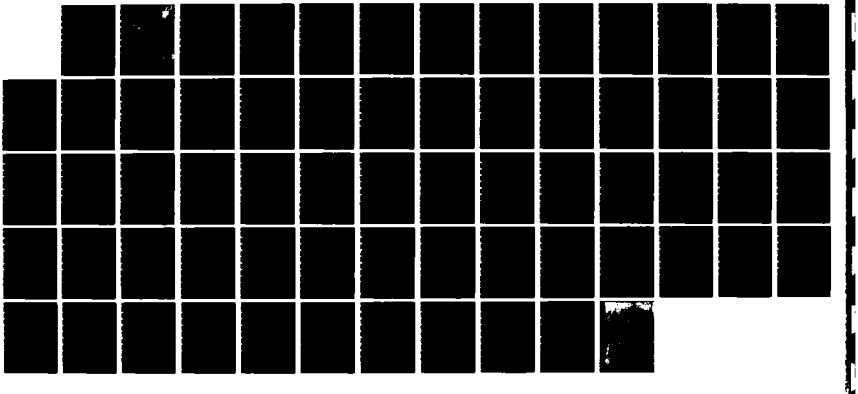
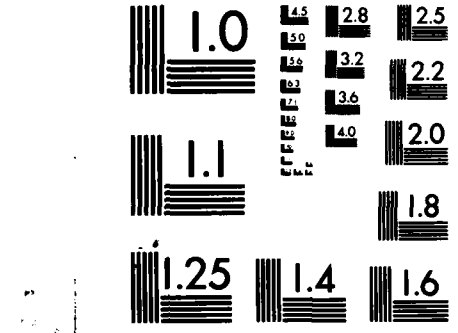
AD-A140 452 ANNOTATION LANGUAGE DESIGN FOR ADA (ANNA)(U) STANFORD 1/1
UNIV CA COMPUTER SYSTEMS LAB D C LUCKHAM JAN 84
RADC-TR-83-298 F30602-80-C-0022

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-83-298
Interim Report
January 1984

ANNOTATION LANGUAGE DESIGN FOR ADA (ANNA)

Stanford University

Dr. David C. Luckham

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-298 has been reviewed and is approved for publication.

APPROVED: Richard M. Evans

RICHARD M. EVANS
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Acting Technical Director
Command and Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-83-298	2. GOVT ACCESSION NO.: AD-A140 952
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): ANNOTATION LANGUAGE DESIGN FOR ADA (ANNA)	5. TYPE OF REPORT & PERIOD COVERED: Interim Report Oct 1980 - Sep 1981
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Dr. David C. Luckham	8. CONTRACT OR GRANT NUMBER(s): F30602-80-C-0022
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Stanford University, Computer Systems Laboratory, Stanford CA 94305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55811907
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (COES), Griffiss AFB NY 13441	12. REPORT DATE: January 1984
	13. NUMBER OF PAGES: 66
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Richard M. Evans (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming Language	Formal Annotation Language
Ada Programming Support Environment	Program Documentation
Assertions	Program Verification
Error Detection	Proof of Correctness
First Order Logic	Specification Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This interim report covers research work on Annotation language design for Ada undertaken during October 1980 - September 1981. The major goal of this research is the design and development of programming tools that may be incorporated into an Ada Programming Support Environment during the mid-1980 time frame. Since Ada is a very advanced language containing many essential new features such as tasking, and standard Ada tools such as compilers do not yet exist, our research has been structured so as to

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

approach the major goal b first studying the error detection problem for subsets of Ada corresponding to already highly used languages such as Pascal. The error detection problem is an important starting point because this attempts to analyze programs for common errors without assuming that the programs have accompanying annotations. At the start of this project no formal annotation language for Ada existed. This phase of our research effort was to design an annotation language for Ada, called ANNA. This would provide a basis for verification of Ada programs in general. The report gives a preliminary manual for ANNA.

Comments provide a natural and universally accepted method of describing the intended behavior of a program. In this report we propose a language, ANNA, which extends Ada by providing a formal comment facility. In ANNA formal comments are written with the same precision as programs, and included as an extension of Ada programs. Formal comments are either virtual Ada text or annotations. Since annotations have a well defined syntactical structure in ANNotated Ada, they can be processed by tools such as verifiers, optimizers, documentation systems and support tools for program development.

In making this proposal, we have had four principal considerations.

1. Constructing annotations should be easy for the Ada programmer, and should depend as much as possible on notation and concepts of Ada.
2. ANNA should possess language features that are widely used in the specification and documentation of programs.
3. Anna should provide a formal framework within which different theories of specifying programs may be applied to Ada.
4. Annotations should be equally well suited for different possible applications, not only for formal verification but also for specification of program parts during program design and development

The ANNA design requirements place heavy emphasis on developing the ways in which Anna can be used for specification and how it may be extended in the future. As a consequence of the choice of a first order annotation language, different theories and techniques of specifying programs may be applied using ANNA. For example, previous work on assertional specification of Pascal programs may be formulated in ANNA since any programming concept may be defined by the first order axiomatic method (axioms are simply stated as annotations) and used in annotations. It is also clear that the algebraic method of specifying abstract data types may be applied to packages in ANNA.

ANNA is incomplete, and may require future extensions. First, some possibly useful specification concepts are not provided. Consider for instance modal operators. These have to be defined axiomatically at the moment, but it may be useful to include them among the basic predefined operators in later versions. Secondly, ANNA does not include tasking. An extension to include task annotations may require the introduction of new predefined attributes, for example task type collections, and the semantics of task annotations will have to be defined.

ABSTRACT

This interim report covers research work on Annotation language design for ADA undertaken during October 1980 - September 1981. The major goal of this research is the design and development of programming tools that may be incorporated into an Ada Programming Support Environment during the mid-1980's time frame. Since Ada is a very advanced language containing many essential new features such as tasking, and standard Ada tools such as compilers do not yet exist, our research has been structured so as to approach the major goal by first studying the error detection problem for subsets of Ada corresponding to already highly used languages such as Pascal. The error detection problem is an important starting point because this attempts to analyse programs for common errors without assuming that the programs have accompanying annotations. At the start of this project no formal annotation language for Ada existed. The second phase of our research effort was to design an annotation language for Ada, called ANNA. This would provide a basis for verification of Ada programs in general.

Work on error detection and the RUNCHECK verifier is covered in the first Interim report for October 1979 - September 1980. This second report deals with our work on the design of ANNA.

Comments provide a natural and universally accepted method of describing the intended behaviour of a program. In this report we propose a language, Anna, which extends Ada by providing a formal comment facility. In Anna formal comments are written with the same precision as programs, and included as an extension of Ada programs. Formal comments are either virtual Ada text or annotations. Since annotations have a well-

defined syntactical structure in ANNotated Ada, they can be processed by tools such as verifiers, optimizers, documentation systems and support tools for program development.

In making this proposal, we have had four principal considerations.

1. Constructing annotations should be easy for the Ada programmer, and should depend as much as possible on notation and concepts of Ada.
2. Anna should possess language features that are widely used in the specification and documentation of programs.
3. Anna should provide a formal framework within which different theories of specifying programs may be applied to Ada.
4. Annotations should be equally well suited for different possible applications, not only for formal verification but also for specification of program parts during program design and development.

The Anna design requirements place heavy emphasis on developing the ways in which Anna can be used for specification and how it may be extended in the future. As a consequence of the choice of a first order annotation language, different theories and techniques of specifying programs may be applied using Anna. For example, previous work on assertional specification of Pascal programs [Hoare 69, Hoare and Wirth 73, SVG 79, Luckham 79] may be formulated in Anna since any programming concept may be defined by the first order axiomatic method (axioms are simply stated as annotations) and used in annotations. It is also clear that the algebraic method of specifying abstract data types may be applied to

packages in Anna.

Anna is incomplete, and may require future extensions. First, some possibly useful specification concepts are not provided. Consider for instance modal operators. These have to be defined axiomatically at the moment, but it may be useful to include them among the basic predefined operators in later versions. Secondly, Anna does not include tasking. An extension to include task annotations may require the introduction of new predefined attributes, for example task type collections, and the semantics of task annotations will have to be defined.

Concurrently with design work on Anna, further work on error detection has continued. This work has been concerned mainly with liveness errors in tasking. Since specifications for tasks are temporarily omitted from ANNA, this aspect of error detection was given special priority. This work is currently in progress, as is the Anna design, and will be covered in our final report.

The report gives a preliminary manual for ANNA. Currently, a guide on the use of ANNA in specifying and annotating Ada program is being written.

Table of Contents

1. INTRODUCTION

Comments provide a natural and universally accepted method of describing the intended behaviour of a program. In this paper we propose a language, Anna, which extends Ada by providing a formal comment facility. In Anna formal comments are written with the same precision as programs, and included as an extension of Ada programs. Formal comments are either virtual Ada text or annotations.

Since annotations have a well-defined syntactical structure in ANNotated Ada, they can be processed by tools such as verifiers, optimizers, documentation systems and support tools for program development.

In making this proposal, we have had four principal considerations.

1. Constructing annotations should be easy for the Ada programmer, and should depend as much as possible on notation and concepts of Ada.

2. Anna should possess language features that are widely used in the specification and documentation of programs.

3. Anna should provide a formal framework within which different theories of specifying programs may be applied to Ada.

4. Annotations should be equally well suited for different possible applications, not only for formal verification but also for specification of program parts during program design and development.

1.1 ADAPTION TO ADA

Goal (1) has had a major influence on both the syntax and semantics of Anna. The Anna syntax closely resembles that of Ada. Formal comments occur within the Ada comment frame work. Anna programs are therefore acceptable by Ada translators. Concepts in Anna are extensions of those in Ada. For example Boolean expressions are extended to allow quantification. Collections of access types are available in annotations using the attribute notation of Ada. The central specification concept in Anna, the declarative annotation, is a generalization of the constraint concept in Ada.

1.2 CONVENTIONAL SPECIFICATION TECHNIQUES

Goal (2) requires that the basic annotation language is a first order logic extending Ada Boolean expressions with quantifiers and implication operators. This choice is clearly dictated by the fact that most comments (informal or formal) are Boolean relationships between program variables. The inclusion of Ada text as formal comments - called virtual Ada text - now gives us a powerful comment facility without affecting the execution behavior of the underlying Ada program. For example specifications containing elements of a computation that are not manipulated by the program, e.g., history sequences of values of a variable, may be expressed in Anna by using both virtual Ada text and annotations.

However, this facility is not quite powerful enough. For example, the most successful method of specifying access variable manipulations requires that annotations may refer to objects that are not available in the programming language at all, i.e., Collections [Luckham and Suzuki 79]. (see also section 6.3.2. [Ichbiah et al. 79b].) Therefore access type collections and standard operations on them are added to Anna as predefined attributes; they may appear only in annotations. Similarly the concepts of state and sequence of state transitions are fundamental in the specification of packages, so Anna provides the basic sequence building operations in a notation adapted to Ada (see also [Luckham and Polak 80b]).

To allow the correctness proof of programs raising exceptions (see for example [Bron et al. 77]), propagation annotations are included in a notation adapted to Ada from [Luckham and Polak 80a].

1.3 FUTURE SPECIFICATION TECHNIQUES

Goal (3) is concerned with developing the ways in which Anna can be used for specification and how it may be extended in the future. As a consequence of the choice of a first order annotation language, different theories and techniques of specifying programs may be applied using Anna. For example, previous work on assertional specification of Pascal programs [Hoare 69, Hoare and Wirth 73, SVG 79, Luckham 79] may be formulated in Anna since any programming concept may be defined by the first order axiomatic method (axioms are simply stated as annotations) and used in annotations. It is also clear that the algebraic method of specifying abstract data types may be applied to packages in Anna.

Anna is incomplete, and may require future extensions. First, some possibly useful specification concepts are not provided. Consider for instance modal operators. These have to be defined axiomatically at the moment, but it may be useful to include them among the basic predefined operators in later versions. Secondly, Anna does not include tasking. An extension to include task annotations may require the introduction of new predefined attributes, for example task type collections, and the semantics of task annotations will have to be defined.

1.4 SPECIFICATION DURING PROGRAM DEVELOPMENT

Goal (4) expresses our concern for a wide applicability of the language. Use of Anna should not be restricted to only verification of existing programs in the conventional sense. Anna is also suitable for the formal specification of subprograms and packages during program development at a stage where respective bodies are not yet available. Such specifications may be used to simulate interfaces at the development stage. They will provide the basis for a proof of correct use of a subprogram or package independent of and prior to implementation, as well as a proof of correct implementation, (i.e., consistency of the body with the specifications). A specification will accompany a program through all stages of its development, for instance by successive transformation (see [Bauer et al. 78]), and may even be generated as part of the development process (see [Broy and Krieg-Bruckner 80]).

1.5 STRUCTURE OF THE REPORT

This Report is structured in the same way as the Ada Reference Manual [ARM, Ichbiah et al.80; see also 79a,b]. It should be read as an extension to that document, informally describing the lexical elements, syntax and semantics of Anna.

The syntax of Anna is an extension of the syntax of Ada; the syntax rules denote extensions or replacements of rules in Ada with the same name. An Anna rule is often an Ada rule with additional clauses; in this case three dots enclose references to the corresponding Ada rules. It is understood that appropriate substitutions are performed. Consider for example

```
expression ::=
        . . . ada_expression . . . |
          quantifier domain {; domain } => boolean_expression
```

In this Anna syntax rule, the right hand side of the Ada syntax rule for expressions should be considered to be substituted at the position of . . . *ada_expression* . . . Now "*boolean*_expression" with the italic identifier, *boolean*, refers to the Anna syntax rule; any occurence of expression has the extended meaning of Anna. Note, however, that the Anna syntax only applies to annotations and not to the underlying Ada program.

The separation of virtual Ada text and annotations in Anna from the underlying Ada program by comment prefixes is not represented in the syntax of Anna, but is nevertheless lexically enforced to make every legal Anna program a legal Ada program.

2. LEXICAL ELEMENTS

The lexical elements and conventions of Anna are those of Ada, with the extensions described in this chapter.

2.1 CHARACTER SET

The following additional characters may be used in an annotation:

(g) Additional special characters

¬ → ∃ ∀ ◆ ≠ ≤ ≥

2.2 LEXICAL UNITS AND SPACING CONVENTIONS

The following compound symbols may be used in annotations:

-> <->

2.3 - 2.6 No additions.

2.7 FORMAL COMMENTS

Anna permits two kinds of formal comments, virtual Ada text and annotations.

2.7.1 VIRTUAL ADA TEXT

A comment starting with a double hyphen followed by a colon is called a virtual Ada text and may be referred to only in formal comments. A virtual Ada text is regarded as a comment in an Ada program. In an Anna program, however, it is considered to be part of the program text and must satisfy the lexical, syntactic and semantic rules of Ada with one exception: bodies of virtual subprograms and packages need not be supplied.

Example of a virtual variable and a virtual function:

```
--:  GHOSTX : INTEGER;
--:  function LENGTH return INTEGER range 0 .. SIZE;
--   Ghostx and Length may only be referred to in formal comments
```

2.7.2 ANNOTATIONS

A comment starting with a double hyphen followed by a vertical bar (or exclamation mark) is called an *annotation*. In an Anna program, such an annotation must satisfy the lexical, syntactic and semantic rules of Anna.

An annotation is a quantified Boolean expression -- a Boolean expression that may contain quantifiers (section 4.11). A variable that is quantified is called a *logical* variable; its scope of visibility is the scope of the quantifier in the annotation. A variable that is not quantified is said to be a *free* variable of the annotation. A free variable must be a *program variable* -- i.e. a variable declared in the Anna text in which the annotation appears, and to which the normal Ada scope rules apply. Program variables may be either arguments or parameters of an annotation, depending on context. Parameters are evaluated when the annotation is elaborated.

Annotations may be generic and are instantiated according to the Ada rules.

Every annotation has a scope of application that is a region of program text for which it has a meaning. The scope of an annotation is determined according to the Ada scope rules from its position in the text.

The syntax and informal semantics of annotations in Anna are described in Chapters 3 - 12 of this document.

Examples of annotations:

```
subtype EVEN is INTEGER;
  --| where for all X:EVEN => X mod 2 = 0;
  -- all values of subtype EVEN must be divisible by 2.
  -- X is a logical variable.
  -- Note that this annotation cannot be expressed as an Ada constraint.

  MAX  :  INTEGER;
  --| 0 ≤ MAX ≤ 100;
  -- MAX is a program variable argument of the annotation (section 3.2).
  -- the values of MAX must satisfy the annotation throughout its scope.

  type FUNNY is array (1..MAX) of INTEGER;
  -- | where for all A  :  FUNNY; I: INTEGER range 1..MAX =>
                                  I mod 2 = Ø → A(I) ≤ MAX;
  -- MAX is a program variable in the annotation. Since this is a type constraint
  -- (section 3.3) MAX is a parameter of the annotation. All occurrences of MAX
  -- in the annotation are elaborated in the same way as in the type declaration.
  -- The values of FUNNY are then constrained to satisfy the elaborated annotation.

function IS_EVEN (Y:INTEGER) return BOOLEAN;
  --| return Y in EVEN;
  -- the values returned by IS_EVEN must satisfy the constraint on the subtype, EVEN.
  -- Y is a program variable argument of the annotation.

function IS_PRIME (P:NATURAL) return BOOLEAN;
  --| return not exist X,Y:NATURAL =>
  --|       (X > 1 and Y > 1)→ X * Y = P
```

```
-- X and Y are logical variables in the return annotation defining primeness
-- which constrains the values returned by IS_PRIME. P is a program variable
-- argument and is evaluated at each call.
```

2.7.3 POSITION OF FORMAL COMMENTS

The legal position of formal comments is restricted by the syntax of Anna. Formal comments may not appear in an Anna program everywhere an Ada comment may appear. For example, virtual Ada text may appear only where it would be legal in Ada if the comment delimiters preceding it were removed. Similarly, a declarative annotation may only appear in an Ada declarative part; a statement annotation only in a statement position.

The syntactic description of Anna omits `--:` and `--|`, it being understood that each line of an Anna formal comment must start with one of these prefixes. Thus a formal comment that runs over several lines must start with a new comment prefix on each continuation line.

Example of an annotation extending over more than one line:

```
procedure PUSH (E : in Elem);
  --| raise OVERFLOW => STACK.LENGTH() = SIZE
  --|          and [STACK;PUSH(E)] = STACK;
```

2.8 No addition.

2.9 RESERVED WORDS

The following additional words are reserved in annotations:

exist **that** **requires** **where** **axiom**

2.10 TRANSLITERATION

The following additional replacements are always allowed for characters that may not be available:

¦	!
¬	not
∧	and
∨	or
→	->
↔	<->
∃	exist
∀	for all
≠	/=
≤	<=
≥	>=

3. DECLARATIVE ANNOTATIONS

Declarative annotations in Anna are annotations placed in a declarative part. They extend the Ada concept of type constraint to other forms of declarations. However, more general Boolean expressions are permitted in declarative annotations than Ada constraints.

Declarative annotations are constraints over their scope of visibility with one exception. The Anna package axiom declaration, permits annotations that are axiomatic to be associated with declarations of packages. Axioms are guaranteed to be satisified by all entities declared in the visible part of the package.

In this chapter we deal with annotations on object, type and subtype declarations. Declarative annotations on subprograms are described in chapter 6, and on packages in chapter 7.

3.1 SYNTAX AND SCOPE OF DECLARATIVE ANNOTATIONS

declaration :: = . . . *ada_declaration* . . .
| *declarative*_annotation;

declarative_annotation :: = *boolean*_expression |
*type*_constraint |
*subprogram*_constraint |
*axiomatic*_annotation
type_constraint :: = **where** *quantified*_ expression
subprogram_constraint :: = **where** *boolean*_ expression
axiomatic_annotation :: = **axiom** *quantified*_expression

The scope of a declarative annotation is the same as for a declaration at the same position. Thus a declarative annotation in an enclosed scope only holds in that scope.

A declarative annotation in Anna that is not a package axiom is an Anna constraint. This is a generalization of the concept of a constraint in Ada. An Ada constraint can be expressed as an annotation: for example,

```
INDEX : INTEGER range  0 .. SIZE;
```

is equivalent in Anna to

```
INDEX : INTEGER;    --| 0 ≤ INDEX ≤ in SIZE;
```

On the other hand, the generality of annotations permits the expression of a wider variety of programming concepts and design intentions, even non-constructive or undetermined ones.

Example of an Anna constraint not expressible as an Ada type constraint:

```
subtype ODD_PRIME is INTEGER;
--| where for all X : ODD_PRIME => ODD(X) and PRIME(X);
--ODD and PRIME are previously declared boolean valued functions.
```

Example of an annotation whose truth is undetermined:

```
subtype FERMAT is INTEGER;
--| where for all F : FERMAT =>
--|       exist X, Y, Z : NATURAL =>
--|              X ** F + Y ** F = Z ** F;
```

Declarative annotations are elaborated according to the Ada rules for elaboration of declarations with some minor extensions:

1. Variables bound by quantifiers are not elaborated.

2. Program variables (i.e. unquantified variables) in type annotations and package axioms are treated as parameters of the constraint and are elaborated Program variables appearing in any other kind of annotations (e.g., object constraints) are treated as the names of constrained objects and therefore are not elaborated.

3. Variables with mode in are elaborated.

4. An expression containing a virtual function is not elaborated if a body is not supplied for that function.

5. Elaboration of objects and type annotations takes place when elaboration of Ada text at the same position would take place. Elaboration of annotations in a subprogram specification takes place when that specification part is elaborated except that in mode expressions are not elaborated then (see section 6).

Notes:

1. An annotation may not be computable in general since it may be an arbitrarily quantified boolean expression in Anna. Checking of the correctness of such annotations must rely on proof-theoretic methods.

2. Parameters of an annotation may not be computed at elaboration if they depend on values of virtual functions. Instead their values will be deduced by proof methods from the annotations of the virtual functions.

3.2 ANNOTATIONS ON OBJECT DECLARATIONS

A declarative annotation on an object is a boolean expression appearing in a declarative part where the object is visible such that the object occurs as a free variable in the expression. The annotation constrains the values of the object throughout the scope of the annotation.

In general, for a declarative annotation on an object, X

```
X : T;        --| C(X);
```

any use of X is conceptually equivalent to a call CHECK(X), where

```
function CHECK(A : T) return T is
begin
   if C(A) then
      return A;
   else
      raise CONSTRAINT_ERROR;
   end if;
end;
```

except that the predefined exception CONSTRAINT_ERROR will not be raised.

Examples of object annotations:

```
LIMIT  : INTEGER := 10_000;
   --| - 2**in N ≤ LIMIT ≤ 2**in N;
   -- where N is a visible variable at this point; any value of LIMIT must satisfy
   -- the inequality bounds for the value of N when the constraint is elaborated.

generic
        SIZE : INTEGER;
package DATA is
        BOUND : INTEGER;  --| 0 < BOUND < INTEGER'LAST mod SIZE;
            . . .
end DATA;
   -- the constraint on BOUND is generic and is instantiated by the actual value of SIZE.
   -- In any instantiation of DATA all values of BOUND must obey the actual constraint.


--: function F (X : INTEGER) return INTEGER;
--| return that Y => PRIME(Y) and 2**X < Y and
--|  for all Z : INTEGER => PRIME(Z) and 2**X < Z → Y ≤ Z;

X : INTEGER; --| X < F(in  X);
-- the parameter F(in X) is not elaborated since F has no body; its value
-- must be inferred from the specifications for F.
```

If a declarative object annotation in Anna involves several program variables, none of which has the mode in, then it must hold for all of these variables.

Example of a declarative annotation on two variables:

```
M, N : INTEGER := 0;     --| N ≤ M;

-- the value of N is constrained to be less than or equal to the
-- value of M throughout the scope of the declaration
```

3.3 ANNOTATIONS ON TYPE AND SUBTYPE DECLARATIONS

A declarative annotation may be used to constrain a type or subtype declaration. Such an annotation must immediately follow the constrained declaration and is bound to that declaration by the reserved word, **where**. Access types may not be constrained.

A type (subtype) constraint is a quantified expression preceeded by **where** and a single universal quantifier over that type or subtype. All other quantifiers in a type constraint must be over types that have previous complete declarations.

After elaboration a type (subtype) constraint must be a closed annotation --i.e., all variables are quantified. Any program variable in a type (subtype) constraint is treated as a parameter and is elaborated. Thus all program variables in type constraints have the default in mode (section 4.12).

Type constraints have the form, (see Section 4.11):

```
type_constraint ::= where quantifier identifier : subtype_indication =>
                                    quantified_expression
```

A type constraint is interpreted as restricting the domain of values of that type. Thus if we have,

```
type T is T';
    where for all X:T => C(X);
```

then the values of type T are constrained to be in the set, {X | X in T' and C(X)}. Such a constraint will be consistent with the underlying program if, within its scope, the constraint is true of the values of all objects declared of type T, all values of parameters of type T of procedure calls, and all values of type T returned by functions.

Examples of an Anna type constraints:

```
type DATE is
        record
            DAY     : INTEGER range 1 . . 31;
            MONTH   :  MONTH_NAME;
            YEAR    : INTEGER range 1 . . 4000;
        end record;

--| where for all X : DATE => X.MONTH = "FEBRUARY" → X.DAY <= 29;

type MAT is array (1 . . N, 1 . . N) of REAL;
subtype DIAGONAL_MAT is MAT;
--| where for all M : DIAGONAL_MAT; I,J : NATURAL =>
--|             not I = J → M(I,J) = 0.0 and
--|             I = J → M(I,J) ≤ X;
-- X is a global program variable and is elaborated as a parameter of the
-- subtype annotation.
```

Notes

1. A type constraint also constrains the domain of values of the type in annotations.

2. The leading **where** binds a type constraint to the preceding type (subtype) declaration and avoids ambiguity with constraints on global objects.

3. All free variables in a type constraint are treated as in parameters and are elaborated (section 4.12).

4. Since the only operations on access values are allocation, assignment, and equality test, the ability to constrain such values is trivial. So constraints on access types are not permitted.

5. A type constraint is equivalent to a set of declarative constraints placed within its scope on objects and subprograms.

Successive annotations on types and subtypes are equivalent to conjunctions of constraints and therefore describe intersections of subsets of values of the type. A union of subsets can be obtained by disjunction of constraints in a type or subtype annotation.

Examples of successive subtype annotations:

```
subtype NATURAL is INTEGER;
      --| where for all X:NATURAL => X ≥ 1;
subtype EVEN is INTEGER;
      --| where for all X:EVEN => X mod 2 = 0;
subtype EVEN_NATURAL is NATURAL;
      --| where for all X:EVEN_NATURAL => X mod 2 = 0;
subtype NATURAL_EVEN is INTEGER;
      --| where for all X:NATURAL_EVEN => X ≥ 1 and X mod 2 = 0;
 -- Even_Natural and Natural_Even are equivalent
subtype ZERO_INTEGER is INTEGER;
      --| where for all X:ZERO_INTEGER => X = 0;
subtype Positive_INTEGER is INTEGER;
      --| where for all X:POSITIVE_INTEGER =>
      --| X in ZERO_INTEGER or X in NATURAL;
```

Examples of type constraints with inner quantification:

```
subtype GREATER_ONE is NATURAL;
  --| where for all X : GREATER_ONE =>
  --|       exist Y : NATURAL =>  Y < X;
  --values of GREATER_ONE are constrained to be greater than 1

subtype SOME_INTEGER is INTEGER;
  --| where for all X : SOME_INTEGER =>
  --|       for all Y : INTEGER =>  X = Y;
  -- SOME_INTEGER is empty -- i.e., no values of INTEGER satisfy
-- the constraint.
```

*References: quantified_*expression, quantifier 4.11.

3.4 No addition.

3.5 No addition.

3.6 ANNOTATIONS ON ARRAY TYPES

A method of annotating programs containing array, record and pointer types in Pascal has been given in [Luckham & Suzuki 79]. This method is based upon the use in annotations of *Collections* of accessed values and predefined functions on arrays, records and Collections. It is applicable to Ada. Anna provides notational facilities to do this.

In Anna, new names (primary terms) are introduced that denote sequences of assignment operations on arrays and records. For access types, *Collections* and the predefined operations on collections are introduced as attributes of access types. Sequences of operations on collections are names (primary terms) in Anna.

3.6.1 - 3.6.3 No addition.

3.6.4 ARRAY OPERATION SEQUENCES

For every array type T the following new names (primary terms) of type T are defined:

```
array_operation_sequence  ::=  [array_name; array_store_operation
                                 {; array_store_operation}]
 array_store_operation    ::=  expression {, expression} => expression
```

For an array type T, the operation sequence, [A; I_1 ,...., I_n => C] where A is an object of type T, I_1 to I_n are index values for the n dimensions of T, and C is a value of the component type of T, denotes the value of A after the value of the component A(I_1 ,...., I_n) is replaced by C. Sequences with more than one array store operation denote values of A after the corrresponding sequence of changes to the values of its elements.

Examples of array operation sequences in annotations:

```
[MY_TABLE; 3 => 16]
-- denotes a value of MY_TABLE after MY_TABLE(3) := 16;

SPACE : ELEM_ARRAY;
   . . .
[SPACE; INDEX => E; INDEX' => F]
-- denotes a value of type ELEM_ARRAY.
```

Array operation sequences may be used as names in indexed components in Anna.

Example of an array operation sequence in an indexed component:

```
[MY_TABLE; 3 => 16](3)
-- = 16;
```

Note: Array operation sequences satisfy the standard axiomatic relationship with array selection, e.g., in the one dimensional case:

```
[A; I => E] (J) =  if I=J then E else A(J);
```

Example: Annotation of a generic SWAP procedure specification using an array sequence:

```
generic
      type ITEM is private;
      type VECTOR_RANGE is INTEGER range <>;
      type VECTOR is array(VECTOR_RANGE) of ITEM;
procedure SWAP ( A : in out VECTOR; I,J : VECTOR_RANGE);
      --| where out A = [in A; I => in A(J); J => in A(I)];
```

3.7 ANNOTATIONS ON RECORD TYPES

3.7.1 - 3.7.3 **No addition.**

3.7.4 RECORD OPERATION SEQUENCES

For every record type T the following new names (primary terms) of type T are defined:

```
record_operation_sequence  ::=  [record_name; record_store_operation
                                      {; record_store_operation}]
     record_store_operation  ::=  identifier => expression
```

For any record type, T, the operation sequence of length one, [R, I => C] where R is an object of type T, I is an identifier of a component, and C is a value of the corresponding component type, denotes the value of R after the value of the component R.I is replaced by C. I must not be the identifier of a discriminant, and R.I must exist for the corresponding variant. Sequences with more than one record store operation denote values of R after the corresponding sequence of changes have been made to its components.

Examples of record operation sequences in annotations:

```
[BIRTHDATE, YEAR => BIRTHDATE.YEAR +1]
   -- denotes BIRTHDATE with a change to the YEAR component.

PRINTER : PERIPHERAL;
  . . .

[PRINTER; LINE_COUNT => 1]
  -- denotes a value of PRINTER of type PERIPHERAL with LINE_COUNT set to 1;
```

Record operation sequences may be used as names in selected components.

Example of a record operation sequence in a selected component:

```
[PRINTER; LINE_COUNT => 1].LINE_COUNT = 1
```

Note: Record operation sequences satisfy the standard axiomatic relationship with record selection, e.g.,

```
[R; I => E].J =  if I=J then E else R.J;
```

3.8 ANNOTATIONS ON ACCESS TYPES

In annotations, an access type T has an associated package attribute, T'COLLECTION, that encapsulates the collection of allocated objects. The visible operations of T'COLLECTION correspond to the Ada constructs applicable to access type objects. These operations are also available in Anna as attributes of T.

Note: [ARM], section 3.8, refers to *Collections*: "The objects created by an allocator and designated by the values of an access type form a *collection* implicitly associated with the type." In Anna the collection is introduced explicitly for annotation.

3.8.1 ACCESS TYPE ATTRIBUTES

For every access type T, defined as,

type T is access S;

the following attributes are defined:

T'COLLECTION - the package *collection* of allocated objects of type S (see Appendix G). Notationally, it is treated as any other package in Anna (see section 7). The type of its states is denoted by T'TYPE; variables of this type may be declared and used in annotations. The name, T'COLLECTION in annotations is a variable whose value is the current state of the package, T'COLLECTION.

T'TYPE - the type of states of T'COLLECTION.

T'NULL - the **null** value of access type T, a visible constant of the T'COLLECTION package denoting the same value as S'(**null**) in Ada.

T'ALLOCATE(X : **out** T).
- a procedure attribute (i.e., a visible procedure of T'COLLECTION) corresponding to **new** S; the **out** value of X is the newly allocated value. Values for constraints are added if necessary.

T'ELEMENT (X : T) - a Boolean valued function attribute, returning TRUE if X has been allocated a value in T'COLLECTION and FALSE if it has not.

3.8.2 COLLECTION OPERATION SEQUENCES

For every access type T the following new names (primary terms) of type T'TYPE are defined:

```
collection_operation_sequence :: = [collection_name; collection_operation
                                    {; collection_operation}]
        collection_operation  :: = allocate_operation |
                                    selected_component => expression
          allocate_operation  :: = identifier'ALLOCATE (identifier)
            collection_name   :: = identifier
```

Collection operation sequences denote states of the collection.

Examples of Collection operation sequences:

```
[T'COLLECTION; T'ALLOCATE (X)]
-- the collection after an allocation, X := new S;
[T'COLLECTION; X => A]
-- the collection after, X.all := A;
[T'COLLECTION; T'ALLOCATE(X); X => A]
-- the collection after, X := new S(A); for expression A,
-- or after, X := new S.A; for aggregate A;
[C; X.F => E]
-- the collection state that results by starting in state C and performing
-- the assignment, X.F := E;
```

Collection operation sequences may be used as names in indexed and selected components. Semantically meaningful terms are obtained if states of T'COLLECTION are selected by function calls of the form, T'ELEMENT(X), or are indexed using objects of type T.

Examples:

```
[T'COLLECTION; ...].T'ELEMENT(X)
-- true if the value X has been allocated in the collection
[T'COLLECTION; ... ](X)
-- denotes the value of X.all
```

Example annotations on access variables

```
type T is access S;
X:T := new S(A);
Y:T := new S(B);
--| X =/ Y and T'COLLECTION(X) = A;

declare
    U, V : T;
begin
    U := new S(A);
  --| T'COLLECTION.T'ELEMENT(U) = TRUE and
  --| T'COLLECTION.T'ELEMENT(V)= FALSE;
    V := U;
    U.all := B;
```

```
  --| T'COLLECTION(V) = B;
end
  --| out T'COLLECTION =
  --|         [in T'COLLECTION; T'ALLOCATE(U); U => A; U => B];
```

Notes: The Collection of a derived type is the same as that of its parent type. Axioms for collections are given in Appendix G.

4. NAMES AND EXPRESSIONS IN ANNOTATIONS

4.1 NAMES IN ANNOTATIONS

The set of names in Ada is extended in Anna by special attributes, by selected components denoting result values of subprogram calls, and by operation sequences denoting package states.

```
name  ::=  ... ada_name ...
         | operation_sequence
```

4.1.1 - 4.1.2 No addition.

4.1.3 SELECTED COMPONENTS FOR RESULT VALUES

Components of the result value of a function call can be denoted as selected components in Ada. In Anna, the out values of a procedure call (that is final values of formal out or in out parameters) can be expressed as selected components of a call to the 'OUT function attribute of that procedure (see section 6 and Appendix A). In this case, the named parameter form should be used for the corresponding parameter association to increase readability. The actual parameter denotes the in value of the parameter association and can be expressed by an expression (in contrast to Ada, where it must denote a variable).

Examples of out values of procedure calls:

```
POP'OUT(E => X).E        -- final value of actual parameter X
SWAP'OUT (U => 3,V => 5).U       -- value = 5
```

4.1.4 ATTRIBUTES

An attibute in Anna can be a package state (which is a value in Anna) or the type of a package state (see section 7).

4.2 - 4.3 No addition.

4.4 EXTENDED EXPRESSIONS IN ANNOTATIONS

Expressions in annotations are extended by the usual logical implication operators and quantifiers, in values, out values, conditional expressions, and operation sequences (see 7.7.5).

```
expression ::=
        ... ada_expression ...
        | relation → relation
        | relation ↔relation
```

```
        | quantified_expression
        | conditional_expression

conditional_expression ::= if relation then expression
                                    else expression.

relation ::=

        ...ada_relation...
        |  simple_expression {relational_operator
                simple_expression}

primary ::=

        ... ada_primary...
        | in_value
        | out_value
        | operation_sequence
```

4.5 OPERATORS AND EXPRESSION EVALUATION

```
logical_operator ::= and | or | xor | → | ↔
```

The conditional control form, **if then else**, has the same precedence as the logical operators. As in Ada, operators at the same level are applied in textual order from left to right in the evaluation of expressions.

Note: The most general forms of expressions in annotations cannot always be evaluated.

4.5.1 LOGICAL OPERATORS

Operator	*Operation*	*Operand Type*	*Result Type*
→ ↔	implication	BOOLEAN	BOOLEAN

The implication operator → has the usual mathematical meaning. A ↔B is defined as (A → B) and (B → A).

In evaluating a conditional expression form, **if** A **then** B **else** C, the BOOLEAN relation A is first evaluated. If A is true then B is evaluated. If A is false then C is evaluated.

Examples of conditional expressions:

```
if T'COLLECTION.T'ELEMENT (X) then T'COLLECTION(X) = A else FALSE
-- this expression has a value when X has not been allocated
-- so that T'COLLECTION(X) does not have a value,
-- whereas T'COLLECTION.T'ELEMENT(X) → T'COLLECTION(X) = A
-- does not have a value.
```

```
[A;I=>E] (J) = if I = J then E else A(J)
-- the conditional expression is used here simply as a shorthand
-- for : I = J → E and not I = J → A(J), the two expressions
-- being equivalent when I=J, E, and A(J) are all defined.
```

Note: → and ↔ have the meaning of ≤ and = on boolean expressions, except that their precedence is lower to allow relations as subexpressions.

Example of an implication operator:

```
St1 = St2 ↔ St1.INDEX = St2.INDEX
--is equivalent to.
(St1 = St2 → St1.INDEX = St2.INDEX) and
(St1.INDEX = St2.INDEX → St1 = St2)
```

4.5.2 RELATIONAL OPERATORS

A sequence of relational operators is defined as a sequence of conjunctions in the usual mathematical way:

```
A op1 B op2 C = (A op1 B and B op2 C)
```

Example of a sequence of relational operators:

```
S ≤ SQRT(N) < S + 1
  --is equivalent to
S ≤ SQRT(N) and SQRT(N) < S + 1
```

4.6 - 4.10 No addition.

4.11 QUANTIFIED EXPRESSIONS

Quantified boolean expressions have the usual mathematical meaning in annotations.

```
quantified_expression ::= quantifier domain {;domain} => boolean_expression

        domain ::= identifier_list : subtype_indication
    quantifier ::= for all | [not] exist [that]
```

The variables in the domain of a quantifier are called *logical* variables and are said to be *bound* by that quantifier. Any variable in a quantified expression that is not bound by a quantifier is called *free*; it must be a program variable and must obey the usual Ada visibility rules. A quantified expression in which all variables are bound by quantifiers is called *closed*.

The scope of the quantified variables in the quantifier prefix is delimited at the left by => and extends over the subsequent boolean expression. Thus in an expression of the form

```
for all X:S => P(X) and exist Y:T => Q(X,Y)
```

the scope of X extends over both P(X) and Q(X,Y), whereas that of Y is restricted to Q(X,Y).

Intuitively, the quantified variables are interpreted as ranging over the set of values of the subtype_indication. Thus:

for all R:S => P(X) means "for all values of X of (sub)type S, P(X) is true";

exist Y:T => Q(X,Y)
means "there exists a value of Y of (sub)type T such that Q(X,Y) is true".

exist that X:S => P(X)
means "there exists a unique value of X such that P(X) is true", - i.e. the standard iota operator in formal logic:

Examples of quantified expressions:

```
for all X:NATURAL => X ≥ 1
for all N:NATURAL =>
            exist that S:NATURAL => S ≤ SQRT(N) < S + 1;
```

Note: Type constraints restrict the domain of values of a type declaration. Hence quantification in the scope defining the constraint is permitted only over previous complete type declarations (section 3.3).

4.12 IN VALUES AND OUT VALUES

The execution of a program can be formally modeled as a sequence of state transitions [FORM81]. At each transition, the values of program variables may change. In an annotation, the values of an expression containing program variables can be denoted relative to the initial state when the scope is entered, the final state when the scope is exited, or relative to all states while the computation is in the scope of the annotation. The initial or final values are denoted by prefixing the modes in or out to the expression.

```
primary ::= ...Ada_primary...
          | mode primary
  mode  ::= in | out
```

The in value of an expression E is written as in(E) and the in value of a variable V is written as in V. In an annotation this denotes the *initial* value of E or V upon entry to the scope of that annotation. Similarly the out value of expression E or variable V is written as out (E) or out V respectively. In an annotation, it denotes the *final* value of E or V upon exit from the scope of that annotation. The *current value* of an expression E is written as E. In an annotation, it denotes the value of the expression in each possible state that can be observed as a result of a state transition in that scope.

An in value is elaborated when the annotation is elaborated, except in the case of annotations of a subprogram specification.

The modes in and out may not be applied to logical variables -- i.e., variables bound by quantifiers in the annotation.

The application of modes to expressions must observe the following rules:

1. **out** may not appear in an expression prefixed by **in**.

2. If modes are nested in expressions, the innermost mode applies.

3. For each mode, ⊗, and any function identifier, F, ⊗(F(E)) means F(⊗(E)).

4. For each mode, ⊗, ⊗(**for all** Y : T => E means **for all** Y : T => ⊗(E) where ⊗ may not be applied to any occurrence of Y in E.

5. For each mode, ⊗, and any indexed component, A(I), ⊗(A(I)) means (⊗ A)(⊗(I)).

6. For each mode, ⊗, and any seclected component, R.X, ⊗(R.X) means (⊗ R).X.

Notes: In an annotation, constancy of a variable can be denoted by **in** V = V. If the scope of the annotation is a simple statement then V = **out** V.

Example of **in** *values in object annotations:*

```
X,Y:INTEGER;
--|X**2 + Y**2 ≤ in X**2 + in Y**2;
-- throughout the scope of X,Y, the sum of squares of their values is bounded by
-- by the sums of squares of their initial values
```

Example of **in** *and* **out** *values in object annotations:*

```
type COLOR is (RED, BLUE, YELLOW);
            . . .

 C : COLOR; --| in C ≥ C ≥ out C;
 -- all values of C must obey the constraint throughout the scope;
 -- Its initial value must be its largest, and the final value its smallest.

 X:INTEGER;      --| out X ≤ X;

 -- A consequence is out X ≤ in X;
```

Example of a type constraint with an **in** *parameter:*

```
MAX : INTEGER
type FUNNY is array (1..MAX) of INTEGER;
-- | where for all A : FUNNY; I:INTEGER range 1..MAX=>
                        I mod 2 = 0 → A(I) ≤ MAX;
-- all occurrences of MAX in the annotation are elaborated in the same way
-- as in the type declaration. Thus MAX is treated as having a default in mode.
```

Example of **in** *and* **out** *modes in a procedure specification:*

```
procedure SORT (A : in out INTEGER_VECTOR);
 --| where out(PERMUTATION (A, in A) and ORDERED (A, in A));
 -- the annotation means PERMUTATION (out A, in A) and
```

```
-- ORDERED (out A, in A);
```

4.13 DEFINEDNESS OF ANNOTATIONS

An Ada expression may not have a value, e.g., if it contains an uninitialized variable. Similarly, an annotation may or may not have a value (i.e., be defined). This generally depends on whether its subrelations and subexpressions have values. In particular its subrelations and subexpressions must be defined over all values in the domains of quantification of the logical variables occurring in them. Annotations that do not have a value are called undefined.

The rules defining the value of Anna quantified expressions extend those for evaluating Ada expressions as follows:

1. Ada operators are evaluated according to the Ada rules.

2. A term, f(X), in the case where f is a virtual Anna function without a body is defined if X has a value and the return expression in the result annotation of f has a value for the value of X.

3. A → B has a value if both A and B have values.

4. if A then B else C has a value if A has the value TRUE and B has a value, or A has the value FALSE and C has a value.

5. for all X : T => A(X), exist X : T => A(X) *have values if* A(X) *has a value for every* value of the (sub)type T.

Examples of values of quantified expressions:

```
for all X,Y : INTEGER => X DIV Y = X DIV Y;
-- is undefined since X DIV 0 is undefined.

for all X,Y : INTEGER => if Y= 0 then TRUE
                                 else X DIV Y = X DIV Y;
-- is defined since the conditional expression has a value for all
-- values of X and Y. Its value is TRUE.
```

Note: The classical interpretation of the logical connectives requires that expressions such as A or B, A → B, etc. do not have values if either A or B does not have a value. Since annotations extend the set of Ada expressions it is clear that A or B may be undefined. Conditional expressions are introduced into Anna to permit the programmer to construct annotations that are defined from expressions built up from partially defined functions.

5. STATEMENT ANNOTATIONS

A statement annotation is an annotation in a statement position.

```
statement ::= ...ada_statement...
              |statement_annotation

statement_annotation ::= boolean_expression
```

The scope of a statement annotation is the immediately preceding statement; the scope may therefore be empty.

A statement annotation is a constraint that must hold during the execution of the statement in its scope. It is equivalent to a declarative constraint in the declaration part of a virtual block whose body is the given statement. Elaboration of a statement annotation takes place immediately before execution of the statement in its scope.

A statement annotation,

```
--| C;
```

containing only in and out values (see 4.11) or else annotating a simple statement, is conceptually equivalent to a call ASSERT (C), where

```
procedure ASSERT(B:BOOLEAN) is
begin
    if B then
          return TRUE;
    else
          raise CONSTRAINT_ERROR;
    end if;
 end;
```

Examples of statement annotations:

```
X := 0;          --|X = 0;
X := X + 1;      --|X = in X + 1;
--note that in annotations of simple statements X = out X

if A(X) > A(X + 1) then
     Y := A(X + 1);
     A(X + 1) := A(X);
     A(X) := Y;
end if
--| out A(X) ≤ out A(X +1);
-- note that A(X) <= A(X + 1) is not always true, e.g., before or after
-- the first assignment, so the out-mode cannot be deleted.
```

Example of annotations of a loop:

```
while LOW < HIGH loop
      MID := (LOW + HIGH)/2;
```

```
        if X > A(MID) then
              LOW := MID + 1;
        else
              HIGH := MID;
        end if;
    end loop;

    --| in LOW <= LOW and HIGH <= in HIGH;
    --| LOW <= MID+1 and MID <= HIGH;

    --| ORDERED(A, in LOW, in HIGH) and
    --| (ISININTERVAL(X, A, in LOW, in HIGH) ->
         ISININTERVAL(X, A, LOW, HIGH))
    -- this constraint must be true of computation states
    -- throughout the execution of the loop body.
```

Notes:

1. In annotations on simple statements, a current variable X denotes out X.

2. A statement annotation inside a loop acts as an inductive assertion in the sense of the Hoare *invariant* [Hoare 69]: the assertion must hold each time the annotated statement is executed within the loop.

Reference: in and out values 4.11.

6. SUBPROGRAM ANNOTATIONS

Subprogram annotations are declarative annotations appearing in a subprogram specification. The scope of application of a subprogram annotation is the same as the subprogram declaration. Consequently a subprogram annotation must be true of every call to the subprogram and it must be true over the subprogram body.

6.1 ANNOTATIONS ON SUBPROGRAM DECLARATIONS

Annotations of subprogram declarations include constraints on formal parameters and results of computations, and conditions under which exceptions may be propagated. Annotations of parameter values and computation results are dealt with in this chapter; for propagation annotations see section 11.3.

```
subprogram_declaration ::=
                    ...ada_subprogram_declaration...
                      {where boolean_expression;}
                      [result_annotation;]
                      {propagation_annotation;}
```

The different kinds of annotations of a subprogram declaration permit statement of (i) annotations on formal parameters (either in the parameter list or by means of the **where** clause) that constrain parameter values on entry to, during execution of, and on exit from a subprogram, (ii) constraints on a returned function value (by means of a result annotation), and (iii) constraints governing propagation of exceptions.

The reserved word, **where** binds a *boolean*_expression annotation to a subprogram specification; this is a declarative constraint on the subprogram. It must be true of all calls (i.e., it has the semantics of a statement annotation on all calls), and it must be true of the implementation (i.e., it behaves as a declarative constraint in the body of the subprogram).

These annotations permit formulation of entry/exit conditions in general. The in value in V refers to the value of a variable upon entry to the subprogram, the out value **out** V refers to its value upon exit. A parameter annotation containing only in values is equivalent to an entry condition, i.e., a condition that must be true before every call (section 6.3). An annotation containing only in and out values of variables is equivalent to an exit condition. However, a parameter annotation containing a current value of a variable V is a stronger constraint since it holds throughout the execution of the body and implies corresponding entry and exit conditions.

Annotations of a subprogram declaration are elaborated when the declaration is elaborated. At that time the operations performed are exactly those performed in the Ada elaboration of the declaration that apply to the annotation - i.e., the elaboration of expressions that are legal in the underlying Ada declaration. The resulting annotation constrains both the subprogram body and all calls to the subprogram. As a constraint on the body it is then treated as an annotation in the declarative part of the subprogram body. The Anna elaboration of in mode expressions is performed when that declarative part is elaborated. The corresponding constraint on a call is obtained by the substitution of actual parameters for those formals of the subprogram specification that appear in the annotation.

Examples of annotations of subprogram declarations

```
procedure RIGHT_INDENT (MARGIN : out LINE_POSITION);
     --| where 1 ≤ out MARGIN ≤ 120;
     --an exit condition, see section 6.3.

function "/" (NUMERATOR, DENOMINATOR : INTEGER) return INTEGER;
       --| where DENOMINATOR ≠ 0;
       --| an entry condition.

procedure INCREMENT (X : in out INTEGER);
     --| where -2**36 < X < 2**36;
     -- a parameter constraint that must hold during execution.
     --| where out X = in X + 1;
     -- an exit condition.

function COMMON_PRIME (M,N : INTEGER) return INTEGER;
     --| return P : NATURAL => M mod P = 0 and
     --|         N mod P = 0 and IS_PRIME (P);
     -- a result annotation, see section 6.5.
```

The generic parameters of a generic subprogram may appear in its annotations. In this case the annotations are also generic and define a template for annotations obtained by applying the Ada rules for instantiation of generic subprograms. Each instance of the subprogram is annotated by the corresponding instance of the generic annotation.

Note: For an **in out** parameter X, the declarative parameter annotation

in X = out X

means that the value upon entry is the same as the value upon exit, whereas,

in X = X

means that throughout the subprogram body each value of X is the same as the value upon entry, i.e., the value of X remains constant.

6.2 FORMAL PARAMETER ANNOTATIONS

```
parameter_declaration ::=
                              ...ada_parameter_declaration...
                                | declarative_parameter_annotation

declarative_parameter_annotation ::= boolean_expression
```

Declarative parameter annotations may be given on individual parameters in the formal part as well as following the reserved word **where** subsequent to the subprogram specification. In either case the semantics is the same -- i.e., the constraining condition must hold for calls and throughout the scope of the body of a subprogram.

Objects declared globally to a subprogram declaration or body are treated as if they were additional parameters: constants correspond to additional **in** and variables to additional **in out** parameters.

Examples of formal parameter annotations:

```
procedure EXCHANGE (FROM, TO : in INDEX; --| FROM < TO);

procedure P (X: in INTEGER;   --|X > 0;
                Y,Z: in out INTEGER);
          --| where Y ≤ Z;

   --here X > 0 and Y ≤ Z must hold throughout the body,
   --and imply the entry assertion in X > 0 and in Y ≤ in Z
   --and the exit assertion out Y ≤ out Z
```

Note:

Annotations constraining parameter values may be placed in the parameter list or in the where clause; the choice is basically a matter of taste in maintaining the Ada formatting of the formal part of a subprogram specification.

6.3 ANNOTATIONS ON SUBPROGRAM BODIES

```
subprogram_body ::=
        ... Ada_subprogram_specification ...
                {boolean_expression;}
                [result_annotation;]
                {propagation_annotation;}
        is
                declarative_part
        begin
                sequence_of_statements
       [exception
                {exception_handler}]
        end [designator];
```

For a subprogram body, the semantics of annotations of the subprogram specification part of the body, including result and exception annotations, is the same as for annotations in a subprogram declaration. The syntax differs only in that the reserved word **where** may be omitted.

If a subprogram declaration and its body or stub occur in the same declarative part, the annotation of the specification parts of both must be identical. As in Ada, if a subprogram specification part with annotations is repeated, the second occurrence is never elaborated.

Example of an annotation of a subprogram body:

```
procedure SWAP (U,V: in out ELEM)
--| out U = in V and out V = in U;
is
        T : Elem := U;
begin
        U := V; V := T;
end SWAP;
```

Notes:

1. For a virtual Ada subprogram specification, a body need not be supplied.

2. The reserved word **where** may be omitted from annotations of the specification part of a subprogram body since no ambiguity with annotation of other units can result.

6.4 ANNOTATIONS ON SUBPROGRAM CALLS

Annotations on subprogram calls are a special case of statement annotations. A subprogram call, like any simple statement, is considered to be an atomic operation. Thus, a current value, X, in an annotation of a subprogram call denotes the final value, **out** X. Similarly, the values of an actual **in out** parameter on entry to and exit from a call can be constrained by a call annotation; its values during execution cannot be constrained by a call annotation.

Annotations of subprogram declarations impose constraints on the **in out** values of actual parameters of calls. The actual constraint on a call is obtained by substituting the actual parameters of the call for the formals in the declarative annotation.

Examples of annotations on subprogram calls:

```
SWAP (A, B);      --| out A = in B and out B = in A;
--constraint on a call imposed by the previous annotation on the body of SWAP

P (F, I, Z);      --| in I ≤ in F and out I ≤ out Z;
```

6.4.1 THE 'OUT ATTRIBUTE.

To facilitate construction of expressions involving the **out** values of procedure parameters resulting from calls, Anna associates a function attribute, 'OUT, with each procedure. If P is a procedure, P'OUT is a function returning a record whose components are the **out** values of the **in out** and **out** parameters of P. These values can be selected using the names of the formal parameters of P. The parameters to P'OUT are the same as those of P.

Examples of expressions using the 'OUT attributes to denote out parameter values:

```
procedure BINARY_SEARCH( X : ORDERED_ARRAY; K : KEY; R : out INDEX);
                .  .  .
--| A (BINARY_SEARCH'OUT(X => A, K => 10, R => PLACE).R) = 10;


procedure POPTOP (Y : out ITEM);
--| [STACK; PUSH(X)].POPTOP'OUT(Y).Y = X;
--the out value of Y after applying in sequence PUSH(X) and then
--POPTOP(Y) to a STACK package is denoted by the left side expression.
```

6.5 RESULT ANNOTATIONS FOR FUNCTION SUBPROGRAMS

```
result_annotation ::=
   return[[that] domain => ] expression
```

A result annotation specifies the result of a function. It is preceded by the reserved word **return** and will normally contain as free program variables the formal parameters of the function and its global variables. The quantifier **that** is expected to be useful in characterizing unique solutions to expressions.

Example of a result annotation:

```
function SQRT (N: NATURAL) return NATURAL;
     --| return that S: NATURAL => S**2 ≤ N < (S+1)**2;
```

6.6 EXAMPLES.

1. Dijkstra's Dutch National Flag program

```
-- CONCEPTS is a virtual generic package defining the annotation concepts
-- PERMUTATION and SWAPPED. It is instantiated for each context (i.e.
-- set of types). CONCEPTS does not have a body.

--: generic
--:     type INDEX is (<>);
--:     type ELEMENT is private;
--:     type GENERIC_ARRAY is array (INDEX) of ELEMENT;
--: package CONCEPTS is

--: function SWAPPED(A: GENERIC_ARRAY; I,J: INDEX) return GENERIC_ARRAY;

--: function PERM (A, B: GENERIC_ARRAY) return BOOLEAN;

--| axiom  for all A,B: GENERIC_ARRAY; I,J: INDEX =>

--|     SWAPPED(A,I,J)(J) = A(I)  ∧
--|     SWAPPED(A,I,J)(I) = A(J)  ∧
--|     (K≠I ∧ K≠J → SWAPPED (A,I,J)(K) = A(k)) ∧

--|     PERM(A,A)  ∧
--|     (PERM(A,B) → PERM(B,A))  ∧
--|     (PERM(A,B) ∧ PERM(B,C) → PERM(A,C))  ∧
--|     PERM(SWAPPED(A,I,J),A);

--: end CONCEPTS;

constant N: INTEGER := ...;
subtype INDEX is INTEGER range 1 .. N;
type COLOR is (RED,WHITE,BLUE);
type COLOR_ARRAY is array (INDEX) of COLOR;

--: package CONCEPT_INSTANCE is new CONCEPTS(INDEX,COLOR,COLOR_ARRAY);
--: use CONCEPT_INSTANCE;

--: function FINAL (C: COLOR; A: COLOR_ARRAY; I,J: INTEGER)
--:    return BOOLEAN;
--|          return  for all K in range I .. J-1 =>  A(K)=C;
```

```
function SWAP (A: COLOR_ARRAY; I,J: INTEGER) return COLOR_ARRAY;
        --| return  SWAPPED(A,I,J);

procedure DUTCH(A: in out COLOR_ARRAY;  N: INTEGER; I,J: out INTEGER)
        --| in  1 ≤ N;
        --| PERM(out A, in A);
        --| out ( 1≤I ∧ I≤J ∧ J≤N ∧ FINAL(BLUE,A,1,I)
        --| ∧ FINAL(WHITE,A,I,J) ∧ FINAL(RED,A,J+1,N+1));

                                    -- end of ANNA procedure specification

is

    K : INTEGER;  --|  K ≤ N+1;
    --| PERM(A,in A);
    -- A and K are constrained throughout the body.
begin
    I := 1;
    J := 1;
    K := N+1;

    loop  --| FINAL(BLUE,A,1,I) ∧ FINAL(WHITE,A,I,J) ∧
          --| FINAL(RED,A,K,N+1);                          -- see Note 1
        exit when J >= K;
        if A[J] = BLUE  then
            A := SWAP(A,I,J);
            J := J+1;
            I := I+1;
        elsif  A[J] = WHITE  then
            J := J+1;
        else                -- A[J] = RED,          see Note 2
            K := K-1;
            A := SWAP(A,J,K);
        end if;
    end loop;
    --|  1≤I ∧ I≤J ∧ J≤K;                                  -- see Note 3
end DUTCH;
```

Notes:

1. This assertion has the same semantics as a loop invariant in Hoare's sense.

2. This is an informal comment, but it could be written as an annotation.

3. A statement annotation of the loop statement (i.e., it holds everywhere within the loop.)

Each annotation in the body of DUTCH appears at the place where its scope is largest. These annotations are sufficient to prove the **out** specification of the procedure. Note also that the inequalities annotating the loop cannot be expressed as object constraints in the declarative part as I and J are out parameters that cannot be initialized except in the statement part. This is a case of ADA getting in the way of the most natural form of annotation.

Alternative definitions:

```
FINAL could be defined by:
--| return  if I≥J then TRUE
--|          else  (A[J-1]=C ∧ FINAL(C,A,I,J-1))
--|               ∨ (A[I]=C ∧ FINAL(C,A,I+1,J));

SWAPPED could be defined as:
--|      SWAPPED(A,I,J)(K)
--|                 =  if K=I then A(J)
--|                    elsif K=J then A(I)
--|                    else  A(K);
```

7. PACKAGE ANNOTATIONS

New annotation concepts are introduced to permit adequate annotation of packages and of programs that use packages.

The basic concept used in annotations of the visible part of a package or of a program using a package is the *package state*. Viewed from the outside, a package is treated conceptually as a composite object of some new (not defined in Ada) type. The values of this type are called *states*. In Anna, package states are denoted by identifiers and by sequences of operations. For each package the *type of its states* and the value of its *initial state* are attributes of that package. The name of a package may be used to denote its state in annotations.

A second fundamental concept is the *package axiom*. Axioms are annotations declared within the visible part using the reserved word, **axiom**. They express properties of the visible elements of a package that are guaranteed by the implementation in the package body. Axioms behave as *promises* outside of a package and may be assumed by all programs using the package. The package body is constrained by the axioms; that is, the implementation of the package must satisfy the package axioms.

This chapter describes the new annotation concepts for packages.

7.1 PACKAGE STRUCTURE

Package annotations may appear in the visible, private, and body parts of packages. For the purposes of annotation, the private part of a package should be considered as part of the package body. However, Anna maintains the Ada package structure, so annotations may appear separately in both parts. Annotations in the visible part of a package specification are called *visible annotations*; annotations in the private part or body are called *hidden annotations*. The information contained in the Ada package specification is only sufficient in general for correct syntactic use of the package. The purpose of visible annotations is to *complete* the description of a package. The information conveyed by the visible annotation should be sufficient to understand how the package functions and to make it unnecessary for a user to inspect the implementation in the private part and body. Visible annotations of a package specification have many possible uses. First, they provide information that may be used to classify the package (in a database say) independently of any implementation. Second, they can be used in determining correct interfacing either during bottom up program development when the implementation of a package should be concealed, or during top down development when an implementation is not yet available. Thirdly, during top down development, visible annotations defining the desired properties of a package provide a guide for the implementor; the package body must satisfy the visible annotations. Annotations of the private and body parts of a package have two purposes. Description of the intended behaviour of the implementation of the package in the package body, and definition of a representation of all concepts used in the visible annotation in terms of the local elements of the package body.

A generic package declaration defines a template for packages obtained by instantiation of that declaration. Annotations of a generic package may contain the generic parameters. Such annotations therefore define a template for package annotations obtained by applying the Ada generic package instantiation to them. Instances of a generic package are annotated by corresponding instances of the generic annotation.

For a virtual Ada package specification in Anna, a private part or body need not be supplied.

7.2 VISIBLE ANNOTATIONS IN PACKAGE SPECIFICATIONS

Annotations in the declaration part of a package specification are called *visible* annotations. They are declarative annotations and their scope of application is the scope of the package declaration. Visible annotations are constraints over their scope of application except in the case where they are declared as the axiom of the package (section 7.2.1).

Annotations on any declared item in a package specification may be included in that specification. These may be annotations of types, objects, or subprograms as before. They may also be annotations of exceptions (Section 11), generic parameters (Section 12), or other packages. An annotation of an individual item declared in the package specification has the same semantics in its scope of application as any annotation of the same kind of declaration; i.e., it has the semantics that would result if the name of the item were qualified by the package name and the package specification boundary was deleted.

Annotations in a package declaration are elaborated when the declaration is elaborated.

Generic formal parameters may appear in annotations. The formal parameters are replaced by the corresponding actual generic parameters in the generic instantiation at elaboration and must obey the Ada matching rules. An annotation in the generic part of a generic package declaration may be used to constrain the actual generic parameters in instances of the package.

Notes:

1. Annotations of array record and access types (sections 3.6 - 3.8) are special cases of annotations of the visible parts of packages.

2. To state visible properties, it is often necessary to introduce auxiliary operations, for example, by means of declarations in virtual Ada text. Such auxiliary operations cannot be used in the underlying Ada program; their only purpose is to aid in the annotation (see section 2.7.1).

3. Visible annotations may refer to individual items or may describe properties about the interrelation of visible items.

4. Visible annotations will normally include statements about package states, in particular, statements about sequences of package operations, since such statements provide a very powerful means of expressing properties of groups of visible items in the package specification (see section 7.7).

7.2.1 PACKAGE AXIOMS

In Anna an axiom may be declared in a package visible part. An axiom is a quantified boolean expression following the reserved word, **axiom**.

package_axiom ::= axiom *quantified*_Boolean_expression

After elaboration a package axiom must be a closed quantified expression not containing free program variables (section 4.11).

In general, an axiom will be a conjunction of Boolean expressions which are referred to as the package *axioms*. Package axioms express properties of the visible entities of the package that are guaranteed to hold true in the scope of visibility of its declaration. Thus, for example, in analyzing the correctness of a program that uses a package, the package axioms may be assumed. Package axioms constrain the body of the package; i.e., the implementation of the package must satisfy its axioms.

Example of axiomatic annotations for the package STANDARD:

```
package STANDARD is
            . . .

--| axiom
--| (for all A, B, N : INTEGER => A mod B = (A + N * B) mod B) and
--| (for all A, B : INTEGER =>
--|   A = (A/B)*B + (A rem B))    and
--|   (-A)/B = -(A/B) = A/(-B)       and
--|   A rem (-B) = A rem B     and
--|   (-A) rem B = -(A rem B))  and . . .;

            . . .

end STANDARD;
```

Example of generic axioms of a generic Stack package:

```
generic
    type ELEM is private;
    . . .
    package STACK is
    . . .

  --| axiom
  --|     for all St : STACK'TYPE; X, Y : ELEM =>
  --|          [St; PUSH(X); POP(Y)] = St and
  --|           St'INITIAL.LENGTH() = 0   and . . .;

    . . .
    end STACK;
```

Note: If axioms are to be given for a user-defined type, the type declaration must be encapsulated in a package. Generally this will be a private type declaration and the visible operations of the package will be the constructors and selectors of the type.

7.3 HIDDEN ANNOTATIONS IN PACKAGE BODIES

Hidden annotations of a package consist of the annotations in the private and body parts. Hidden annotations have two purposes.

1. To specify the intended behaviour of the implementation of the package. For this purpose annotations of types, objects, statements, subprograms, and packages are used to specify all entities declared within the body. These entities include both the bodies of entities declared in the visible part of the package and entities that are local to the body.

2. To specify the implementation of all virtual entities declared in the visible part of the package. Such entities will normally be declared for use in visible annotations. All visible entities must be redeclared in the package body; this includes both Ada entities and virtual Anna entities. Hidden annotations must specify the virtual entities even though bodies need not be supplied for virtual units. In particular a virtual private type must have a completed declaration either in the package body or private part.

Example of an auxiliary declaration for annotation of a STACK package:

```
--: function LENGTH return INTEGER range 0 .. MAXSTACK;
      --| return STACK.INDEX;
-- LENGTH is a virtual function used in visible annotations of STACK.
-- This specification with annotation is declared in STACK body where
-- the local variable, INDEX, is visible.
```

Notes:

1. Although Anna auxiliary operations must have annotated specifications in the package body, no explicit bodies need be supplied.

2. The package state type, 'TYPE, has a default type definition in the package body (section 7.7.4).

3. The declarations of private types and annotations of visible (actual and virtual) entities allows verification of the desired properties stated in the visible annotations in terms of the implementation chosen in the body.

7.4 ANNOTATIONS ON PRIVATE TYPES

Private types may be annotated by both type constraints and package axioms.

Constraints on private type declarations must obey the same restrictions as any type constraint and have the same semantics (section 3.3).

A constraint on a private type declaration in the visible part of a package will constrain only the optional discriminant and default values since no other structure is visible.

Private type annotations in a visible part will in general be axiomatic. These will be given using the

visible operations of the package, including virtual operations declared in the visible part for purposes of annotation. The axioms will define the properties of the allowable operations on the type. Thus the Ada package and private type constructs together with the Anna package axioms provide a powerful method of defining abstract data types independently of any implementation.

A constraint on a private type declaration in the private or body part is a type constraint over the package private and body parts. As such it constrains all values assigned to variables of the type within the hidden part of the package. As a consequence, parameter values and function return values of the type must obey the constraint for all subprograms whose bodies are in the package body. Since values assigned to variables of the type declared outside the package can only be constructed using the visible subprograms of the package, it follows that the values of outside variables will also obey the hidden constraint. Hence, such constraints may be assumed on entry to any visible subprogram.

Annotations on a **limited private** type declaration may use equality on that type (even though equality is not available in the Ada text). It must be remembered that equality in annotations always denotes logical identity - i.e., it obeys the substitutivity axiom.

Notes:

1. Visible annotations on private types are essentially abstract data type specifications.

2. Virtual private types declared in the visible part of a package may have their full declaration in either the private or body part.

3. Hidden constraints on private types generalize Hoare's concept of Monitor invariant (section 7.7).

Example of a visible axiomatic annotation of a private type

```
package KEY_MANAGER is
    type KEY is private;
    NULL_KEY : constant KEY;
    procedure GET_KEY ( K : out KEY);
    function "<" (X,Y : KEY) return BOOLEAN;
    --| axiom for all K : KEY => NULL_KEY < K or NULL_KEY = K;
        . . .
end KEY_MANAGER;
```

7.5 - 7.6 No addition.

7.7 PACKAGE STATES

The basic new concept introduced in Anna for use in package annotations is the *package state*. This concept is intended to facilitate writing external annotations, i.e., visible package annotations that express properties of the visible entities, and also annotations of programs that use the package.

Viewed from the outside a package is a composite object of some new (not defined in Ada) type. The values of this type are called package *states*. The type of the states is introduced as the 'TYPE attribute of a package. The structure of a package state is not visible from the outside. Thus a package state type behaves exactly as a private type exported by that package.

Whenever a call on a visible procedure of a package is performed, a change in the package state may result. States resulting from the execution of a sequence of procedure calls can be denoted in Anna by *operation sequences*. Operation sequences are terms in Anna.

7.7.1 STATE TYPES

The type of all states of a given package P is denoted by the attribute P'TYPE. It is called the *state type* of P. The state type behaves in Anna as a virtual private type declared in the visible part of P. It is introduced as an attribute so that the virtual declaration is omitted. It may be used in annotations and virtual program text exactly as any other type. If P is a generic package and Q is an instantiation of P, then annotations for Q are obtained from the annotations of P by applying the Ada generic instantiation rules and by replacing P by Q everywhere. Thus, if P is generic then "P" acts like a generic parameter in annotations. Equality is predefined on state types in Anna. However it may be redefined by a virtual declaration of " = " in the package body (see section 7.7.3).

Examples of annotations of state types:

```
for all St1,St2 : PLOTTING_DATA'TYPE => S1 = S2;
-- the PLOTTING_DATA package has no body so its
-- states are all trivially equal [ARM, 7.2].

for all S1,S2 : RATIONAL_NUMBERS'TYPE; X,Y : RATIONAL =>
        S1.EQUAL(X,Y) = S2.EQUAL(X,Y);
-- the value of the function EQUAL on any two rationals is independent
-- of the state of the rational numbers package [ARM, 7.3].

for all S : STACK'TYPE ; X, Y : ELEM =>
      [S ; PUSH(X) ; POP(Y)] = S
-- for any state S of a STACK package, if a PUSH operation
-- is performed followed by a POP, then the resulting state
-- is equal to S. Note that STACK is generic and the annotation is
-- a template for all instances.
```

7.7.2 CURRENT AND INITIAL STATES

The *current* and *initial* states of any package are new primary terms in Anna. The *current* state of a package at any point in a program is denoted by the name of the package. The *initial state* is denoted by the attribute, 'INITIAL. The initial state of a package is the state after the elaboration of its declaration (that is, after execution of its body, if any).

Examples of current states:

```
STANDARD            -- current state of the package STANDARD,
STACK_INSTANCE      -- current state of the package STACK_INSTANCE
```

Examples of initial states:

```
KEY_MANAGER'INITIAL \-- initial state of the KEY_MANAGER package [ARM, 7.4.1].
STACK'INITIAL\-- initial state of any instance of the generic STACK package;
```

7.7.3 STATE COMPONENTS

Inside a package body the state type may be treated as if it has an implementation as a record whose components are *all* the local objects of the package body. Here, in addition to *objects* in the Ada sense, the states of local packages are also included.

More precisely, let a:T_1, b;T_2... be the local objects declared in the body of package P. Let m, n, ... be the nongeneric packages declared in the body of P. Then P'TYPE may be treated as if in the declarative part of the body of P there is a type declaration of the form:

```
type P'TYPE is record
                a : T1;
                b : T2;
                  . . .
                m : m'TYPE;
                n : n'TYPE;
                  . . .
end record;
```

This Anna convention permits selection on states using the names of local objects. Selected components of states may appear in expressions wherever the component names are visible.

Examples of expressions involving internal state components:

```
KEY_MANAGER'INITIAL.LAST_KEY = 0
-- internally any state of the KEY_MANAGER package has a LAST_KEY
-- component; its initial value is 0 [ARM 7.4.1].

for all S : KEY_MANAGER'TYPE; K : KEY =>
    [S; GET(K)].LAST_KEY = S.LAST_KEY + 1;
-- application of the GET prodecure in any state of KEY_MANAGER
-- increments the LAST_KEY component of that state.

for all S : STACK'TYPE => 0 ≤ S.INDEX and S.INDEX ≤ SIZE;
-- an internal generic invariant on all states of all instances of STACK.
```

```
Equality on state types is predefined as the identity on P'TYPE.  It may be
redefined for a given package by a virtual declaration of the function "="
on state types in the package body.  Annotations of such a declaration must
imply that the "=" satisfies the axioms for equality
(see examples in section 7.8).
```

7.7.4 OPERATION SEQUENCES

A state transition is the effect of executing a statement or of an elaboration. A subset of all possible state transitions that may occur during execution of a program are those due to subprogram calls (including calls to functions with side effects on global variables). Those package states resulting from sequences of completed calls to visible subprograms of the package are given a special notation in Anna as operation sequence terms. This provides a convenient and powerful method for annotating the visible parts of packages.

An operation sequence in Anna denotes the state of a package after execution of a (sequence of) subprogram call(s) to that package.

```
operation_sequence ::=
        [state_name {; subprogram_call}]

state_name ::=
        identifier | operation_sequence
```

If the *state_name* in an operation sequence is of type M'TYPE then the following subprogram calls must be to visible subprograms of M and the resulting sequence has type M'TYPE. The unqualified names of subprograms may be used in operation sequences since no ambiguity can result.

Suppose P and Q are visible subprograms of M, and S is an operation sequence of type M'TYPE. The state of M immediately succeeding S after completing a call, P(...), is denoted by

[S ; P(...)].

The notation for operation sequences is defined by:

[S ; P ; Q] = [[S ; P] ; Q].

Examples of operation sequences:

```
[KEY_MANAGER; GET_KEY(A); GET_KEY(B)]
-- the state of KEY_MANAGER after two calls to GET_KEY.

[STACK; PUSH (E)]
[STACK; PUSH (X); POP (Y)]
```

Notes:

1. Techniques for specifying module constructs by means of sequences of operations have been previously suggested by Parnas and Bartussek [PB 77], Dahl [Da 78], and Luckham and Polak [LP 80].

2. An operation sequence denotes a state only under the assumption that all subprogram calls in it terminate and do not propagate exceptions; otherwise it is meaningless (section 4.13).

7.7.5 STATE RELATIVE RESULT VALUES

```
state_relative_result_value ::= state_name . primary
```

In package annotations it is necessary to denote the values of functions and procedure parameters resulting when package operations are applied in certain states. These values are said to be computed *relative* to the state of the package. They are denoted by selection on states. The result of applying a package function, f, in state, St, is denoted by St.f. Similarly the selection notation for the final values of procedure out parameters may be applied to a state using the 'OUT attribute of that procedure (see Section 4.1.3).

Examples of state oriented result values:

```
[St;PUSH(X)].POP'OUT(E=>Y).E = X
-- the final value of the actual out parameter Y of a call to POP
-- in state [St;PUSH (X)] is  X;  E is the formal out parameter
-- in the specification of POP.

NULL_KEY < [KEY_MANAGER].GET_KEY'OUT(K=>A).K
-- NULL_KEY is < the out value of A resulting from a
-- call to GET_KEY in the current state of the
-- KEY_MANAGER package.
```

Notes:

The selection notation, POP'OUT(E=>Y).E , denotes the final value of the actual out parameter, Y say, bound to the formal E resulting from the procedure call, POP(Y). Use of the named parameter notation with 'OUT functions is recommended for clarity (see section 4.1.3).

7.8 EXAMPLES OF PACKAGES WITH ANNOTATIONS

1. Example of a STACK package.

```
generic
      type ELEM is private;
      SIZE:NATURAL;
      --| SIZE > 0;          --constraint on generic parameter SIZE.
package STACK is

      OVERFLOW,UNDERFLOW:exception;

      --: function LENGTH return INTEGER range 0..SIZE;
      --: function "=" (S, T : STACK'TYPE) return BOOLEAN;
      -- this specification indicates that " = " will be
      -- redefined in the package body.

      procedure PUSH(E:in ELEM);
           --| raise OVERFLOW =>
           --|       STACK.LENGTH() = SIZE and
           --|       [STACK;PUSH(E)] = STACK;
```

```
        procedure POP (E : out ELEM);
            --| raise UNDERFLOW =>
            --|       STACK.LENGTH() = 0 and
            --|       [STACK;POP(E)] = STACK;



        --| axiom for all St:STACK'TYPE; X,Y:ELEM =>
        --|   [St;PUSH(X);POP(Y)] = St        and
        --|   [St;PUSH(X)].POP(E => Y).E = X and
        --|   St'INITIAL.LENGTH()        = 0   and
        --|   [St;PUSH(X)].LENGTH()      = St.LENGTH() + 1 and
        --|   [St;POP(X)].LENGTH()       = St.LENGTH() - 1;

    end STACK;

    package body STACK is

        type ELEM_ARRAY is array(1..SIZE) of ELEM;
        SPACE : ELEM_ARRAY;
        INDEX : INTEGER range 0..SIZE := 0;

-- this declaration of the state type is not necessary but is included for
-- completeness; it is the default assumed in Anna, see section 7.7.3.
        --: type STACK'TYPE is record
        --:                       SPACE : ELEM_ARRAY;
        --:                       INDEX : INTEGER range 1 .. SIZE;
        --:                     end record;
        --| where for all S : STACK'TYPE => 0 <= S.index <= size;
        --  this constraint on STACK'TYPE is equivalent to a monitor invariant
        --  in Hoare's sense [Hoare75].

        --: function LENGTH  return INTEGER range 0..SIZE;
            --| return STACK.INDEX;

        -- redefinition of function " = " on STACK'TYPE.
        --: function "=" (S, T : STACK'TYPE) return BOOLEAN;
            --| return  S.INDEX = T.INDEX  and
            --|    (for all K:INTEGER range 1 .. S.INDEX =>
            --|          S.SPACE(K) = T.SPACE(K));

        procedure PUSH(E:in ELEM)
            --| where out INDEX = in INDEX + 1 and
            --|       out SPACE = ELEM_ARRAY'UPDATE
            --|                     (in SPACE, out INDEX => E);

        is
        begin
            if INDEX = SIZE then
                raise OVERFLOW;
            else
                INDEX := INDEX +1;
                SPACE(INDEX) := E;
            end if;
```

```
        end PUSH;

        procedure POP(E:out ELEM)
             --| where out INDEX = in INDEX - 1 and
             --|  out SPACE = in SPACE and
             --|  out E = out SPACE (in INDEX);
        is
        begin
            if INDEX = 0 then
                raise UNDERFLOW;
            else
                  E := SPACE(INDEX);
                  INDEX := INDEX - 1;
            end if;
        end POP;

    end STACK;
```

2. Example of a symbol table package

```
generic
   type TOKEN is private;
   N: INTEGER;
package SYMTAB is

   OVERFLOW, UNDEFINED :exception;

   --: function SIZE  return INTEGER range 0 .. N;
   --: function "=" (SS, TT : SYMTAB'TYPE) return BOOLEAN;

   function DEFINED (S: STRING) return BOOLEAN;

   procedure INSERT(S: STRING; I: TOKEN);
   --| raise OVERFLOW => SYMTAB.SIZE = N;

   function LOOKUP(S: STRING) return TOKEN;
   --| raise UNDEFINED => SYMTAB = SYMTAB'INITIAL;

   procedure ENTERBLOCK;

   procedure LEAVEBLOCK;

--| axiom
--| for all SS : SYMTAB'TYPE; S,T : STRING; I : TOKEN =>
--| [SYMTAB'INITIAL; LEAVEBLOCK] = SYMTAB'INITIAL and
--| SYMTAB'INITIAL.DEFINED(S) = FALSE and
--| [SS;ENTERBLOCK;LEAVEBLOCK]=SS and
--| [SS;ENTERBLOCK].DEFINED(S) = FALSE and
--| [SS; ENTERBLOCK].LOOKUP(S) = SS.LOOKUP(S) and
--| [SS;INSERT(S,I);LEAVEBLOCK]=[SS;LEAVEBLOCK] and
--| [SS;INSERT(S,I)].DEFINED(T) = if S = T then TRUE
```

```
--|           else SS.DEFINED(T) and
--| [SS;INSERT(S,I)].LOOKUP(T) = if S = T then I else SS.LOOKUP(T);

end SYMTAB;



package body SYMTAB is

   type ELEM is record
                        LEVEL : INTEGER;
                        MEMBER : STRING;
                        TOK : TOKEN;
                     end record;

   type STORE is array (1 .. N) of ELEM;
   subtype INDEX_RANGE is INTEGER range 0 .. N;

   TABLE : STORE;
   LEXLEVEL : INTEGER := 0;
   INDEX : INDEX_RANGE := 0;

   -- this declaration of the state type is not necessary but is included for
   -- completeness; it is the default assumed in Anna, see section 7.7.3.
   --: type SYMTAB'TYPE is record
   --:                                TABLE : STORE;
   --:                             LEXLEVEL : INTEGER;
   --:                                INDEX : INDEX_RANGE;
   --:                          b[end  record];


   --: function SIZE  return INTEGER range 0 .. N;
       --| return SYMTAB.POS;

   --: function "=" (SS, TT : SYMTAB'TYPE) return BOOLEAN;
       --| return  SS.POS = TT.POS and
       --| for all I : INTEGER range 1 ..SS.POS => SS.TB(I) = TT.TB(I);

   function DEFINED (S: STRING) return BOOLEAN is
       --| return exist I : INDEX_RANGE range 1 .. INDEX =>
       --|      TABLE (I).MEMBER = S and  TABLE (I).LEVEL = LEXLEVEL;
        I : INDEX_RANGE;
        J : ELEM;
   begin
      if INDEX = 0 or else LEXLEVEL > TABLE (INDEX).LEVEL then
          return FALSE;
      else
          I := INDEX;
            loop
             J := TABLE (I);
             exit when J.LEVEL < LEXLEVEL;
             if J.MEMBER = S then return TRUE; end if;
             I := I-1;
            end loop;
          return FALSE;
```

```
        end if;
    end DEFINED;

    procedure INSERT(S: STRING; I: TOKEN) is
        --| raise OVERFLOW => SYMTAB.SIZE = N;
    begin
        if INDEX = N then
            raise OVERFLOW;
        else
            INDEX := INDEX + 1;
            TABLE (INDEX) := ELEM'(LEXLEVEL, S, I);
        end if;
    end INSERT;

    function LOOKUP(S: STRING) return TOKEN is
        --| raise UNDEFINED => SYMTAB = SYMTAB'INITIAL;
        --| return  J :  TOKEN that exist I : INDEX_RANGE
        --|                range 1 .. INDEX =>
        --|                  TABLE (I) .MEMBER = S and  TABLE (I).TOK = J;
        I : INDEX_RANGE;
    begin
        I := INDEX;
        if I = 0 then raise UNDEFINED; end if;
        loop
            exit when I = 0;
            if TABLE (I).MEMBER = S then
                    return TABLE (I).TOK;
            else
                I := I-1;
            end if;
        end loop;
        raise UNDEFINED;
    end LOOKUP;

    procedure ENTERBLOCK is
        --| out LEXLEVEL = in LEXLEVEL + 1;
    begin
        LEXLEVEL := LEXLEVEL + 1;
    end ENTERBLOCK;

    procedure LEAVEBLOCK is
        --| out (LEXLEVEL = in LEXLEVEL - 1 and
        --|      INDEX = that I ( in TABLE (I).LEVEL = in
        --|                LEXLEVEL - 1 and
        --|                    in TABLE (I+1).LEVEL = in LEXLEVEL ));
    begin
        loop
           exit when TABLE (INDEX).LEVEL < LEXLEVEL;
           INDEX := INDEX - 1;
        end loop;
        LEXLEVEL := LEXLEVEL - 1;
    end LEAVEBLOCK;

end SYMTAB;
```

8. VISIBILITY RULES IN ANNOTATIONS

8.1 No addition.

8.2 SCOPE OF A DECLARATIVE ANNOTATION

- The scope of a declarative annotation is the smallest enclosing scope of the declaration of any name in the declarative annotation, at most the scope of a declaration at the beginning of the respective declarative part.

- The scope of a declaration in the domain of a quantified expression extends from the declaration to the end of the expression.

- The scope of a virtual declaration, i.e., a declaration made within a virtual text sign, --:, is the same as if the virtual text sign is removed and the declaration is considered to be in the Ada text at the same point.

8.3 VISIBILITY

- The identifier of a formal out or in out parameter of a procedure is also visible as a selected component in expressions in annotations containing a 'OUT function attribute.

- An identifier declared in the domain of a quantified expression is directly visible in the expression subsequent to =>.

- A virtual declaration, i.e., a declaration made within a virtual text sign, --:, is visible only in other formal comments (virtual text and annotations) within its scope.

9. TASK ANNOTATIONS

Omission: A theory of annotations of tasks and multitask systems has yet to be developed. Extensions of Anna to tasks are planned. This will almost certainly involve introducing new predefined attributes associated with tasks and extending the language of expressions in annotations with modal operators.

10. PROGRAM STRUCTURE

```
context ::=
      [context_annotation]
      {with_clause [use_clause] [context_annotation]}

context_annotation ::=
    limited [to {name_list; [declarative_annotation;]}]
```

A context annotation restricts the visibility of variables further than the Ada context for the purposes of verification. If it is omitted, the usual Ada rules of visibility apply. Otherwise all variables required in the annotated unit must appear in a context annotation; it has the same meaning as if the global variables were (additional) formal parameters.

For any access type T, T'Collection denotes the associated collection variable and is thus required for any change of the collection, that is allocation for the access type. Functions are pure if their required_list is empty, and side-effect free, if constancy is specified for all visible variables.

Examples of context annotations:

```
--| limited
    package body STACK is
        type ELEM_ARRAY is ARRAY(1..SIZE) of ELEM;
        SPACE:ELEM_ARRAY;
        INDEX:INTEGER_range 0..SIZE := 0;

        procedure PUSH(E:in ELEM) is separate;
        --|  where out INDEX = in INDEX + 1 and
        --| out SPACE = ELEM_ARRAY'UPDATE (in SPACE, out INDEX, E);
                        ...
    end STACK;


--| limited to SPACE,INDEX;
    separate (STACK)
    procedure PUSH(E;in ELEM) is
                        ...
    end PUSH;
```

11. EXCEPTION ANNOTATIONS

Exception annotations are either annotations of an exception handler, or else propagation annotations in a subprogram declaration. Their purpose is to specify conditions under which a handler expects an exception, and conditions under which a subprogram expects to propagate an exception. If the annotations are sufficiently complete, they will enable proofs of consistency between programs and annotations to be obtained in the presence of exceptional behaviour (see [Luckham and Polak 80a]).

11.2 ANNOTATION OF EXCEPTION HANDLERS

Exception handlers may be annotated exactly as any other sequence of statements. Most important, an annotation at the beginning of the sequence of statements of an exception handler must be true before the handler is executed. (Such an annotation therefore constrains the situation in which the exceptions that are to be handled by that handler may be raised.

Note: An annotation at the beginning of a sequence of statements (of a compound statement, subprogram body, etc.) is an annotation of a **null** statement and constrains the entry state.

Example of a handler annotation

```
begin
--sequence of statements
exception
    when SINGULAR | NUMERIC_ERROR =>
        --| DET(A) = 0;      -- entry condition for handler
        PUT ("A IS SINGULAR");
    when others =>
        PUT("FATAL ERROR");
        raise ERROR;
end;
```

11.3 PROPAGATION ANNOTATIONS

```
propagation_annotation ::=
      raise exception_choice {| exception_choice}
                [=> boolean_expression]
```

A propagation annotation for a subprogram specifies the exceptions that may be raised as the result of a call to that subprogram, and the condition under which they are raised. A subprogram call is equivalent to a **raise** statement for one of the exceptions whenever an exception is propagated as the result of the call. The boolean expression subsequent to => must then hold.

Examples of propagation annotation:

```
procedure PUSH (E: in ELEM);
     --| raise OVERFLOW => STACK'.LENGTH() = SIZE;
```

```
procedure BINARY_SEARCH (A : in ARRAY_OF_INTEGER;
                                    KEY : in INTEGER;
                                    POSITION : out INTEGER )
   --| A( out POSITION) = KEY;
   --| raise NOT_FOUND =>
   --|     ORDERED (A) -> for all I in range A'RANGE =>
   --|                                    not KEY = A(I);
```

12. ANNOTATION OF GENERIC UNITS

A generic declaration includes a generic part and declares a generic subprogram or a generic package. The generic part may include the definition of generic parameters.

Annotations appearing in generic units may contain the generic parameters of the unit. Such annotations are templates that may be instantiated according to the Ada rules for matching actual and formal generic parameters. Instances of a generic unit are then annotated by the same instances of the annotation of the generic parent unit.

12.1.1 ANNOTATION OF GENERIC PARAMETERS

Generic formal parameters may be annotated analogously to formal parameter annotations (Sec. "6.2").

PREDEFINED ANNA ATTRIBUTES

Attribute of any access type:

COLLECTION	T'COLLECTION is a package denoting the collection of all allocated objects.
NULL	T'NULL corresponds to S'(null).
ALLOCATE	T'ALLOCATE(X : in out T) is a procedure attribute corresponding to new S; the out value of X is the newly allocated value. Values for constraints are added if necessary.
ELEMENT	T'ELEMENT (X : T) is a Boolean valued function attribute, returning TRUE if X has been allocated a value in T'COLLECTION.

Attribute of any procedure:

OUT	P'OUT is a function returning a record containing the final values of the out parameters of P after a call. The formal parameter names of P are the component names of this record.

Attributes of any package:

AXIOM	X'AXIOM binds an annotation to the visible part of package X. This axiom is then guaranteed to be true of all elements in the visible part.

REFERENCES

[Bauer et al. 78] Bauer, F.L., Broy, M., Gnatz, R., Hesse, W. and Krieg- Bruckner, B., A Wide Spectrum Language for Program Development. In: Robinet, B. (ed.), Program Transformations: 3rd Int. Symp. on Programming, Paris (1978), 1-15.

[Broy and Krieg-Bruckner 80] Broy, M. and Krieg-Bruckner, B., Derivation of Invariant Assertions During Program Development by Transformation, ACM TOPLAS (to appear 1980).

[FORM 81] Formal Definition of the Ada Programming Language, IRIA, V. Donzeau-Gouge, G. Kahn, B. Lang. (Nov. 1980).

[Hoare 69] Hoare, C.A.R., An Axiomatic Basis for Computer Programming, CACM 12, (Oct. 1969), 576-580.

[Hoare 75] Hoare, C.A.R., Monitors: An Operating System Structuring Concept. CACM 18. No.2, (1975).

[Hoare and Wirth] Hoare, C.A.R, and Wirth, N., An Axiomatic Definition of the Programming Language Pascal, Acta Informatica 2 (1973), 335-355.

[Ichbiah et al. 79a] Ichbiah, J.D., Krieg-Bruckner, B., Wichmann, B.A., Ledgard, H.F., Heliard, J-C., Abrial, J-R., Barnes, G.P., and Roubine, O., Preliminary Reference Manual for the Ada Programming Language, ACM SIGPLAN Notices (June 1979), Part A.

[Ichbiah et al. 79b] Ichbiah, J.D., Krieg-Bruckner, B., Wichmann, B.A., Ledgard, H.F., Heliard, J-C., Arbrial, J-R., Barnes, G.P., and Roubine, O., Preliminary Rationale for the Design of the Ada Programming Language, ACM SIGPLAN Notices (June 1979), Part B.

[Luckham and Polak 80a] Luckham, D., and Polak, W., Ada Exception Handling - An Axiomatic Approach, ACM TOPLAS, (May, 1980).

[Luckham and Polak 80b] Luckham, D., and Polak, W., A Practical Method of Documenting and Verifying Ada Programs with Packages, Proc. Ada Conference, (Dec. 1980).

[Luckham and Suzuki 79] Luckham, D.C., and Suzuki, N., Verification of Array, Record and Pointer Operations in Pascal, ACM TOPLAS 1, 2 (October 1979), 226-244.

[SVG 79] Luckham, D.C., German, S.M., v Henke, F.W., Karp, R.A., Milne, P.W., Oppen, D.C., Polak, W., Scherlis, W.L., Stanford Pascal Verifier User Manual, Stanford University, Stanford Verification Group, Report No. 11, 1979.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.