( fix_tor(fx_small) );        --
7085      ('float'):
7086        push( fix_tor(fix_exp(fix_2, -(fl_emax + 1))) );     --
7087      end case;
7088 --
7089 -- ["'", 'mantissa', type_expr]
7090 --
7091    ('mantissa'):
7092      [-, -, type_expr] := attrib_expr;
7093      exec( [['teval_', type_expr],
7094            ['veval_', ["'", 'mantissa_']] ]);
7095 --
7096    ('mantissa_'):
7097      pop( itype );
7098      get_bounds_of(itype);
7099      case type_mark(itype) of
7100      ('fixed'):
7101         push( fx_mantissa );
7102      ('float'):
7103         push( fl_mantissa );
7104      end case;
7105 --
7106 -- ["'", 'large', type_expr]
7107 --
7108    ('large'):
7109      [-, -, type_expr] := attrib_expr;
7110      exec( [['teval_', type_expr],
7111            ['veval_', ["'", 'large_']] ]);
7112 --
7113    ('large_'):
7114      pop( itype );
7115      get_bounds_of(itype);
7116      case type_mark(itype) of
7117      ('fixed'):
7118        push( rat_tor( fix_mul(
7119              fix_sub(fix_exp(fix_2, fx_mantissa), fix_1),
7120             fx_small), ada_max_digits)   );
7121      ('float'):
7122        push( rat_tor( fix_mul (
7123            fix_exp(fix_2, fl_emax),
7124            fix_sub(fix_1, fix_exp(fix_2, -fl_mantissa))),
7125                    ada_max_digits)  );
7126      else
7127        SYSTEM_ERROR( 'large_' );
7128      end case;
7129 --
```

```
7130 -- ["'", 'epsilon', type_expr]
7131 --
7132    ('epsilon'):
7133      [-, -, type_expr] := attrib_expr;
7134      exec( [['teval_', type_expr],
7135            ['veval_', ["'", 'epsilon_']] ]);
7136 --
7137    ('epsilon_'):
7138      pop( itype );
7139      get_bounds_of(itype);
7140      push( rat_tor(fix_exp(fix_2, 1 - fl_mantissa),
7141                    ada_max_digits) );
7142 --
7143 -- ["'", 'clock', task_expr]
7144 --
7145 -- This is an implementation defined attribute. Task_expr evaluates to
7146 -- a task_ivalue. The value returned is the integer number of milliseconds
7147 -- the named task has been executing.
7148 --
7149    ('clock'):
7150      [-, -, task_expr] := attrib_expr;
7151      exec( [['veval_', task_expr],
7152            ['veval_', ["'", 'clock_']] ]);
7153 --
7154    ('clock_'):
7155      pop( [tag, taskid] );
7156      push( TASK_CLOCKS(taskid)(1) );
7157 --
7158 --
7159 --
7160    else
7161      SYSTEM_ERROR('unknown attribute ' + attrib_expr(2));
7162  end case;
7163 --
7164  drop fix_1, fix_2, fl_emax, fx_small, fx_max, fx_mantissa,
7165       fl_mantissa, fix_10, fix_tenth;
7166 --
7167 end proc ATTRIBUTE_EVAL;
```

Enum_lit_of

```
7169 proc ENUM_LIT_OF(enum_map, pos);
7170  return enum_map(pos) ? '';
7171 end proc ENUM_LIT_OF;
```

Enum_ord_of

```
7173 proc ENUM_ORD_OF(enum_map, val_ex);
7174    v := if val_ex(1) = "" then val_ex
7175         else UPPER_CASE_OF(val_ex) end; --
7176    if exists y = enum_map(i) | y = v then
7177      return i;
7178    else
7179      return om;
7180    end if;
7181 end proc ENUM_ORD_OF;
```

OEVAL: Object evaluation (Chap.4)

```
7183 proc OEVAL_PROC(eval_action);
7184 --
7185 --
7186 -- This procedure is use to process most of the 'oeval_' instruction,
7187 -- which is used to evaluate OEXPR's to
7188 -- produce the resulting IOBJECT. These instructions are
7189 -- never present explicitly in the generated intermediate
7190 -- code, but are generated dynamically as an expression
7191 -- is evaluated. The single parameter is the next action
7192 -- to be performed. General possibilities are subexpressions
7193 -- to be evaluated and pushed onto VALSTACK, or actions which
7194 -- process VALSTACK entries.
7195
7196 -- Essentially the steps taken by this procedure correspond
7197 -- to the steps in a conventional code generator. Only at
7198 -- the bottom level are actions performed which correspond
7199 -- to normal run time actions.
7200
7201 -- Control is initially received when an OEXPR is encountered
7202 -- which must be evaluated, and the initial step is to generate
7203 -- an 'oeval_' instruction with the expression as its parameter.
7204
```

4.1 Names

```
7206     if is_simple_name(eval_action) then
7207 --
7208 -- Named Object
7209 --
7210 --   A string name appearing in a context requiring an OEXPR is
7211 --   taken to be a domain entry in EMAP, whose corresponding
7212 --   range element is the referenced object. The reaction is simply
7213 --   to get the EMAP value (which must be the desired IOBJECT) and
7214 --   stack it.
7215
7216      object := GLOBAL_EMAP(eval_action) ? EMAP(eval_action); --
7217
7218      if absent(object) then
7219         exec([['raise', 'PROGRAM_ERROR',--
7220           'Access to ' + str(eval_action) + ' before elaboration']]);
7221      else
7222        push( object );
7223      end if;
```

```
7224 --
7225     elseif not is_operation(eval_action) then
7226 --
7227       SYSTEM_ERROR('oeval :' + str eval_action);
7228
7229  -- Otherwise we have an operation which is uniquely identified
7230  -- (as usual) by its opcode (first element of the tuple).
7231
7232     else
7233       case opcode(eval_action) of
7234 --
7235 -- Quoted IOBJECT
7236 --
7237 --   ['iobject', oexpr]
7238 --
7239          ('iobject'):
7240            [-, oexpr] := eval_action;
7241            push( oexpr );
7242 -- S+ 4.1.1 Indexed components
7243 --   ['[ ]', array_oexpr, subscript_vexpr]
7244 --
7245 --   The array_oexpr is an OEXPR which evaluates to the array IOBJECT,
7246 --   and subscript_vexpr evaluates to integer or enum IVALUEs which
7247 --   are the subscript values.  The result is the selected component
7248 --   IOBJECT.
7249 --
7250
7251       ('[ ]'):
7252         [-, array, subscript_list] := eval_action;
7253         loop while subscript_list /= [ ] do
7254           subscript frome subscript_list; --PK79
7255           exec ( [['veval_', subscript],
7256                   ['oeval_', ['$subscript']] ]);
7257         end loop;
7258         exec( [['oeval_', array]] );
7259
7260       ('$subscript'):
7261         pop( subscript );
7262         pop( array );
7263         [-, seq, lowbound, highbound] := array;
7264         lowval := CONTENTS (lowbound);
7265         highval := CONTENTS (highbound);
7266         if absent(lowval) or absent(highval) then
7267           SYSTEM_ERROR('$subscript');
7268         elseif lowval = UNINITIALIZED or highval = UNINITIALIZED then
7269           SYSTEM_ERROR('$subscript');
7270         elseif subscript < lowval or subscript > highval then
7271           exec([['raise', 'CONSTRAINT_ERROR',
7272                           'subscript out of range']]);
```

```
7273        else
7274          push( seq (subscript - lowval + 1) );
7275        end if;
7276
```

4.2 Slices

```
7278 --  ['[..]', array_oexpr, slice_texpr]
7279 --
7280 --  The array_oexpr is an OEXPR which evaluates to the array IOBJECT.
7281 --  The slice_texpr entry is a TEXPR which evaluates to an ITYPE for
7282 --  discrete type whose range specifies the slice to be taken. If
7283 --  slice_texpr is a constraint operation, then it must be an
7284 --  'index_range' constraint.
7285 --
7286
7287      ('[..]'):
7288        [-, array, slice] := eval_action;
7289        exec ( [['oeval_', array],
7290                ['teval_', slice],
7291                ['oeval_', ['$slice']] ]);
7292
7293      ('$slice'):
7294        pop( slice );
7295        pop( array );
7296        [-, seq, lowbound, highbound] := array;
7297        [-, slicelo, slicehi] := slice;
7298        lowval := CONTENTS (lowbound);
7299        highval := CONTENTS (highbound);
7300
7301        if absent(lowval) or absent(highval) then
7302
7303          SYSTEM_ERROR('$slice');
7304
7305        elseif lowval = UNINITIALIZED or highval = UNINITIALIZED then
7306
7307          SYSTEM_ERROR('$slice');
7308
7309        elseif slicehi < slicelo then -- null slice(special bounds check)
7310                                      -- RM 4.1.2 para. 2, 3
7311          push(['array_iobject', [ ],
7312               CREATE_COPY(slicelo, Ω, Ω),
7313                  CREATE_COPY(slicehi, Ω, Ω)] );
7314
7315        elseif slicelo < lowval or highval < slicehi then
7316
7317          exec ( [['raise', 'CONSTRAINT_ERROR', 'out of range'] ]);
7318
7319        else
```

```
7320
7321        push( ['array_iobject',
7322                seq(slicelo - lowval + 1 .. slicehi - lowval + 1),
7323                CREATE_COPY(slicelo, Ω, Ω),
7324                CREATE_COPY(slicehi, Ω, Ω)] );
7325
7326      end if;
```

4.1.3 Selected Components (record)

```
7328 --  ['.', record_oexpr, fieldname]
7329 --
7330 --  The record_oexpr is an OEXPR which evaluates to the record IOBJECT.
7331 --  The fieldname is the name (a SETL string) of the field whose
7332 --  corresponding IOBJECT will be the returned result. Note that we must
7333 --  check for the field actually existing (may have wrong variant
7334 --  present).
7335
7336      ('.'):
7337        [-, record_oexpr, fieldname] := eval_action;
7338        exec ( [['oeval_', record_oexpr],
7339                ['oeval_', ['$select', fieldname]]]);
7340
7341      ('$select'):
7342        [-, fieldname] := eval_action;
7343        pop( record );
7344        [-, objdeclist, -] := record;
7345        if present(objdeclist(fieldname)) then
7346          fields := objdeclist('fields_present');
7347          if present(fields) then   -- record with unconstrained variants
7348
7349            if fieldname ∉ (CONTENTS(fields) ? {}) then
7350              exec ([['raise', 'CONSTRAINT_ERROR',
7351                      'field absent']]);
7352            else
7353              push( objdeclist(fieldname) );
7354            end if;
7355
7356          else -- no variants or constrained discriminants
7357
7358            push( objdeclist(fieldname) );
7359
7360          end if;
7361
7362        else -- fieldname absent from this object
7363
7364          exec( [['raise', 'CONSTRAINT_ERROR', 'field absent'] ]);
7365
7366        end if;
```

```
7367 --
7368 --
7369 -- Constrain A Record Object
7370 --
7371 --  ['make_constrained', oexpr, texpr]
7372 --
7373 --  Oexpr evaluates to a record IOBJECT, a copy of which, with the
7374 --  constr_discr_list set to that of the ITYPE corresponding to the
7375 --  texpr, is left on the stack. This is used as part of parameter
7376 --  passing, when the formal is constrained, but the actual is
7377 --  unconstrained.
7378 --
7379      ('make_constrained'):
7380        [-, oexpr, texpr] := eval_action;
7381        exec( [['oeval_', oexpr],
7382               ['teval_', texpr],
7383               ['oeval_', ['make_constrained_']] ]);
7384 --
7385      ('make_constrained_'):
7386        pop( itype );
7387        pop( iobject );
7388        [-, objdeclist] := iobject;
7389        [-, -, constr_discr_list] := itype;
7390        push( ['record_iobject', objdeclist, constr_discr_list] );
```

4.1.3 Selected Components (dereference)

```
7392 --  ['@', access_vexpr]
7393 --
7394 --  The access_vexpr entry is a VEXPR which when evaluated returns
7395 --  an IVALUE of access type (or of a renamed object). The result
7396 --  returned is the IOBJECT extracted from this value.
7397 --
7398        ('@'):
7399          [-, name] := eval_action;
7400          exec( [['veval_', name],
7401                 ['oeval_', ['$@']] ]);
7402 --
7403        ('$@'):
7404          pop( access_value );
7405          if is_access_ivalue(access_value) then    --
7406            [-, iobject] := access_value;
7407            if iobject /= NULL then
7408              push( iobject );
7409            else
7410              exec( [['raise', 'CONSTRAINT_ERROR', 'null access']] );
7411            end if;
7412          else
7413            SYSTEM_ERROR('oeval (dereferencing)');
```

```
7414          end if;
```

4.7 Qualifications

```
7416 -- Array Index Qualification
7417 --
7418 --  ['qual_index', typename, object]
7419 --
7420 --  The qual_index form is used to check the bounds of an array object
7421 --  for conformity to the bounds specified by an array type.The first
7422 --  parameter is the name of the type, and the second is an expression
7423 --  for the array to be qualified. This form is used for checking an
7424 --  actual array parameter against a constrained formal.
7425 --
7426       ('qual_index'):
7427         [-, typename, object] := eval_action;
7428         exec( [['oeval_', object],
7429               ['teval_', typename],
7430               ['oeval_', ['qual_index_']] ]);
7431 --
7432       ('qual_index_'):
7433         pop( itype );
7434         iobject := [TOP_VALSTACK];
7435         [-, index_list, itypename] := itype;
7436
7437         loop while index_list /= [ ] do
7438           index_itype fromb index_list;
7439
7440           [-, l_val, u_val] := GET_ITYPE(index_itype);
7441
7442           [-, iobject, lb, ub ] := iobject(1);
7443
7444           lbval := CONTENTS (lb);
7445           ubval := CONTENTS (ub);
7446
7447           if (u_val /= ubval or l_val /= lbval) then -- not same bounds
7448             exec([['raise', 'CONSTRAINT_ERROR',
7449                           'incompatible bounds']]);
7450             quit;
7451           end if;
7452 --
7453       end loop while;
7454 --
7455 --    ['qual_discr', typename, o_expr]
7456 --
7457 --    The  qual_discr form is used to check a record for conformity to
7458 --    the discriminants specified by a record type. The first parameter
7459 --    is the name of the type, and the second is an expression for the
7460 --    record to be qualified.
```

```
7461 --
7462      ('qual_discr'):
7463        [-, typename, o_expr] := eval_action;
7464        exec( [['oeval_', o_expr],
7465              ['teval_', typename],
7466              ['oeval_', ['qual_discr_']] ]);
7467 --
7468      ('qual_discr_'):
7469        pop( itype );
7470        iobject := TOP_VALSTACK;
7471        [-, objdeclist] := iobject;
7472        [-, component_list, constr_disc_list] := itype;
7473
7474        if is_set(constr_disc_list) then
7475                                          -- Constrained record type,
7476                                    -- ensure that record value
7477                                    -- has same values for
7478                                    -- discriminants.
7479
7480          [types_objdec_list, -] := component_list;
7481
7482 --
7483             if exists f ∈ constr_disc_list,
7484                          [d, [t, v]] ∈ types_objdec_list |
7485                  d = f and CONTENTS(objdeclist(f)) /= v(2) then
7486            exec( [['raise', 'CONSTRAINT_ERROR', 'discriminant'] ]);
7487          end if;
7488
7489        end if;
```

raise

```
7491 --  ['raise', exception_name]
7492 --
7493 --  Exceptions may be raised within expressions by use of this form.
7494 --  The raise is simply pushed onto the statement sequence.
7495 --
7496     ('raise'):
7497       exec( [eval_action] );
7498
```

call

```
7500 --
7501 --  a call in an expression can appear (to our knowledge) only in some
7502 --  weird cases of attributes. An object is built to contain the value
7503 --  returned by the function call
7504 --
7505     ('call'):
```

```
7506      itype frome eval_action;
7507      push( itype );
7508      exec( [eval_action,
7509            ['oeval_', ['call_']]]);
7510   ('call_'):
7511      pop(obj_val);
7512      pop(itype);
7513      obj_val := CREATE_COPY(obj_val, itype, Ω);
7514      push(obj_val);
7515
7516    else
7517 --
7518      SYSTEM_ERROR('attempt to evaluate unknown object expression '
7519        + str eval_action );
7520 --
7521    end case;
7522  end if is_simple_name(eval_action);
7523 end proc OEVAL_PROC;
```

TEVAL: Type evaluation (Chap.3)

```
7525 proc TEVAL_PROC(type_expr);
7526 --
7527 -- This procedure carrys out the bulk of the type expression evaluation
7528 -- needed by 'teval_' . Type_expr is the expression to be evaluated.
7529 --
7530 -- Forms processed by TEVAL_PROC:
7531 -- ------------------------------
7532 --
7533       if absent(type_expr) or type_expr = [ ] then
7534         SYSTEM_ERROR('in TEVAL_PROC: type_expr absent or [ ]');
7535       end if;
7536       case opcode(type_expr) of
7537 --
7538 -- Quoted ITYPE
7539 --
7540 -- An ITYPE is essentially a constant value, and can be referenced
7541 -- directly in a TEXPR by using the form:
7542 --
7543 --   ['itype', itype]
7544 --
7545 -- where itype is any of the allowable ITYPE values, including the
7546 -- string name of an ITYPE whose value is obtained through EMAP.
7547 --
7548       ('itype'):
7549         [-, typeval] := type_expr;
7550         push( typeval );
7551 --
7552 -- Anonymous ITYPE
7553 --
7554 --   ['tname', texpr]
7555 --
7556 --   Used when a type name must be supplied for a texpr.
7557 --   If texpr is already a name it is simply returned.If not, a name
7558 --   is generated, and a type entry is created in EMAP for the value
7559 --   of the texpr.
7560 --
7561       ('tname'):
7562         [-, texpr] := type_expr;
7563         if is_simple_name(texpr) then
7564           push( texpr );
7565         elseif opcode(texpr) = 'itype' then
7566           [-, itype_name] := texpr;
7567           push( itype_name );
```

```
7568        else
7569          new_name := '$anon' + (str newat);
7570          push( new_name );
7571          exec( [['type', new_name, texpr]]);
7572        end if;
7573 -- Private type with delayed elaboration
7574 --   [ 'delayed', delayed_statements ]
7575 --
7576 --    Internal type used to delay elaboration of subtypes of private
7577 --    types
7578 --
7579    ('delayed'):
7580        push( ['delayed', [ ]] );--
7581 -- Fixed former
7582 --
7583 --   ['fixed_former', lower_bound_ivalue, upper_bound_ivalue, delta]
7584 --
7585 -- This forms returns a fixed ITYPE. The language rules garantee that
7586 -- the bounds and the delta are static expressions (ivalues).
7587
7588     ('fixed_former'):
7589        [-, lbd, ubd, delta] := type_expr;
7590        push( ['fixed', lbd(2), ubd(2), delta(2)] );
7591 --
7592 -- Enumeration Former
7593 --
7594 --   ['enum_former', lower_bound_ivalue, upper_bound_ivalue, enum_map]
7595 --
7596 --   This form returns an enumeration ITYPE, whose map
7597 --   is given by the enum_map entry, and whose bounds are given
7598 --   by the bounds ivalues, which are non_negative integers.
7599 --   This form is used when an enumeration type is constructed
7600 --   which is not declared in the source as a subtype of a previously
7601 --   defined enumeration type. The language rules guarantee that all
7602 --   quantities may be evaluated statically in this case.
7603 --
7604 --
7605       ('enum_former'):
7606         [-, lbd, ubd, enum_map] := type_expr;
7607         push( ['enum', lbd, ubd, enum_map] );
7608 --
7609 -- Access Former
7610 --
7611 --  ['access_former', object_texpr]
7612 --
7613 --  Object_texpr evaluates to the type of the access object.
7614 --
7615       ('access_former'):
7616         [-, t_expr] := type_expr;
```

```
7617        exec( [['teval_', ['tname', t_expr]],
7618              ['teval_', ['access_former_']] ]);
7619 --
7620      ('access_former_'):
7621        pop( typename );
7622        push( ['access', typename, CURTASK, #ENVSTACK] );
7623 --
7624 --
7625 -- Array Former
7626 --
7627 --  ['array_former', [index_texpr, index_texpr, ...],
7628 --                              element_texpr, is_constrained]
7629 --
7630 --  Again this is similar to the corresponding ITYPE form, except
7631 --  that ITYPE references are replaced by TEXPR's which
7632 --  must be evaluated to obtain the ITYPE for the array, and the entry
7633 --  for the index type is a sequence, to account for the case of
7634 --  multi-dimensional arrays
7635 --
7636       ('array_former'):
7637         [-, index_list, element_name] := type_expr;
7638         push( ['array', index_list, element_name, false] );
7639
7640 --
7641 -- Record Former
7642 --
7643 --  ['record_former', component_list]
7644 --
7645 --    component_list  ::=  [objdec_list, variant_part]
7646 --    objdec_list     ::=  [[field_kind, field_name, field_texpr,
7647 --                           field_vexpr]...]
7648 --    field_kind      ::=  'field' | 'object'
7649 --    variant_part    ::=  [variant_name, altern_list] | [ ]
7650 --    altern_list     ::=  { [{choice, ...}, component_list], ... }
7651 --    choice          ::=  choice_vexpr | 'others'
7652 --
7653 --  The record former is very similar in structure to a record ITYPE.
7654 --  The differences are that ITYPE references are replaced by TEXPR
7655 --  references, and IVALUE's are replaced by VEXPR's. Note that if
7656 --  several different fields are defined at once (using comma), the
7657 --  front end expands them as defined in RM 3.7.3.
7658 --
7659 --  Discriminant fields have 'object' as the field_kind so that
7660 --  they may be extracted readily to form the disc_list for the itype.
7661 --  If the 'others' choice appears in altern_list, it must be the
7662 --  only component of the list for the last variant.  Note that the
7663 --  field_vexpr should be explictly
7664 --  qualified by the type of the field, if the type does not depend
7665 --  on a discriminant (Impl. Guide 3.7, RM 3.7, 3.2) .
```

```
7666 --
7667       ('record_former'):
7668         [-, component_list] := type_expr;
7669         [objdec_list, variant_part] := component_list;
7670
7671 --  There may be subtypes in the objdec_list, which must be
7672 --  elaborated first, then we will come back here...
7673
7674         inner_subtypes := [t: t ∈ objdec_list | opcode(t) = 'type'];
7675         if inner_subtypes /= [ ] then
7676           objdec_list := [t:t ∈ objdec_list | opcode(t) /= 'type'];
7677           component_list := [objdec_list, variant_part];
7678           exec(inner_subtypes with
7679               ['teval_', ['record_former', component_list]]);
7680           return;
7681         end if;
7682
7683         if variant_part /= [ ] then
7684           [variant_name, altern_list] := variant_part;
7685         else
7686           variant_name := Ω;
7687           altern_list := {};                          --
7688         end if;
7689         exec([['teval_', ['record_former_', #objdec_list, #altern_list,
7690                           variant_name]]]);
7691
7692         disc_list := {};
7693         loop for field_desc ∈ objdec_list do
7694           [field_kind, field_name, field_texpr, field_vexpr] :=
7695                                                       field_desc;
7696           field_name := field_name(1);    --
7697
7698           if field_kind = 'object' then
7699                                        -- extract discriminants
7700             disc_list with:= field_name;              --
7701           end if;
7702
7703 -- Put proper initialization expression in place. If there is none,
7704 -- then we supply a dummy one which assigns the value uninitialized.
7705
7706           if present(field_vexpr) then
7707             exec( [['push_', field_vexpr]]);          --
7708           else                          -- put place holder on stack
7709             exec( [['push_', ['ivalue', UNINITIALIZED]] ]);  --
7710           end if;
7711
7712           exec( [['teval_', ['tname', field_texpr]],
7713                  ['push_', field_name] ]);                 --
7714         end loop for field_desc;
```

```
7715
7716          exec([['push_', disc_list]]);    -- save discriminants -
7717
7718         loop for component_list = altern_list(choice_set) do
7719          exec( [['push_', #choice_set],                    --
7720                ['teval_', ['record_former', component_list]] ]);
7721          loop for choice ∈ choice_set do
7722           if is_range(choice) then                          --
7723            exec( [['push_', choice] ]);                  --
7724           elseif choice = 'others' then
7725            exec( [['push_', 'others'] ]);                --
7726           else -- single value
7727            exec( [['veval_', choice] ]);
7728           end if;
7729          end loop for choice;
7730         end loop for component_list;
7731 --
7732 --
7733      ('record_former_'):
7734
7735       [-, num_objs, num_alts, variant_name ] := type_expr;
7736       objdec_list := [ ];
7737
7738       loop for i ∈ [1..num_objs] do
7739        pop( field_vexpr );
7740        pop( field_name );
7741        pop( field_itype );
7742        objdec_list with:= [field_name, [field_itype, field_vexpr]];
7743       end loop for i;
7744
7745        pop( disc_list );
7746
7747       if absent(variant_name) then
7748        variant_part := [ ];
7749       else
7750        altern_list := {};                          --
7751        loop for i ∈ [1..num_alts] do
7752         pop( component_itype );
7753         [-, component_list] := component_itype;
7754         pop( num_choices );
7755         choices := {};
7756         loop for j ∈ [1..num_choices] do
7757          pop( choice );
7758          if is_range(choice) then       --
7759           [-, lc, uc] := choice;
7760           choices +:= {lc..uc}; --
7761          else
7762           choices with:= choice;
7763          end if;
```

```
7764            end loop for j;
7765
7766            altern_list (choices) := component_list;
7767
7768          end loop for i;
7769
7770          variant_part := [variant_name, altern_list];
7771
7772        end if absent(variant_name);
7773
7774        component_list := [objdec_list, variant_part];
7775        push( ['record', component_list, Ω, disc_list] );
7776
7777 --
7778 --
7779 -- Task Former
7780 --
7781 -- ['task_former', entry_list, priority]
7782 --
7783 -- Task_former processes a task type declaration. The body of the
7784 -- task is added by a 'task_body' statement (q.v.). The entry_list
7785 -- is a list of object declarations for the entries of the task.
7786 -- The type of the entry must have been previously declared by
7787 -- a type declaration using entry_former (q.v.).
7788 --
7789   ('task_former'):
7790     [-, entry_list, priority] := type_expr;
7791     exec( [['teval_', ['task_former_',
7792               priority(1) ? UNDEF_PRIO, #entry_list] ]]);
7793     ename_list := [ ];
7794     loop forall [-, [ename], etype] ∈ entry_list do
7795       ename_list with:= ename;
7796       exec( [['teval_', ['tname', etype]]] );
7797     end loop forall;
7798     push( ename_list );
7799 --
7800   ('task_former_'):
7801     [-, priority, num_entries] := type_expr;
7802     etype_list := [ ];
7803     loop forall i ∈ [1..num_entries] do
7804       pop( etype );
7805       etype_list with:= etype;
7806     end loop forall;
7807     pop( ename_list );
7808     entryids := [ ];
7809     loop forall i ∈ [1..num_entries] do
7810       ename frome ename_list;
7811       etype frome etype_list;
7812       etype_desc := GET_ITYPE(etype);
```

```
7813
7814      if type_mark(etype_desc) = 'array' then
7815        [-, [index_type], elmt_type] := etype_desc;
7816        exec([['object', [ename], elmt_type]]);
7817        get_bounds_of(GET_ITYPE(index_type));
7818
7819        loop forall i ∈ {lbd..ubd} do
7820            entryids with:= [ename, i];
7821        end loop forall;
7822
7823      else
7824        exec([['object', [ename], etype]]);
7825        entryids with:= [ename, 1];
7826      end if;
7827    end loop forall;
7828
7829    taskbody := '$body_of_' + str newat;
7830    push( ['task', entryids, priority, taskbody] );
7831 --
7832 -- Entry former
7833 --
7834 -- ['entry_former', task_name, formals]
7835 --
7836 -- Entries are treated as objects which have type entry. Entry_former
7837 -- creates an entry type in a type declaration. The entry objects
7838 -- are created by the task_former.
7839 --
7840    ('entry_former'):
7841      [-, -, formals] := type_expr; -- for the moment ignore the task name
7842                                    -- the front end guarantees that entry
7843                                    -- names are made unique
7844      push( ['entry', formals] );
7845 --
7846 -- Subtype Operation
7847 --
7848 -- ['subtype', typemark, constraint]
7849 --
7850 -- The subtype operation takes a typemark and a constraint
7851 -- and produces a new type description meeting that constraint.
7852 --
7853        ('subtype'):
7854          [-, typemark, constraint] := type_expr;
7855          [delay_flag, -] := GET_ITYPE(typemark) ? [ ];
7856          if delay_flag = 'delayed' then
7857            push( [ 'delayed', typemark ] );
7858          elseif constraint = [ ] then
7859            push( GET_ITYPE(typemark) );
7860          else
7861              case opcode(constraint) of
```

```
7862 --
7863 -- Index Constraint
7864 --
7865 --  ['subtype', array_itype, ['index', [texpr, texpr, ...]]]
7866 --
7867 --  The index constraint form is used to obtain an array ITYPE
7868 --  by applying the indicated list of index constraints. The
7869 --  array_itype entry type must be an array with the appropriate
7870 --  subscript count. The TEXPR entries evaluate to discrete (integer
7871 --  or enum) ITYPE's which specify the range on each bound.
7872 --  Typically these are explicit ITYPE's evaluated by the front end,
7873 --
7874    ('index'):
7875      [-, index_list] := constraint;
7876      [-, -, component_type] := GET_ITYPE(typemark);
7877      exec( [['teval_', ['$index', #index_list]]] );
7878      exec( [['push_', component_type]] );
7879      loop forall index ∈ index_list do
7880        exec( [['teval_', ['tname', index] ]] );
7881      end loop;
7882 --
7883 -- Range Constraint
7884 --
7885 --  ['subtype', discrete_itype, ['range', lowb_vexpr, highb_vexpr]]
7886 --
7887 --  The range constraint is used to form an ITYPE by constraining
7888 --  an existing integer or enumeration ITYPE with a bounds constraint.
7889 --  The bounds expressions are integer VEXPR's.
7890 --
7891 --
7892 --
7893      ('range'):
7894         [-, l_vexpr, u_vexpr] := constraint;
7895        exec( [['teval_', typemark],
7896              ['veval_', l_vexpr],
7897               ['veval_', u_vexpr],
7898               ['teval_', ['$range']] ]);
7899 --
7900 -- Accuracy Constraint
7901 --
7902 --  ['subtype', fixed_itype, ['delta', lb_vexpr, ub_vexpr, delta_ivalue]]
7903 --  ['subtype', float_itype, ['digits', lb_vexpr, ub_vexpr, digs_ivalue]]
7904 --
7905 --
7906 --  These forms are used to constrain existing float or fixed ITYPE's
7907 --  by applying bounds constraints (given as float or fixed VEXPR's)
7908 --  and either a digits constraint (given as an integer IVALUE), or
7909 --  a delta value (given as a fixed IVALUE).
7910 --
```

```
7911 --
7912                ('delta'):
7913                  [-, l_vexpr, u_vexpr, delta] := constraint;
7914                  exec([['teval_', typemark],
7915                        ['veval_', delta],
7916                        ['veval_', l_vexpr],
7917                        ['veval_', u_vexpr],
7918                        ['teval_', ['$delta']] ]);
7919 --
7920                ('digits'):
7921                  [-, l_vexpr, u_vexpr, digits] := constraint;
7922                  exec([['teval_', typemark],
7923                        ['veval_', digits],
7924                        ['veval_', l_vexpr],
7925                        ['veval_', u_vexpr],
7926                        ['teval_', ['$digits']]]);
7927 --
7928 -- Discriminant Constraint
7929 --
7930 --  ['subtype', record_itype, ['discr', record_aggregate]]
7931 --
7932 --  This form is used to obtain an ITYPE for the result of applying
7933 --  discriminant constraints to an ITYPE for an existing record type.
7934 --  The record_aggregate component is a standard format record
7935 --  aggregate VEXPR whose entries give the names and values
7936 --  of the discriminants which form the constraint. If the ITYPE being
7937 --  created is the type of a record componant of another record, one
7938 --  or more of the discriminant values in the record_aggregate may be
7939 --  of the form ['discr_ref', discrim_name], if these values are the
7940 --  values of the correspondingly named discriminants of the enclosing
7941 --  record.
7942 --
7943 --
7944        ('discr'):
7945          [-, field_list] := constraint;
7946          new_field_list := {};
7947          loop for [field_names, field_vexpr] ∈ field_list do
7948           loop for field_name ∈ field_names do
7949              new_field_list(field_name) := field_vexpr;
7950           end loop;
7951          end loop for;
7952
7953          exec( [['teval_', typemark],
7954                 ['teval_', ['$discr', #new_field_list]] ]);
7955          loop for [field_name, field_vexpr] ∈ new_field_list do
7956             exec( [['veval_', field_vexpr],
7957                    ['push_', field_name ]] );
7958          end loop for;
7959 --
```

```
7960 --
7961          else
7962            SYSTEM_ERROR('constrain');
7963 --
7964          end case opcode(constraint);
7965 --
7966        end if;
7967 --
7968          ('$index'):
7969            [-, nbdim] := type_expr;
7970            pop( component_name );
7971            index_list := [ ];
7972            loop while (nbdim -:= 1) >= 0 do
7973               pop( index_name );
7974               index_list with := index_name;
7975            end loop;
7976            push( ['array', index_list, component_name, true] );
7977 --
7978 --
7979          ('$range'):
7980            pop( valhi );
7981            pop( vallo );
7982            pop( typedesc );
7983              case type_mark(typedesc) of
7984              ('integer'):
7985                 get_bounds_of(typedesc);
7986                if is_discr_ref(vallo)                          --
7987                   or is_discr_ref(valhi) then                  --
7988                  push( ['integer', vallo, valhi] );
7989                  elseif vallo > valhi then -- null range
7990
7991 -- temp cut out constraint checks for null ranges see IG 3.6.1.a
7992
7993 --                if lbd <= vallo and vallo <= ubd then
7994                    push( ['integer', vallo, valhi] );
7995 --                 else
7996 --                   exec( [['raise', 'CONSTRAINT_ERROR',
7997 --                                    'out of bounds']] );      -
7998 --                 end if;
7999                  elseif lbd <= vallo and valhi <= ubd then
8000                     push( ['integer', vallo, valhi] );
8001               else
8002                     exec([['raise', 'CONSTRAINT_ERROR', 'out of bounds']]);
8003                end if;
8004              ('enum'):
8005                 get_bounds_of(typedesc);                                --
8006                if is_discr_ref(vallo)                          --
8007                   or is_discr_ref(valhi) then                  --
8008                  push( ['enum', vallo, valhi, enumlist] );
```

```
8009              elseif vallo > valhi then -- null range
8010
8011  -- see above
8012  --                 if lbd <= vallo and vallo <= ubd then
8013                     push( ['enum', vallo, valhi, enumlist] );
8014  --                 else
8015  --                   exec( [['raise', 'CONSTRAINT_ERROR',
8016  --                                 'out of bounds']] );
8017  --                 end if;
8018                   elseif lbd <= vallo and valhi <= ubd then
8019                     push( ['enum', vallo, valhi, enumlist] );
8020                 else
8021                     exec([['raise', 'CONSTRAINT_ERROR', 'out of bounds']]);
8022                 end if;
8023               else
8024                 SYSTEM_ERROR('$index_range');
8025             end case type_mark(typedesc);
8026  --
8027  --
8028           ('$digits'):
8029             pop( valhi );
8030             pop( vallo );
8031             pop( new_digits );
8032             pop( typedesc );
8033             get_bounds_of(typedesc);
8034             if type_mark(typedesc) /= 'float' then
8035               SYSTEM_ERROR('$digits');
8036             elseif new_digits > digits or new_digits <= 0 then
8037               exec([['raise', 'CONSTRAINT_ERROR', 'digits']]);
8038             elseif lbd <= vallo and valhi <= ubd then
8039               push( ['float', vallo, valhi, new_digits] );
8040             else
8041               exec( [['raise', 'CONSTRAINT_ERROR', 'out of bounds']]);
8042             end if;
8043  --
8044           ('$delta'):
8045             pop( valhi );
8046             pop( vallo );
8047             pop( new_delta );
8048             pop( typedesc );
8049             get_bounds_of(typedesc);
8050             if type_mark(typedesc) /= 'fixed' then
8051               SYSTEM_ERROR('$delta');
8052             elseif fix_lss(new_delta, delta) then
8053               exec([['raise', 'CONSTRAINT_ERROR', 'too small delta']]);
8054             elseif fix_leq(lbd, vallo) and fix_leq(valhi, ubd) then
8055               push( ['fixed', vallo, valhi, new_delta] );
8056             else
8057               exec( [['raise', 'CONSTRAINT_ERROR', 'out of bounds']]);
```

```
8058          end if;
8059 --
8060       ('$discr'):
8061         [-, num_field_values] := type_expr;
8062         pop( rectype );
8063         field_map := {};
8064         loop for i ∈ [1..num_field_values] do --
8065           pop( field_name);
8066           pop( field_value );
8067           field_map( field_name ) := field_value;
8068         end loop for;
8069         push( APPLY_DISCR(field_map, rectype) );
8070 --
8071       (""):
8072         ATTRIBUTE_EVAL(type_expr); -- for 'range only
8073 --
8074    else
8075      SYSTEM_ERROR('attempt to elaborate unknown type ' + str type_expr);
8076 --
8077    end case;
8078 end proc TEVAL_PROC;
```

OPERATORS

OPBINARY

```
8081 proc OPBINARY(opname);
8082 --
8083 -- OPBINARY( opname)
8084 -- ----------------
8085 --
8086 -- Evaluates binary operator opname. The operands are assumed to have
8087 -- been evaluated onto VALSTACK. This is the lowest level of the
8088 -- evaluation process for arithmetic expressions by 'veval_' .
8089 --
8090  macro check_overflow;
8091   case type VALSTACK(#VALSTACK) of
8092    ('INTEGER'):
8093     if abs VALSTACK(#VALSTACK) > ADA_MAX_INTEGER then
8094      exec([['raise', 'NUMERIC_ERROR', 'integer overflow']]);
8095     end if;
8096    ('REAL'):
8097     if abs VALSTACK(#VALSTACK) > ADA_MAX_REAL then
8098      exec([['raise', 'NUMERIC_ERROR', 'real overflow']]);
8099     end if;
8100    else
8101     SYSTEM_ERROR('check_overflow');
8102   end case
8103  endm;
8104
8105  macro fix_0; fix_fri(0) endm;      --
8106 --
8107 -- Operations evalutated by OPBINARY are:
8108 --
8109    pop( op2 );
8110    pop( op1 );
8111    if absent(op1) or absent(op2) then
8112     SYSTEM_ERROR('opbinary');
8113     return;
8114    end if;
8115    case opname of
8116
8117
8118 -- '+i'   integer addition
8119 -- '+fl'  floating addition
8120
```

```
8121      ('+i', '+fl'):
8122        push( op1 + op2 );
8123        check_overflow;
8124
8125
8126 -- '-i'   integer subtraction
8127 -- '-fl'  floating subtraction
8128
8129      ('-i', '-fl'):
8130        push( op1 - op2 );
8131        check_overflow;
8132
8133
8134 -- '*fl'  floating multiplication
8135
8136      ('*fl'):
8137        if (abs op1 < ADA_SMALL_REAL) or (abs op2 < ADA_SMALL_REAL) then
8138          push( 0.0 );
8139        elseif log abs op1 + log abs op2 > log ADA_MAX_REAL then
8140          exec( [['raise', 'NUMERIC_ERROR', 'real overflow in *']]);
8141        else
8142          push( op1 * op2 );
8143          check_overflow;
8144        end if;
8145
8146 -- '/fli'
8147 -- '*fli'
8148      ('/fli','*fli'):
8149         push( op1 );
8150         push( float(op2) );
8151         OPBINARY(opname(1..3));
8152
8153 -- '+fx'  fixed addition
8154
8155      ('+fx'):
8156        push( fix_add(op1, op2) );
8157
8158 -- '-fx'  fixed subtraction
8159
8160      ('-fx'):
8161        push( fix_sub(op1, op2) );
8162
8163 -- '*fx'  fixed multiplication
8164
8165      ('*fx'):
8166        push( fix_mul(op1, op2) );
8167
8168 -- '/fx'  fixed division
8169
```

```
8170      ('/fx'):
8171        if fix_eql(op2, fix_0) then       --
8172          exec( [['raise', 'NUMERIC_ERROR', 'divide by zero'] ]);
8173        else
8174          push( fix_div(op1, op2) );
8175        end if;
8176
8177 -- '*fxi'  multiply fixed by integer
8178
8179      ('*fxi'):
8180        push( fix_mul(op1, fix_fri(op2)) );                    --
8181
8182 -- '*ifx'  multiply integer by fixed
8183
8184      ('*ifx'):
8185        push( fix_mul(fix_fri(op1), op2) );                    --
8186
8187 -- '/fxi'  divide fixed by integer
8188
8189      ('/fxi'):
8190        if op2 = 0 then
8191          exec( [['raise', 'NUMERIC_ERROR', 'divide by zero']] );
8192        else
8193          push( fix_div(op1, fix_fri(op2)));                   --
8194        end if;
8195
8196 -- '*i'    integer multiplication
8197
8198       ('*i'):
8199        sign_result := (sign op1) * (sign op2);
8200        if sign_result = 0 then
8201          push( 0 );
8202        else
8203          [op1, op2] := [abs op1, abs op2];
8204          if (op1 < ADA_MAX_INTEGER_SQRT)
8205              and (op2 < ADA_MAX_INTEGER_SQRT) then
8206            push( sign_result * op1 * op2 );
8207          else
8208            [a, b] := [op1 div ADA_MAX_INTEGER_SQRT,
8209                       op1 mod ADA_MAX_INTEGER_SQRT];
8210            [c, d] := [op2 div ADA_MAX_INTEGER_SQRT,
8211                       op2 mod ADA_MAX_INTEGER_SQRT];
8212            if a /= 0 and c /= 0 then
8213              exec( [['raise', 'NUMERIC_ERROR', 'overflow']] );
8214            elseif (result := b*c + a*d) >= ADA_MAX_INTEGER_SQRT then
8215              exec([['raise', 'NUMERIC_ERROR', 'overflow']]);
8216            else
8217              push( sign_result *
8218                     (result*ADA_MAX_INTEGER_SQRT + b*d) );
```

```
8219              check_overflow;
8220            end if a;
8221          end if (op1;
8222          end if sign_result;
8223
8224 -- '/i'   integer division
8225
8226      ('/i'):
8227        if op2 = 0 then
8228          exec( [['raise', 'NUMERIC_ERROR', 'divide by zero']] );
8229        else
8230          push( op1 div op2 );
8231            check_overflow;
8232        end if;
8233
8234 -- '/fl'  floating division
8235
8236      ('/fl'):
8237        if abs op2 < ADA_SMALL_REAL then
8238          exec([['raise', 'NUMERIC_ERROR', 'overflow']]);
8239        elseif abs op1 < ADA_SMALL_REAL then
8240          push( 0.0 );
8241        elseif log abs op1 - log abs op2 > log ADA_MAX_REAL then
8242          exec( [['raise', 'NUMERIC_ERROR', 'overflow']] );
8243        else
8244          push( op1 / op2 );
8245          check_overflow;
8246        end if;
8247
8248 -- '='   equality test
8249
8250      ('='):
8251        push( test(EQUAL_VALUES(op1, op2)) );
8252
8253 -- '/='   inequality test
8254
8255      ('/='):
8256        push( test(not EQUAL_VALUES(op1, op2)) );
8257
8258 -- '<'   less then test
8259
8260      ('<'):
8261        push( test(LESS_THAN_VALUES(op1, op2)) );
8262
8263 -- '<='  less then or equal test
8264
8265      ('<='):
8266        push(test(LESS_THAN_VALUES(op1, op2) or
8267                  EQUAL_VALUES(op1, op2)));
```

```
8268
8269 -- '>'  greater then test
8270
8271      ('>'):
8272        push( test(LESS_THAN_VALUES(op2, op1)) );
8273
8274 -- '>='  greater then or equal test
8275
8276      ('>='):
8277        push(test(LESS_THAN_VALUES(op2, op1) or
8278                  EQUAL_VALUES(op1, op2)));
8279
8280 -- '**i'  integer to integer power
8281
8282      ('**i'):
8283        if op2 < 0 then
8284          exec( [['raise', 'CONSTRAINT_ERROR', 'negative exp']] );
8285        end if;
8286        if op1 >= 0 then
8287          sign_result := 1;
8288        else
8289          sign_result := if even op2 then 1 else -1 end;
8290          op1 := abs op1;
8291        end if;
8292        result := 1;
8293        loop forall i ∈ [1..op2] do
8294          result *:= op1;
8295          if result > ADA_MAX_INTEGER then
8296            exec([['raise', 'NUMERIC_ERROR', 'overflow']]);
8297            quit;
8298          end if;
8299        end loop forall;
8300        push( sign_result * result );
8301
8302 -- '**fl' float to integer power
8303
8304      ('**fl'):
8305        if op1 = 0.0 and op2 < 0 then
8306          exec( [['raise', 'NUMERIC_ERROR', 'divide by zero' ]] );
8307        end if;
8308        if op1 >= 0.0 then
8309          sign_result := 1.0;
8310        else
8311          sign_result := if even (abs op2) then 1.0 else -1.0 end;
8312            op1 := abs op1;
8313        end if;
8314        result := 1.0;
8315        loop forall i ∈ [1..abs op2] do
8316          result *:= op1;
```

```
8317            if result > ADA_MAX_REAL then
8318              quit;
8319            end if;
8320        end loop forall;
8321        if op2 >= 0 then
8322           push( sign_result * result );
8323           check_overflow;
8324        elseif result < ADA_SMALL_REAL then
8325            exec([['raise', 'NUMERIC_ERROR', 'underflow']]);
8326        else
8327           push( sign_result / result );
8328            check_overflow;
8329        end if;
8330
8331 -- '&'   string concatenation
8332
8333     ('&'):
8334        [-, seq_op1, l_op1, u_op1] := op1;
8335        [-, seq_op2, l_op2, u_op2] := op2;
8336        len_op1 := if l_op1 <= u_op1 then
8337                     u_op1 - l_op1 + 1
8338                   else 0
8339                   end;
8340        len_op2 := if l_op2 <= u_op2 then
8341                     u_op2 - l_op2 + 1
8342                   else 0
8343                   end;
8344        if len_op1 = 0 then
8345          push( op2 );
8346        elseif len_op2 = 0 then
8347          push( op1 );
8348        else
8349          push(['array_ivalue', seq_op1 + seq_op2,
8350               l_op1, l_op1 + len_op1 + len_op2 - 1 ]);
8351        end if;
8352
8353 -- 'and'  boolean and
8354
8355     ('and'):
8356        if is_literal_ivalue(op1) then
8357          push( op1 min op2 );
8358        else
8359          [-, seq_op1, l_op1, u_op1] := op1;
8360          [-, seq_op2, -, -] := op2;
8361          new_seq := [seq_op1(i) min seq_op2(i)
8362                     : i ∈ [1..u_op1 - l_op1 + 1]];
8363          push( ['array_ivalue', new_seq, l_op1, u_op1] );
8364        end if;
8365
```

```
8366 -- 'remi'  integer remainder
8367
8368      ('remi'):
8369        if op2 = 0 then
8370          exec( [['raise', 'NUMERIC_ERROR','modulo zero']] );
8371        else
8372          push( op1 - (op1 div op2)*op2 );
8373          check_overflow; --
8374        end if;
8375
8376 -- 'modi'  integer modulus
8377
8378      ('modi'):
8379        if op2 = 0 then
8380          exec( [['raise', 'NUMERIC_ERROR', 'modulo zero']] );
8381        else
8382            rm := op1 mod op2;
8383            push( if rm = 0 or op2 > 0 then rm else rm + op2 end); --
8384          check_overflow; --
8385        end if;
8386
8387 -- 'or'  boolean or
8388
8389      ('or'):
8390        if is_literal_ivalue(op1) then
8391          push( op1 max op2 );
8392        else
8393          [-, seq_op1, l_op1, u_op1] := op1;
8394          [-, seq_op2, -, -] := op2;
8395          new_seq := [seq_op1(i) max seq_op2(i)
8396                    : i ∈ [1..u_op1 - l_op1 + 1]];
8397          push( ['array_ivalue', new_seq, l_op1, u_op1] );
8398        end if;
8399
8400 -- 'xor'  boolean exclusive or
8401
8402      ('xor'):
8403        if is_literal_ivalue(op1) then
8404          push( test(op1 /= op2) );
8405        else
8406          [-, seq_op1, l_op1, u_op1] := op1;
8407          [-, seq_op2, -, -] := op2;
8408          new_seq := [test(seq_op1(i) /= seq_op2(i))
8409                    : i ∈ [1..u_op1 - l_op1 + 1]];
8410          push( ['array_ivalue', new_seq, l_op1, u_op1] );
8411        end if;
8412
8413      else
8414        SYSTEM_ERROR('attempt to evaluate unknown binary operator '
```

```
8415                    + str opname);
8416    end case;
8417 --
8418 --
8419 --
8420 drop check_overflow, fix_0;  --
8421 end proc OPBINARY;
```

OPUNARY

```
8423 proc OPUNARY(opname);
8424 --
8425 -- OPUNARY( opname)
8426 -- ---------------
8427 --
8428 -- Evaluates unary operators for 'veval_' . The operand has been
8429 -- evaluated onto VALSTACK.
8430 --
8431 -- Operators evaluated by OPUNARY are:
8432 --
8433    pop( op1 );
8434    if absent(op1) then
8435      SYSTEM_ERROR('opunary');
8436      return;
8437    end if;
8438    case opname of
8439
8440 -- '+ui'  unary integer plus
8441 -- '+ufl' unary float plus
8442 -- '+ufx' unary fixed plus
8443
8444      ('+ui', '+ufl', '+ufx'):
8445        push( op1 );
8446
8447 -- '-ui'  unary integer minus
8448 -- '-ufl' unary float minus
8449
8450      ('-ui', '-ufl'):
8451        push( -op1 );
8452
8453 -- '-ufx' unary fixed minus
8454
8455      ('-ufx'):
8456        push( fix_umin(op1) );
8457
8458 -- 'not'  boolean not
8459
8460      ('not'):
8461        if is_literal_ivalue(op1) then
```

```
8462            push( test(op1 = boolean_false) );
8463          else
8464            [-, seq_op1, l_op1, u_op1] := op1;
8465            new_seq := [test(seq_op1(i) = boolean_false)
8466                        : i ∈ [1..u_op1 - l_op1 + 1]];
8467            push( ['array_ivalue', new_seq, l_op1, u_op1] );
8468          end if;
8469
8470 -- 'absi'  integer absolute value
8471 -- 'absfl' float absolute value
8472
8473       ('absi', 'absfl'):
8474         push( abs op1 );
8475
8476 -- 'absfx' fixed absolute value
8477
8478       ('absfx'):
8479         push( fix_abs(op1) );
8480
8481
8482 -- 'row' rows a scalar operand of &
8483
8484       ('row'):
8485         push( ['array_ivalue', [op1], 1, 1] );
8486
8487       else
8488         SYSTEM_ERROR('attempt to evaluate unknown unary operator '
8489                      + str opname);
8490     end case;
8491 end proc OPUNARY;
```

EQUAL_VALUES

```
8493 proc EQUAL_VALUES(op1, op2);
8494 --
8495 -- EQUAL_VALUES(op1, op2)
8496 -- ---------------------
8497 --
8498 -- Used in comparison operators to test for equality of IVALUES.
8499 -- (Setl = cannot be used because of definition of equality for
8500 -- array and record values RM 4.5.2 para. 5 and 6.)
8501 --
8502 --
8503   if is_fixed_ivalue(op1) then
8504     return fix_eql(op1, op2);
8505   elseif is_array_ivalue(op1) then
8506     [-, seq1, l_op1, u_op1, null1] := op1;    --
8507     [-, seq2, l_op2, u_op2, null2] := op2;
8508     -- two null arrays are equal 4.5.2(5)
```

```
8509    if    ((null1 = 'null') or (u_op1 < l_op1))
8510      and ((null2 = 'null') or (u_op2 < l_op2)) then
8511      return true;
8512    elseif  u_op1 - l_op1 /= u_op2 - l_op2 then
8513      return false;
8514    else
8515      return (forall i ∈ [1..u_op1 - l_op1 + 1]
8516            | EQUAL_VALUES(seq1(i), seq2(i)) );
8517    end if;
8518  elseif is_record_ivalue(op1) then
8519    [-, r_map1] := op1;
8520    [-, r_map2] := op2;
8521    dr1 := domain r_map1;
8522    dr2 := domain r_map2;
8523
8524    if 'fields_present' ∈ dr1 then
8525      dr1 := r_map1('fields_present');
8526    end if;
8527    if 'fields_present' ∈ dr2 then
8528      dr2 := r_map2('fields_present');
8529    end if;
8530
8531    if dr1 /= dr2 then
8532      return false;
8533
8534    else
8535
8536      return (forall f_name ∈ dr1
8537            | EQUAL_VALUES(r_map1(f_name), r_map2(f_name)) );
8538
8539    end if;
8540
8541  else -- other ivalues (in particular access values)
8542       -- can make do with Setl =
8543    return op1 = op2;
8544  end if;
8545 end proc EQUAL_VALUES;
```

LESS_THAN_VALUES

```
8547 proc LESS_THAN_VALUES(op1, op2);
8548 --
8549 -- LESS_THAN_VALUES(op1, op2)
8550 -- -------------------------
8551 -- Used in comparison operators to test for less than.
8552 -- RM 4.5.2 requires test to be availiable only for discrete
8553 -- values and one-dimensional arrays of discrete values.
8554 --
8555 --
```

```
8556  if is_literal_ivalue(op1) then
8557    return op1 < op2;
8558  elseif is_fixed_ivalue(op1) then
8559    return fix_lss(op1, op2);
8560  elseif is_array_ivalue(op1) then
8561    [-, seq1, l_op1, u_op1] := op1;
8562    [-, seq2, l_op2, u_op2] := op2;
8563    len1 := (u_op1 - l_op1 + 1) max 0;
8564    len2 := (u_op2 - l_op2 + 1) max 0;
8565    loop for i ∈ [1 .. len1 min len2] do
8566      if seq1(i) > seq2(i) then
8567        return false;
8568      elseif seq1(i) < seq2(i) then
8569          return true;
8570      end if;
8571    end loop for;
8572    return len1 < len2;
8573  else
8574    SYSTEM_ERROR('LESS_THAN_VALUES');
8575  end if;
8576 end proc LESS_THAN_VALUES;
```

DEBUGGING PROCEDURES

DO_DUMPS

```
8579 proc DO_DUMPS(tables);
8580 --
8581 -- DO_DUMPS(tables)
8582 -- ---------------
8583 --
8584 -- Dumps internal tables for debugging. Parameter tables is a tuple of
8585 -- strings which are the names of the tables to be dumped.
8586 --
8587    if TRACE_MODE = 'nolist' then return; end if;
8588    loop forall table ∈ tables do
8589       case table of
8590          ('VALSTACK'):
8591           if VALSTACK /= [ ] then
8592             PRETTY_PRINT(ERRFILE,
8593              [ERR_PLACE() + ' ' + 'VALSTACK--', VALSTACK]);
8594           end if;
8595 --
8596          ('EMAP'):
8597             PRETTY_PRINT(ERRFILE, [ERR_PLACE() + ' ' +
8598                    'EMAP-- ', EMAP]);
8599 --
8600          ('STSQ'):
8601             PRETTY_PRINT(ERRFILE,
8602              [ERR_PLACE() + ' ' + 'STSQ--', STSQ]);
8603 --
8604          ('CONTENTS'):
8605             PRETTY_PRINT(ERRFILE, [ERR_PLACE() + ' ' +
8606                  'CONTENTS--', CONTENTS]);
8607 --
8608          ('ENVSTACK'):
8609             PRETTY_PRINT(ERRFILE, [ERR_PLACE() + ' ' +
8610                    'ENVSTACK-- ', ENVSTACK]);
8611 --
8612          ('WAITING_TASKS'):
8613             PRETTY_PRINT(ERRFILE, [ERR_PLACE() + ' ' +
8614                  'WAITING_TASKS--', WAITING_TASKS]);
8615 --
8616          ('READY_TASKS'):
8617             PRETTY_PRINT(ERRFILE, [ERR_PLACE()  + ' ' +
8618                  'READY_TASKS--', READY_TASKS]);
```

```
8619 --
8620          ('ACTIVE_TASKS'):
8621             PRETTY_PRINT(ERRFILE,
8622                   ['ACTIVE_TASKS--', ACTIVE_TASKS]);
8623 --
8624          ('TASK_PRIO'):
8625             PRETTY_PRINT(ERRFILE,
8626                   ['TASK_PRIO--', TASK_PRIO]);
8627 --
8628          ('TERMINATABLE'):
8629             PRETTY_PRINT(ERRFILE,
8630                ['TERMINATABLE--', TERMINATABLE]);
8631 --
8632          ('TERMINATED_TASKS'):
8633             PRETTY_PRINT(ERRFILE,
8634              ['TERMINATED_TASKS--', TERMINATED_TASKS]);
8635 --
8636          ('COMPLETED_TASKS'):       --
8637             PRETTY_PRINT(ERRFILE,
8638              ['COMPLETED_TASKS--', COMPLETED_TASKS]);
8639 --
8640          ('ABNORMAL_TASKS'):        --
8641             PRETTY_PRINT(ERRFILE,
8642              ['ABNORMAL_TASKS--', ABNORMAL_TASKS]);
8643 --
8644          ('TASKS_DECLARED'):
8645             t_list := [t : [t, -, -] ∈ TASKS_DECLARED];
8646             PRETTY_PRINT(ERRFILE,
8647                ['TASKS_DECLARED--', t_list]);
8648 --
8649          ('HELD_TASKS'):
8650             PRETTY_PRINT(ERRFILE,
8651                   ['HELD_TASKS--', HELD_TASKS]);
8652 --
8653          ('OPEN_ENTRIES'):
8654             PRETTY_PRINT(ERRFILE, [ERR_PLACE() + ' ' +
8655                   'OPEN_ENTRIES--', OPEN_ENTRIES]);
8656 --
8657          ('DELAYED_TASKS'):
8658             delays := {[t, d] : [d, b] = DELAYED_TASKS(t)};
8659             PRETTY_PRINT(ERRFILE,
8660                   ['DELAYED_TASKS--', delays]);
8661 --
8662          ('ENTERING'):
8663             PRETTY_PRINT(ERRFILE,
8664                   ['ENTERING--', ENTERING]);
8665 --
8666          ('MASTER'):
8667             PRETTY_PRINT(ERRFILE, ['MASTER--', MASTER]);
```

```
8668 --
8669            ('TASKENV'):
8670               PRETTY_PRINT(ERRFILE, ['TASKENV--', TASKENV]);
8671 --
8672 --
8673 --
8674 --
8675         end case;
8676      end loop forall;
8677 end proc DO_DUMPS;
8678
```

ERR_LINE

```
8680 proc ERR_LINE;
8681 --
8682 -- ERR_LINE
8683 -- --------
8684 --
8685 -- Returns formated string giving  '*** Execution error' message
8686 -- with the place where the error occured.
8687 --
8688    return '*** Execution error in ' + ERR_PLACE() + ': ';
8689
8690 end proc ERR_LINE;
```

ERR_PLACE

```
8692 proc ERR_PLACE;
8693 --
8694 -- ERR_PLACE
8695 -- ---------
8696 --
8697 -- Returns a string giving the name of the current task type and
8698 -- procedure being executed. Usually used for error reporting.
8699 -- (When the front end starts providing it, the current
8700 -- statement line will also be given).
8701 --
8702 -- If the debug switch is on, the dynamic nesting depth is also
8703 -- given.
8704 --
8705
8706    line := EMAP('line_number') ? '??'; --
8707    return    TASKNAME_MAP(CURTASK) ? '' + ' '
8708           + UNQUALED_NAME(PROC_NAME ? '') + ' '
8709           + (if CDEBUG5 /= 0 or STRACE > 0
8710            then str #(ENVSTACKT(CURTASK) ? [ ]) + ' '
8711            else '' end )
8712           + ' statement ' + str(line) + ' '; --
```

```
8713 --
8714 end proc ERR_PLACE;
```

UNQUALED_NAME

```
8716 proc UNQUALED_NAME(q_name);
8717 --
8718 -- UNQUALED_NAME(q_name)
8719 -- --------------------
8720 --
8721 -- Q_name is the qualified name of an object as received
8722 -- from the front end. It is assumed to be in one of the forms:
8723 --
8724 --  'qual1.qual2.....qualn.original_name#1#2.....#n'
8725 --  'original_name#1#2.....#n'
8726 --  'qual1.qual2....qualn.original_name'
8727 --  'original_name'
8728 --
8729 -- Unqualed_name returns original_name, which ought to be
8730 -- the name of the object as the user wrote it in the source
8731 -- program. If q_name is not in one of the above forms, the
8732 -- result returned is undefined.
8733 --
8734   q_temp := q_name;
8735   res1 := rbreak(q_temp, '.');
8736   res1 ?:= q_temp;
8737   res := break(res1, '#');
8738   return res ? res1;
8739 end proc UNQUALED_NAME;
```

DUMP_TASKING_INFO

```
8741 proc DUMP_TASKING_INFO(file);
8742
8743 -- This procedure dumps information about tasking queues when
8744 -- system becomes inactive.  It reports what each waiting task
8745 -- is waiting for when no task can continue executing.
8746 -- In addition to that this procedure takes care of a very
8747 -- hard-to-debug situtation when a master task which has raised an
8748 -- exception cannot propogate it out of the scope because a
8749 -- dependant task has not completed (possibly precisely because
8750 -- of this exception) .
8751
8752 printa(file); printa(file);
8753 printa(file, 'THE FOLLOWING TASKS ARE WAITING FOR ACCESS TO ' +
8754         'ENTRIES : ');
8755 printa(file, 72 * '-' );
8756 printa(file);
8757 count := 1;
```

```
8758
8759 (for server_task ∈ domain WAITING_TASKS )
8760    s_task := FULL_TASKNAME_MAP(server_task);
8761   (for buzy_entry ∈ domain WAITING_TASKS(server_task) )
8762       b_entry := UNQUALED_NAME(buzy_entry(1)) +
8763             ' #' + str(buzy_entry(2));
8764       (for [queued_task,-] ∈ WAITING_TASKS(server_task)(buzy_entry) )
8765          q_task := FULL_TASKNAME_MAP(queued_task);
8766          PRINT_TASKING_INFO( file, [ 'TASK ', q_task,
8767          ' IS QUEUED ON ENTRY ', b_entry, ' OF TASK ', s_task
8768                                                ] );
8769          if exists [ waiting_master, exname, place ]
8770             ∈ TASKS_WITH_RAISED_EXC |
8771                waiting_master = MASTER(queued_task)
8772          then PRINT_TASKING_INFO( file, ['MASTER ',
8773            FULL_TASKNAME_MAP(waiting_master),
8774                  ' IS HELD FROM ', 'RAISING EXCEPTION ',
8775             exname, ' IN ', place ] );
8776
8777          end if;
8778
8779          -- The next several lines of code are purely for output-
8780          -- formatting purposes.
8781
8782          current_triple := [q_task, b_entry, s_task];
8783          if present(previous_triple) then
8784            if previous_triple(3) /= s_task then
8785               printa(file, 14 * '*    ');
8786                  printa(file);
8787            elseif previous_triple(2) /= b_entry
8788              then printa(file);
8789            end if;
8790          end if;
8791          previous_triple := current_triple;
8792          count +:= 1;
8793       end;
8794    end;
8795 end;
8796
8797 if count = 1 then printa(file, '( NONE )' ); end if;
8798 printa(file);
8799 printa(file, 'THE FOLLOWING TASKS ARE WAITING FOR CALL ON ENTRIES :');
8800 printa(file, 72 * '-' );
8801 printa(file);
8802 count := 1;
8803
8804 loop for [waiting_task, set_of_entries] ∈ OPEN_ENTRIES do
8805    w_task := FULL_TASKNAME_MAP(waiting_task);
8806    loop for [open_entry, -] ∈ set_of_entries do
```

```
8807        o_entry := UNQUALED_NAME(open_entry(1)) +
8808             ' #' + str(open_entry(2));
8809        PRINT_TASKING_INFO( file, ['TASK ', w_task, ' HAS ENTRY ',
8810         o_entry, ' OPEN FOR RENDEZVOUS' ] );
8811
8812
8813        if exists [ waiting_master, exname, place ]
8814           ∈ TASKS_WITH_RAISED_EXC |
8815              waiting_master = MASTER(waiting_task)
8816        then PRINT_TASKING_INFO( file,
8817          ['MASTER ', FULL_TASKNAME_MAP(waiting_master),
8818           ' IS HELD FROM ', 'RAISING EXCEPTION ',
8819              exname, ' IN ', place ] );
8820        end if;
8821        count +:= 1;
8822     end loop;
8823     printa(file);
8824  end loop;
8825
8826 if count = 1 then printa(file, '( NONE )' ); end if;
8827 end proc DUMP_TASKING_INFO;
8828
```

PRINT_TASKING_INFO

```
8830 proc PRINT_TASKING_INFO(file, tuple_of_strings);
8831
8832 -- This procedure is used for formatting error messages issued by
8833 -- DUMP_TASKING_INFO.
8834
8835 next_line := '';
8836 count_of_lines := 1;
8837
8838 loop for next_str ∈ tuple_of_strings do
8839     if absent(next_str) then next_str := MAINTASK--'; end if;
8840     if (#next_line + #next_str) <= 70 then
8841           next_line +:= next_str;
8842     else
8843           printa(file, next_line);
8844          next_line := next_str;
8845           count_of_lines +:= 1;
8846     end if;
8847 end loop;
8848
8849 if count_of_lines = 1 then
8850           printa(file, next_line);
8851 else
8852           printa(file, lpad(next_line, 70));
8853 end if;
```

```
8854
8855 end proc PRINT_TASKING_INFO;
8856
8857
8858 ----------
8859
8860 drop find, top;
8861 drop absent, empty, nonempty, present;      --
8862 drop location, opcode, type_mark;   --
8863 drop is_access_iobject, is_access_ivalue, is_ais_stmt, is_array_iobject,
8864      is_array_ivalue, is_constrained_type, is_discr_ref,        --
8865      is_fixed_iobject, is_fixed_ivalue, is_instruction,
8866      is_label_iobject, is_label_ivalue, is_literal_ivalue, is_location,
8867      is_operation, is_proc_iobject, is_proc_ivalue, is_range,
8868      is_record_iobject, is_record_ivalue, is_simple_name,
8869      is_task_iobject, is_task_ivalue, is_uninitialized;
8870 drop cond_schedule, convert_duration, get_bounds_of;
8871 drop fix_abs, fix_add, fix_div, fix_eql, fix_exp, fix_fri, fix_frr,
8872      fix_frs, fix_geq, fix_gtr, fix_leq, fix_lss, fix_mul, fix_neq,
8873      fix_sub, fix_toi, fix_tor, fix_tos, fix_umin;
8874 drop test, boolean_false, boolean_true;
8875 drop EMAP, ENVSTACK, HANDLER, STSQ, TASKS_DECLARED, VALSTACK;
8876 drop EMAPT, HANDLERT, HEIGHT, STSQT, TASKS_DECLAREDT, VALSTACKT;
8877 drop POP_ENVSTACK, PUSH_ENVSTACK;
8878 drop TOP_VALSTACK, push, pop;
8879 --
8880 --
8881 end module ada - interpreter;
```