TOOLS FOR STORAGE AND RETRIEVAL OF ADA SOFTWARE COMPONENTS IN A SOFTWARE BASE

by

Christopher S. Eagle

March 1995

Thesis Co-Advisors: Man-Tak Shing Luqi

Approved for public release; distribution is unlimited.
# Tools for Storage and Retrieval of Ada Software Components in a Software Base

**Abstract**

One problem facing the Computer Aided Prototyping System (CAPS) project at the Naval Postgraduate School, is the lack of a large repository of existing reliable software components to draw upon for the creation of new prototype designs. Specifically, it is the lack of Prototype System Description Language (PSDL) specifications which describe Ada software components, that prevents Ada software components from being incorporated into the CAPS software base. Previously, PSDL specification had to be generated manually for each Ada software component being added into the software base. This process was time consuming and error prone.

The primary goal of this thesis is to solve this problem by creating a tool which accepts an Ada Package Specification as input and automatically generates its corresponding Prototype System Description Language (PSDL) specification. The Ada package along with its PSDL specification may then be stored directly into the CAPS software base.

The result of this thesis is a translator that examines each declaration contained in an Ada Package Specification and creates a corresponding PSDL specification. This tool allows the CAPS software base to be populated much faster utilizing existing DOD Ada software libraries such as the CAMP, ASSET, RAPID, and CRSS libraries. This tool has demonstrated its effectiveness by translating several complex components of the Common Ada Missile Packages into PSDL specifications.

**Subject Terms**

- CAPS, PSDL, Software Reuse, Syntactic Matching, Component Retrieval, Ada Translation

---

**Report Details**

- **Title:** Tools for Storage and Retrieval of Ada Software Components in a Software Base
- **Author:** Eagle, Christopher S.
- **Performing Organization:** Naval Postgraduate School, Monterey, CA 93943-5000
- **Funding Numbers:**
- **Abstract:** (Maximum 200 words)
- **Distribution/Availability Statement:** Approved for public release; distribution is unlimited.
- **Subject Terms:**
  - CAPS, PSDL, Software Reuse, Syntactic Matching, Component Retrieval, Ada Translation

---

**Security Classification:**

- Unclassified
- Unclassified
- Unclassified

**Price Code:**

- UL

**Number of Pages:** 183

---

**Form Approved OMB No. 0704-0188**

**NSN 7540-01-280-5500**

**Standard Form 298 (Rev. 2-89)**

Prescribed by ANSI Std. 239-18
TOOLS FOR STORAGE AND RETRIEVAL OF ADA SOFTWARE COMPONENTS IN A SOFTWARE BASE

Christopher S. Eagle
Lieutenant, United States Navy
B.S.E., University Of Michigan, 1985

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1995

Author:
Christopher S. Eagle

Approved By:
Man-Tak Shing, Co-Advisor

Luqi, Co-Advisor

Ted Lewis, Chairman,
Department of Computer Science
ABSTRACT

One problem facing the Computer Aided Prototyping System (CAPS) project at the Naval Postgraduate School, is the lack of a large repository of existing reliable software components to draw upon for the creation of new prototype designs. Specifically, it is the lack of Prototype System Description Language (PSDL) specifications which describe Ada software components, that prevents Ada software components from being incorporated into the CAPS software base. Previously, PSDL specification had to be generated manually for each Ada software component being added into the software base. This process was time consuming and error prone.

The primary goal of this thesis is to solve this problem by creating a tool which accepts an Ada Package Specification as input and automatically generates its corresponding Prototype System Description Language (PSDL) specification. The Ada package along with its PSDL specification may then be stored directly into the CAPS software base.

The result of this thesis is a translator that examines each declaration contained in an Ada Package Specification and creates a corresponding PSDL specification. This tool allows the CAPS software base to be populated much faster utilizing existing DOD Ada software libraries such as the CAMP, ASSET, RAPID, and CRSS libraries. This tool has demonstrated its effectiveness by translating several complex components of the Common Ada Missile Packages into PSDL specifications.
# TABLE OF CONTENTS

I. INTRODUCTION ........................................................................................................ 1  
   A. RAPID PROTOTYPING .................................................................................. 1  
   B. THE CAPS SYSTEM .................................................................................... 1  
   C. PROBLEM STATEMENT ........................................................................... 4  
   D. SCOPE ........................................................................................................ 4  
   E. ORGANIZATION OF THESIS ................................................................... 4  

II. SOFTWARE REUSE ............................................................................................... 7  
   A. WHY REUSE? ............................................................................................. 7  
   B. COMPONENT RETRIEVAL AND REUSE ....................................................... 7  
      1. Software Component Retrieval Methods ........................................... 7  
      2. Component Retrieval Within CAPS ................................................... 9  
   C. PROTOTYPE SYSTEM DESCRIPTION LANGUAGE ................................ 10  

III. MAPPING ADA 95 TO PSDL ............................................................................ 13  
   A. INTRODUCTION .......................................................................................... 13  
   B. THE MAPPING SUBSET OF ADA ............................................................... 13  
      1. Basic Rules for Translations .............................................................. 14  
      2. Productions Which do not Translate to PSDL ................................. 17  
      3. Productions Which Translate to PSDL ............................................. 17  
      4. Generic Formal Parameters ............................................................. 18  

IV. THE ADA TO PSDL TRANSLATOR .................................................................. 21  
   A. BACKGROUND ........................................................................................... 21  
   B. THE SYNTHESIZER GENERATOR ............................................................. 21  
   C. THE TRANSLATOR .................................................................................... 22  
      1. SSL Source Files .................................................................................. 22  
      2. Accomplishing the Translation ......................................................... 23  
      3. Assumptions for Proper Translations ............................................... 23  

V. PSDL TYPE COMPONENT RETRIEVAL .............................................................. 29  

vii
I. INTRODUCTION

A. RAPID PROTOTYPING

The classical approach to software development, the waterfall method, is a method in which a project moves forward one phase at a time [Ref. 1]. The phases consist of analysis, design, implementation, and testing. Each decision made in the analysis phase propagates through to the testing phase, and any problems encountered in the testing phase will require a return to the analysis phase to correct.

Rapid prototyping provides an alternative method to the waterfall method. In rapid prototyping, a spiral rather than linear approach is followed, which allows various phases of development to proceed in parallel [Ref. 2]. A prototype is constructed quickly which is used to verify both the users requirements, and the designers interpretation of those requirements. A model for this method is shown in Figure 1 [Ref. 3].

One hinderance to the idea of rapid prototyping is the time required to complete the coding of a system. The concept of software reuse is one that can dramatically reduce the time spent on coding. Utilizing existing software components, prototype designers can rapidly construct systems with significant functionality rather than mere skeletons with large numbers of procedural stubs. These software components are stored and retrieved from a library of software components which is an integral part of the prototype designers tools.

B. THE CAPS SYSTEM

Computer aided prototyping of hard real-time systems is the goal of the Computer Aided Prototyping System (CAPS) project at the Naval Postgraduate School. CAPS provides tools which enable users to graphically specify a system, retrieve existing software components from a software base and integrate them to form the specified
prototype, perform timing analysis, and create a running prototype of a hard real-time system. Together these tools form the CAPS development environment shown in Figure 2.

Timing analysis in the prototype is performed by the schedulers which examine the prototype specifications to produce a static schedule and a dynamic schedule. A static schedule is generated to fulfill the timing requirements of all time critical components in the system, and ensure that each of these components is scheduled as frequently and as long as they require. A dynamic schedule is created to allow those components which are not time critical to be incorporated into the processor schedule on a not to interfere basis with the time critical operators. In order to accomplish the required analysis, the timing requirements of each component in the prototype system must be properly specified. The Prototype System Description Language (PSDL) is a language which has been designed to accomplish exactly this [Ref. 4]. PSDL is a formal description language which provides a means to specify a variety of timing information for software components. Every software component which may be incorporated into a CAPS prototype must have a corresponding
PSDL description to define its characteristics. While there exists a large body of Ada software components which could potentially be utilized in the creation of systems, few if any of these components have had corresponding PSDL descriptions created for them. Before proper PSDL descriptions can be generated for existing Ada components, a thorough understanding of the mapping from Ada package specifications into PSDL specifications is required. As PSDL is not a language with which most Ada programmers are familiar, a major goal of this thesis has been to create a tool which will examine Ada package specifications and automatically generate corresponding PSDL descriptions for those specifications. The CAPS software base provides a database for these Ada components and their PSDL descriptions. The PSDL descriptions serve as the target keys
in searches of the software database. Because prototype systems are specified using PSDL, it becomes necessary to be able to compare one PSDL specification against another. All PSDL components are one of two types, a PSDL type component, or a PSDL operator component. Scott Dolgoff has described and implemented methods for search, retrieval, and integration of PSDL operator components from the software base into CAPS prototypes [Ref. 5]. A secondary objective of this thesis is to develop methods for searching and retrieving PSDL type components from the software base. The implementation of these methods is left for future research.

C. PROBLEM STATEMENT

CAPS has been the focus of research efforts for several years. Many different parts of CAPS have been implemented as the end product of Doctoral and Masters’ theses over the years, during different stages of the design effort, using system models with higher level of maturity as time passes.

The focus of this thesis is to create a tool which allows software base administrators to more rapidly incorporate existing Ada software components into the software base. This is accomplished by automatically generating a PSDL specification, which accurately describes the Ada component, which will be stored along with the Ada component into the software base for later retrieval.

D. SCOPE

The scope of this thesis includes the development of the previously described tool which can operate both interactively and in a batch mode. Additionally, methods for the retrieval of PSDL type components from the software base are developed as a foundation for further research in the area of software retrieval.

E. ORGANIZATION OF THESIS

Chapter II discusses software reuse, and the role which PSDL plays in software reuse within CAPS. Chapter III describes the mapping of Ada package specifications into PSDL component specifications. Chapter IV discusses the design of a tool which
automatically translates Ada package specifications into corresponding PSDL component specifications. Chapter V describes methods for database search and retrieval of PSDL type components. Chapter VI presents conclusions.
II. SOFTWARE REUSE

A. WHY REUSE?

In 1984 it was determined that "of all the code written in 1983, probably less than 15% is unique, novel, and specific to individual applications. The remaining 85% appears to be common, generic and concerned with putting applications onto computers" [Ref. 6]. It is evident that by building large libraries of software components designed for reuse, time can be saved in the construction of future software systems. Additionally, reliability in final products is enhanced by the use of time tested components. Problems being faced today include the availability of large libraries of existing components, and the methods to retrieve and integrate these components into new software systems. The focus of this thesis is to provide a tool to automate the process of populating a library of software components, in this case the CAPS Software Base.

B. COMPONENT RETRIEVAL AND REUSE

In its simplest form, software reuse exists as a simple copy and paste operation in a programmers development environment. Programmers, familiar with their own previously written code, may reuse portions of that code when creating new systems. Two immediate benefits await that programmer, first, time is saved that would otherwise have been spent creating new code from scratch, and two, this recycled code has been debugged, and shown to be reliable within its original system. The goals of software reuse are to make this reuse effort pay off on a larger scale, rather than programmers reusing only their own code, it is desirable to have large libraries of tested and debugged components available to all members of an organization.

1. Software Component Retrieval Methods

The collection of software components into some form of component library itself is no problem, the problem lies in methods for retrieving components from such a library, and integrating those components into a new software system. Three primary methods for
retrieving software components exist: browsing, informal specifications, and formal specifications [Ref. 7]. Each of these methods is described below:

a. Browsers

Browsers simply provide their users a means to scan through a software library in search of something useful. A successful search conducted with a browser will rely heavily on the users ability to recognize a desirable component when it is displayed by the browser. Components within libraries served by browsers must utilize recognizable naming conventions or be thoroughly commented in order for users to identify and evaluate a component as having potential for reuse. A significant amount of time can be spent scanning through large software libraries, which may negate the time savings gained by utilizing any retrieved components.

b. Informal Specifications

Informal specifications are queries constructed by a user with the goal of searching a software library for matching components. This type of search may utilize keywords to describe component behavior, or classify components by functionality. Attributes such as type and numbers of parameters may also be specified in a query of this sort. Successful queries of this sort require users to utilize appropriate keywords in order to locate desirable components. For example a user who queries based on the keyword “list” may not be informed of components in the library named “queue.” The burden of evaluating components located in this manner remains entirely with the user. Users must still perform final evaluation of components retrieved in this manner in order to determine their usefulness.

c. Formal Specifications

Formal specification searches attempt to rigidly define a users requirements. This type of search can be the most automated of all searches, and therefore produces the most accurate and efficient results of the three methods mentioned. Searches may be
conducted to compare a library components syntactic similarity and semantic similarity against a formal specification provided by a user. In syntactic matching, numbers and types of parameters, are compared against components within the library to yield only those components which have signatures which match a users query. Semantic matching attempts to go one step further by examining behaviors identified by the user as desirable, against known behavior of library components. Components fully satisfying the constraints imposed by semantic matching are exactly those which can be integrated directly into the users new systems. [Ref. 8]

2. Component Retrieval Within CAPS

There are currently three methods available for searching the CAPS software base for desirable components. These methods are browsing, keyword search, and PSDL query. [Ref. 5]

a. Browsing Within CAPS

Browsing simply presents the user with a list of available type or operator components which may be further examined, and ultimately selected for inclusion in the users system, should they meet the users requirements.

b. Keyword Search in CAPS

PSDL permits the use of keywords within PSDL components as shown in Figure 3. These keywords are used as the basis of keyword searches within CAPS. The user is presented with a list of all keywords currently used by CAPS software base components. From this list, the user can select one or more keywords, and the search will yield all components containing those keywords. The user may then browse the resulting list to find a specific component which satisfies the users requirements.

c. PSDL Query

A PSDL query is a query by formal specification. In order to perform a PSDL query, the user must provide a PSDL specification as a query. That query is
compared against PSDL components stored in the CAPS software base, and only those components within the software base that satisfy the query are returned. Syntactic matching examines both numbers and types of parameters within the query component in search of a match within the software base. Syntactic matching has been described and implemented by John McDowell [Ref. 9] and Scott Dolgoff [Ref. 5].

```
OPERATOR Addition
SPECIFICATION
  INPUT
    Op1 : Integer,
    Op2 : Integer
  OUTPUT
    Result : Integer
KEYWORDS
  addition, sum
END
```

**Figure 3. PSDL Component with Keywords**

Semantic matching examines not only the external interface to a component, but the internal behavior of the component as well [Ref. 7]. Internal behavior is evaluated by utilizing normalized algebraic expressions which describe the desired behavior in a query specification, and actual behavior of a software base component. These expressions are embedded in PSDL specifications in the form of OBJ3 conditional equations utilizing the *axioms* facility of PSDL.

C. **PROTOTYPE SYSTEM DESCRIPTION LANGUAGE**

PSDL is a text-based language designed to express the specifications of real-time systems. It is based on a graphic model of vertices and edges, in which the vertices represent *operators*, or software process, and the edges represent the conceptual "flow" of data from one operator to another. Each vertex and edge may have associated timing constraint, and the vertices may have associated control constraints.

Formally, the model used is that of an augmented graph,
\[ G = (V, E, T(v), C(v)) \]

where \( G \) is the graph, \( V \) is the set of vertices, \( E \) is the set of edges, \( T(v) \) represents the timing constraints for the vertices, and \( C(v) \) represents the control constraints for the vertices [Ref. 4].

Conceptually, PSDL operators may contain other operators to support the principle of abstraction. Effectively, the prototype may be expressed as a flat graph, or a one level graph containing all the atomic operators and their streams. An atomic operator is one that is implemented in a programming language, vice a composite operator consisting of other operators and streams.

For example, the following diagram shows a PSDL prototype:

This graph represents an operation modelled by the Operator X that accepts one item from Stream A, it performs some operation on the data, and outputs Stream B. The Maximum Execution Time (MET), this is the maximum possible time the operator may take to execute the task, defined as 600 milliseconds.

Figure 4. Example of PSDL Graph
In this example, Operator X is decomposed as follows:

![Decomposed PSDL Graph](image)

Operator X is a composite operator, while Operator X1 and Operator X2 are atomic operators, implemented in Ada or some other language. The timing and control constraints on these atomic operators must be consistent with those of their parent operator. In a single processor the combined METs of these atomic operators can not be greater than their parent. Operator X is really not needed to implement the prototype. It serves merely to abstract the functionality of its child operators. A more detailed description of the PSDL may be found in [Ref. 4] and [Ref. 3].
III. MAPPING ADA 95 TO PSDL

A. INTRODUCTION

In order to perform the translation of Ada software into PSDL specifications, a well formulated set of rules is required which will accept all legal Ada programs, and provide an accurate translation in all cases. Past research within the CAPS project has provided the capability to perform Software Base queries based upon syntactic attributes of desired software components. This syntactic matching process serves as a front end filter for later semantic matching operations. The syntactic characteristics of an Ada package can be observed entirely within the package specification by examining the number of procedure declarations, and the number and type of arguments to those procedures. Semantic matching operations must focus on the actual behavior exhibited by component software packages. Behavioral characteristics can not be inferred from a package specification alone, and it becomes necessary to examine the package bodies of software components in order to provide effective semantic matching services.

The work in this thesis provides additional capabilities for existing syntactic matching services in the CAPS environment. The focus of the translation effort is therefore limited to Ada package specifications. Future research will examine the possibilities of extending these methods to cover Ada package bodies.

B. THE MAPPING SUBSET OF ADA

The CAPS software base, and retrieval tools are set up in a way that requires Ada software components to be contained within an Ada package. For that reason, the starting point for the translation effort, which is the focus of this thesis, is the subset of the Ada 95 language which is required to specify package specifications. This subset is expressed by the abstract syntax of Appendix A. Given this grammar subset, there are productions which have no translation to PSDL, productions which have some form of translation to PSDL, and productions which translate in a nearly one to one manner to PSDL.
1. Basic Rules for Translations

The translator accepts Ada 95 package specifications and produces appropriate PSDL specifications which describe the given Ada component. The input Ada component must be a valid Ada 95 package specification free of syntax errors. The output of the translator will be a valid PSDL component. PSDL components are either data types or operators. A PSDL operator represents a single operation which may have inputs and outputs. A PSDL data type represents a state machine along with the associated operators to manipulate that state machine.

Ada procedures translate rather straightforwardly into PSDL operators as shown in Figure 6. Ada packages which contain a single procedure declaration will translate into PSDL operator components. The name of the PSDL operator component will be that of the

<table>
<thead>
<tr>
<th>Ada:</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure ExampleProc(p1 : type1; p2 : out type2; p3 : in out type3);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PSDL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator ExampleProc</td>
</tr>
<tr>
<td>specification</td>
</tr>
<tr>
<td>input</td>
</tr>
<tr>
<td>p1 : type1,</td>
</tr>
<tr>
<td>p3 : type3</td>
</tr>
<tr>
<td>output</td>
</tr>
<tr>
<td>p2 : type2,</td>
</tr>
<tr>
<td>p3 : type3</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>implementation ada ExampleProc</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

Figure 6. Translation of an Ada Procedure
single procedure contained in the Ada package. For example, the Ada package of Figure 7

```ada
package OneProc_pkg is
    DemoException : exception;
    procedure Proc1(x : integer; y : in out float);
end OneProc_pkg;
```

Figure 7. Ada Package Containing a Single Procedure

becomes the PSDL operator of Figure 8. Ada package specifications which contain either

```psdl
operator Proc1
    specification
        input
            x : integer,
            y : float
        output
            y : float
        exceptions
            DemoException
end
implementation ada Proc1
```

Figure 8. PSDL Translation of Single Procedure Package

no procedure declarations or two or more such declarations will translate into PSDL type components. In this case, the package must contain a type declaration for an abstract data type upon which all of the procedures within the package operate. The name of the PSDL type component will be that of the abstract data type so defined. An example of a well
formed PSDL type component is shown as an Ada package specification in Figure 9 and in

```ada
package Set_pkg is
    type Set is private;
    DemoException : exception;
    procedure Union(sl, s2 : in Set; Result : out Set);
    SecondException : exception;
    procedure Size(s : in Set; result : out integer);

    private
        type Set is array(1..10) of integer;
    end Set_pkg;
end Set_pkg;
```

**Figure 9. Ada Package with Two Procedures**

Figure 10 as its corresponding PSDL specification. Current CAPS prototype generation

```psdl
type Set
specification
operator Union
    specification
    input
        s1, s2 : Set
    output
        Result : Set
    exceptions
        DemoException, SecondException
end
operator Size
    specification
    input
        s : Set
    output
        Result : integer
    exceptions
        DemoException, SecondException
end
end
```

**Figure 10. PSDL Translation of Two Procedure Package**

tools do not allow for the use of functions within Ada software components. Operations
which must return a value should be written as procedures with an additional out parameter which is used to pass the return value to the calling unit. Within an Ada package specification, each procedure will translate into a PSDL operator specification.

2. Productions Which do not Translate to PSDL

The following list represents Ada 95 productions which have no legitimate translation to PSDL:

- Pragmas.
- Object declarations.
- Number declarations.
- Type declarations.
- Subtype declarations.
- Task declarations.
- Function declarations.
- With and Use clauses.
- Generic formal parameters which are packages.

3. Productions Which Translate to PSDL

The following Ada productions have some form of translation into PSDL:

- Package declarations (including generics).
- Procedure declarations (including generics).
- Exception declarations.
- Generic formal parameters (except packages).

Packages translate as described previously. Ada procedures are translated to PSDL by listing the in and in out parameters of the procedure as inputs of the translated PSDL operator, and the out and in out parameters as outputs of the translated PSDL operator. All exceptions declared within a package are listed as PSDL exceptions of all translated PSDL operators. This approach to exceptions is a conservative approximation that includes all possible behaviors, as it is not possible by examining an Ada package specification to determine which declared exceptions will be raised by particular procedures. Restricting
exception declarations to only those operators which actually raise them would require analysis of Ada components to be extended to the package bodies as well. This requires a considerable amount of additional computational effort with relatively little gain in translational accuracy.

4. Generic Formal Parameters

Naming conventions were required in order to properly translate generic formal parameters in an Ada specification into generic formal parameters of a PSDL component. The translations for generic formal type parameters are shown in Figure 11. In order to specify generic function and procedure parameters, it is necessary to utilize the array syntax of PSDL to specify the parameters, and the associated types of the Ada function and procedure parameters. The translation of an Ada generic formal function parameter is shown in Figure 12. PSDL array syntax is utilized to specify the formal parameters and the

<table>
<thead>
<tr>
<th>Ada</th>
<th>PSDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>type t1 is (&lt;&gt;);</td>
<td>t1 : DISCRETE_TYPE</td>
</tr>
<tr>
<td>type t2 is RANGE &lt;&gt;;</td>
<td>t2 : RANGE_TYPE</td>
</tr>
<tr>
<td>type t3 is MOD &lt;&gt;;</td>
<td>t3 : MOD_TYPE</td>
</tr>
<tr>
<td>type t4 is DELTA &lt;&gt;;</td>
<td>t4 : DELTA_TYPE</td>
</tr>
<tr>
<td>type t5 is DELTA &lt;&gt; DIGITS &lt;&gt;;</td>
<td>t5 : DELTA_DIGIT_TYPE</td>
</tr>
<tr>
<td>type t6 is DIGITS &lt;&gt;;</td>
<td>t6 : DIGITS_TYPE</td>
</tr>
<tr>
<td>type t7 is PRIVATE;</td>
<td>t7 : PRIVATE_TYPE</td>
</tr>
</tbody>
</table>

Figure 11. Translation of Generic Formal Parameters

<table>
<thead>
<tr>
<th>Ada:</th>
<th>PSDL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>with function func1(parm1 : integer; parm2 : float) return boolean is &lt;&gt;;</td>
<td>func1 : function [ parm1 : integer, parm2 : float, return : boolean ]</td>
</tr>
</tbody>
</table>

Figure 12. Translation of Generic Formal Function Parameter
return type of the parameter function. The return type is appended to the formal parameters of the function to make up the array components. The identifier *return* in the PSDL translation is guaranteed to be unique since it is a keyword in Ada, and no identifier in Ada may bear that name. It is not necessary to include information about the modes of the various parameters as they can be inferred from the fact that it is an Ada function being translated. All formal parameters are assumed to be *in* parameters, and the return parameter is assumed to be an *out* parameter. The translation of an Ada generic formal procedure parameter takes on a slightly different form. In the case of a procedure, it is necessary to encode the mode information for each formal parameter of the procedure. Again, the syntax for PSDL generic types is utilized. Nesting of PSDL generic instantiations is used to encode parameter mode information for each formal parameter. The identifiers *in*, *out*, and *in_out* are used to indicate the mode of a parameter. The identifier *t* is used as a placeholder, followed by the actual type of the formal parameter. This method maintains consistency of translation between Ada types and PSDL types, and allows for the proper translation of nested type definitions. The translation of an Ada generic formal procedure parameter is shown in Figure 13.

<table>
<thead>
<tr>
<th>Ada:</th>
</tr>
</thead>
</table>
| with procedure proc1(p1 : integer;  
| p2 : out float;  
| p3 : in out boolean) is <>;  
| PSDL: |
| proc1 : procedure [  
| parm1 : in [ t : integer ],  
| parm2 : out [ t : float ],  
| parm3 : in_out [ t : boolean ] ] |

*Figure 13. Translation of Generic Formal Procedure Parameter*
IV. THE ADA TO PSDL TRANSLATOR

A. BACKGROUND

Previous work in this area was completed by Jennie Sealander in 1992 [Ref. 10]. A variety of deficiencies in that work lead to requirements for follow-on work. These deficiencies include:

- No support for the Ada 95 language.
- Failure to handle nested packages.
- Restriction to uppercase only or lowercase only for Ada keywords.
- Does not handle exceptions for PSDL operators.
- Improper handling of generic value, generic array, and generic subprogram parameters.
- No support for in out parameters.

In order to provide an updated translator, the decision was made to build a completely new version rather than attempt to upgrade the existing version. Starting from scratch allowed the selection of a new tool for constructing the translator. While tools such as Kodiak, developed at the University of Minnesota, and Eli, developed at the University of Colorado, are available as compiler/translator generators, the Synthesizer Generator (SynGen) was ultimately selected for implementation of the translator. All of these tools are based upon the concept of attribute grammars as described by Knuth [Ref. 11]. SynGen was selected because it has the capability to generate a syntax-directed editor from the specified attribute grammar. Additionally, SynGen was used to construct the PSDL editor utilized by CAPS, so a common interface is achieved with the Ada 95 editor generated as a part of the translator tool.

B. THE SYNTHESIZER GENERATOR

The Synthesizer Generator is a tool, which through the use of attribute grammars can create a variety of syntax directed editors, translators, and other language based tools. The generated tools are designed to be run within the X graphical environment, but may be
created to operate in batch modes as well. The Synthesizer Specification Language (SSL) is utilized to create editor specifications. SynGen creates C language source files utilizing a user’s SSL files, and other tools such as YACC. These source files are compiled to create a stand-alone final product. SSL constructs are used to specify several aspects of user specified editors including:

- The abstract syntax of a language.
- Context-sensitive relationships.
- Display format.
- Concrete input syntax.
- Transformation rules for restructuring objects. [Ref. 12]

Each of these may be specified in separate files, with an abstract syntax the only requirement for executing SynGen. This allows tools to be constructed in an incremental manner, greatly easing the debugging process.

C. THE TRANSLATOR

The translator constructed utilizing SynGen can be operated in two modes, interactive and batch. In the interactive mode, the translator is a syntax directed Ada 95 package specification editor which simultaneously produces PSDL translations. In the batch mode, an existing Ada package specification is specified as input to the translator which produces two output files, a PSDL translation, and an annotated Ada file which has comments interspersed with the Ada source code, these comments indicate the quality of the translation which has taken place. By examining the annotated Ada source file, users can get an idea as to how complete the translation was. Error messages inserted into the Ada source code, in the form of Ada comment statements, indicate which lines of source presented problems for the translator.

1. SSL Source Files

The translator is constructed from eight SSL source files. Two files specify the abstract syntax of for Ada 95 package specifications, and PSDL. The source code listings
for the abstract syntax files are contained in Appendix A. One file specifies the concrete syntax for Ada 95 package specifications which allows the translator to accept existing Ada 95 text files as input. Appendix D contains the concrete syntax rules for the translator. One file contains SSL functions which are used to compute attributes for Ada 95 productions which translate to PSDL. These functions are contained in Appendix C. Three files are required to specify the unparsing rules for Ada 95 and PSDL. Unparsing rules specify the format which is to be used for display of the underlying syntax trees. The source code for all unparsing rules may be found in Appendix B. The final file contains SSL transformation declarations which specify the manner in which users of the interactive translator may manipulate the syntax tree for Ada specifications. Appendix E contains the source listing for the transformation declarations used by the translator.

2. Accomplishing the Translation

An Ada package specification is translated into PSDL by taking advantage of the way in which SSL treats abstract syntax definitions and user-defined attribute types. In SSL these two concepts are merged so that attributes are in turn root nodes of an abstract syntax tree. In order to perform a translation, both the abstract syntax for Ada 95 and the abstract syntax for PSDL were specified. The Ada production pkg_spec contains a single attribute psdl_trans which serves as the root of a PSDL abstract syntax tree. The attribute, psdl_trans, is computed based upon the structure of the Ada abstract syntax tree rooted at pkg_spec by utilizing a variety of attribute computation functions which extract information from the Ada tree and convert it to nodes in the PSDL tree. By displaying the PSDL abstract syntax tree rooted at psdl_trans, a translation of the Ada pkg_spec is obtained.

3. Assumptions for Proper Translations

There are several restrictions which apply to the use of the translator. These fall into two categories:

- Implementation imposed limitations.
Limitations imposed by Ada to PSDL mapping restrictions.

The translator expects input files to contain only Ada package specifications. Ada package bodies are not recognized by the translator, and will result in no translation being accomplished. Components which place both the package specification and the package body in a single file must be split into two separate files, one containing the specification and the other the package body. The specification file is the file which the translator will process.

The input file may contain zero or more Ada package specifications, but it is recommended that each input file contain only a single package specification in order to produce only a single PSDL component as output. Multiple package specifications in a single file will result in multiple PSDL component specifications in the output file.

Ada functions are not translated into PSDL because the CAPS prototype construction tools provide only for interfacing to Ada procedures. In order to utilize the vast amount of existing Ada functions which have been written, packages which contain functions should be preprocessed to add procedure wrappers for each function interface. This is done by adding an additional procedure within the package specification which contains the same parameters as the function, and an additional out parameter used to pass out the return value. In the package body, a wrapper procedure is inserted which calls the function and passes out the functions return value in its extra out parameter. Appendix F discusses this process and contains examples of how this is done for both generic and non-generic functions within a package.

PSDL does not allow the nested definition of type components. In many cases the outermost package specification in a file may contain one or more nested package specifications. If these nested package specifications translate to PSDL type components, then the outermost package specification is stripped off and each nested package translated as a unique PSDL type component. This makes more of the software components available for reuse. If the outer package was not stripped away, the nested packages would not be
translated at all, and would be unavailable for reuse. Figure 14 show and Ada package

```
package Outer_Pkg is

    procedure OuterProc1( parm1 : in integer);
    procedure OuterProc2(parm2 : out float);

package Inner_Pkg is

    procedure InnerProc1(parm1 : character);
    procedure InnerProc2(parm2 : in out integer);

end Inner_Pkg;
end Outer_Pkg;
```

**Figure 14. Package with Nested Package**

specification containing a nested package. A strict translation of the this package to PSDL is shown in Figure 15. Notice that no translation of Inner_Pkg occurs. Inner_Pkg would

```
type Outer_Pkg
    specification
        operator OuterProc1
            specification
                input
                    parm1 : integer
                end

        operator OuterProc2
            specification
                output
                    parm2 : float
                end
    end
implementation ada Outer_Pkg
end
```

**Figure 15. Strict Translation for Nested Packages**
translate into a PSDL type component, however, no translation is performed because nested types are not allowed in PSDL. By allowing the outermost package in a specification to be stripped away, the translation of Figure 16 is obtained. This form of translation makes many

```ada
operator OuterProc1
  specification
    input
      parm1 : integer
end
implementation ada OuterProc1
end

operator OuterProc2
  specification
    output
      parm2 : float
end
implementation ada OuterProc2
end

type Inner_Pkg
  specification
    operator InnerProc1
      specification
        input
          parm1 : character
end

    operator InnerProc2
      specification
        input
          parm2 : integer
          output
            parm2 : integer
end
end
implementation ada Inner_Pkg
end
```

**Figure 16. Translation with Outer Package Stripped Away**

more packages available within the software base. It is an attempt to allow access to
software in cases where many unrelated packages are bundled together to form a single package simply for containership.
V. PSDL TYPE COMPONENT RETRIEVAL

A. BACKGROUND

Design and implementation of tools for retrieving PSDL operator components was performed by McDowell [Ref. 9] and Dolgoff [Ref. 5]. In particular Dolgoff's work yields a tool which utilizes user interactions to retrieve "best match" operator components from the CAPS software base for integration into prototype systems. It is desirable to extend this tool to allow the retrieval of PSDL type components as well. This chapter discusses considerations for the retrieval of PSDL type components from the software base, while leaving actual implementation of such a tool for future research.

B. PSDL TYPE COMPONENTS

PSDL type components are similar to the "objects" of object-oriented programming languages. PSDL types represent a data object and the associated operators to manipulate that object, within CAPS, they correspond to Ada abstract data types (ADT). Figure 17 shows a partial specification of a generic PSDL set data type. PSDL type components may be either generic, or non-generic, may contain internal type declarations, and may contain zero or more operators which operate on that type.

In order to retrieve a type component from the software base a user must formulate a PSDL query which specifies the user's type component. This will be referred to as the query type component, or simply query component (qc), throughout the remainder of the chapter. Given a query component, the software base is searched in order to find a match for the specified query component. Software base components (sbc) are those PSDL type components residing in the software base which are the objects of search process. Any
software base components which pass through all filtering operations become possible candidates for integration into the user’s prototype system.

```plaintext
type Set
specification
generic
   Element : PRIVATE_TYPE

operator Insert
specification
   input
      NewElement : Element
   output
      NewSet : Set
end

operator Empty
specification
   output
      EmptySet : Set
end

operator In
specification
   input
      Item : Element,
      S1 : Set
   output
      Result : boolean
end
end
```

Figure 17. PSDL Specification of a Set Data Type

C. MATCHING TYPE COMPONENTS

The goal of the matching process is to locate, for the user, as many type components as possible which may suit the requirements of the user’s prototype. In presenting these “matching” types to the user, it is desirable to narrow the range of choices the user must evaluate to those which have the highest likelihood of suiting the user’s needs. In order to
prevent the user from browsing through the entire dictionary of type components within the CAPS software base, several filters are applied, utilizing the user’s query component, to make the list of choices more manageable for the user. These filters are constructed based upon information specified by the users query component. An initial query to the software base utilizing these initial filters will return a set of type components which will be subjected to further processing. The results of this second pass over the components are then displayed to the user, who can browse the list of type components in search of the most suitable for the current prototype system. Once the user has selected a type component for integration into a CAPS prototype, the retrieved component must be made available to the user in a form which will integrate directly into the prototype. In the case of generic type components, it is necessary to first instantiate the component. Following instantiation, integration proceeds similarly for both generic and non-generic type components. A wrapper package must be constructed which suitably renames and instantiates the target component into a component which will integrate directly into the users prototype system.

D. DEFINITIONS

The following definitions are taken from Dolgoff’s thesis and are utilized here for consistency [Ref. 5].

1. PSDL Specification

The PSDL specification for a component denoted by PS.

2. Software Base Component

The software base component is denoted by sbc. The PSDL specification of a software base component is denoted by PS(sbc).

3. Query Component

A query component refers to the component that the CAPS user is in the process of finding a match for and is denoted by qc. The PSDL specification for a query component is denoted by PS(qc).
4. Component Signature

The component signature refers to the types of the component parameters, with separate signatures representing the input and output parameters of software base components. A signature encodes information that describes each instance of parameter types utilized by the component. Figure 18 shows the signature for a PSDL operator component. Ordering of types within a signature is insignificant. For example, the input signature (Boolean, Integer, Range, Integer) is considered a match for the input signature in Figure 18. The types will be mapped by the wrapper package created to integrate a software base component into the users prototype. For PSDL type components, the signatures represent the aggregate of all parameter types utilized by the types operators. Figure 19 shows the signatures for an example type component.

```
operator ExampleOp
  specification
    input
      Parm1 : Integer,
      Parm2 : Integer,
      Parm3 : Boolean,
      Parm4 : Range
    output
      Parm5 : Float
  end

Input Signature:
  (Integer, Integer, Boolean, Range)

Output Signature:
  (Float)
```

Figure 18. Example Operator Component Signatures

component. Ordering of types within a signature is insignificant. For example, the input signature (Boolean, Integer, Range, Integer) is considered a match for the input signature in Figure 18. The types will be mapped by the wrapper package created to integrate a software base component into the users prototype. For PSDL type components, the signatures represent the aggregate of all parameter types utilized by the types operators. Figure 19 shows the signatures for an example type component.

a. Parameter Types

In the simplest case of parameter matching, an input PS(qc) parameter exactly matches an input PS(sbc) parameter. However, the type hierarchy employed by Ada allows types to be matched in some cases where it would appear that no match exists.
The types Private, Discrete, Integer, Range, Natural, Positive, Enumeration, Character, Boolean, Access, Record, Array, String, Digits, Float, Delta, and Fixed are predefined and form the hierarchy depicted in Figure 20. Utilizing these relationships, it can be seen that

```
type ExampleType
  specification
  operator TypeOp1
    specification
    input
      Parm1 : Boolean,
      Parm2 : Integer
    output
      Parm3 : Integer
  end

  operator TypeOp2
    specification
    input
      Parm2 : Integer
  end
end

Input Signature:
  (Boolean, Integer, Integer)

Output Signature:
  (Integer)
```

Figure 19. Example Type Component Signatures

...and input PS(qc) parameter of type Positive can be matched to an input PS(sbc) parameter of type Integer. This is a one way relationship. Input parameter types in a PS(sbc) must accept the entire range of values expressed by the input parameter types of a PS(qc). Conversely, the output parameter types of a PS(qc) must accept all values generated by the output parameter types of a PS(sbc).
b. Input Parameters

Each input parameter has an identifier name, and a type. The identifier name is represented by \( p \). The expression \( \text{input_type}(p, \text{sbc}) \) refers to the parameter type for a specified input parameter \( p \) in a \( \text{PS}(\text{sbc}) \). Similarly, the expression \( \text{input_type}(p, \text{qc}) \) refers to the parameter type for a specified input parameter \( p \) in a \( \text{PS}(\text{qc}) \). The expression \( \text{In}(\text{sbc}) \) refers to the entire set of input parameter identifier names in a \( \text{PS}(\text{sbc}) \), and \( \text{In}(\text{qc}) \) refers to the entire set of input parameter identifier names in a \( \text{PS}(\text{qc}) \).

![Ada Subtype Hierarchy](image)

Figure 20. Ada Subtype Hierarchy

to the parameter type for a specified input parameter \( p \) in a \( \text{PS}(\text{qc}) \). The expression \( \text{In}(\text{sbc}) \) refers to the entire set of input parameter identifier names in a \( \text{PS}(\text{sbc}) \), and \( \text{In}(\text{qc}) \) refers to the entire set of input parameter identifier names in a \( \text{PS}(\text{qc}) \).

c. Output Parameters

Output parameter definitions mirror those of input parameters. The expression \( \text{output_type}(p, \text{sbc}) \) refers to the parameter type for a specified output parameter \( p \) in a \( \text{PS}(\text{sbc}) \). Similarly, the expression \( \text{output_type}(p, \text{qc}) \) refers to the parameter type for a specified output parameter \( p \) in a \( \text{PS}(\text{qc}) \). The expression \( \text{Out}(\text{sbc}) \) refers to the entire set of output parameter identifier names in a \( \text{PS}(\text{sbc}) \), and \( \text{Out}(\text{qc}) \) refers to the entire set of output parameter identifier names in a \( \text{PS}(\text{qc}) \).
d. States

The expression \( ST(sbc) \) is a boolean function that evaluates whether the software base component is a state machine or not. \( ST(qc) \) does the same for a query component.

e. Abstract Data Types

Type components may contain type declarations for abstract data types utilized by the type component. This is not the case for operator components. \( ADT(sbc) \) denotes the set of all abstract data types in a software base type component, while \( ADT(qc) \) represents the set of all abstract data types in a query type component. \( OPS(sbc) \) denotes the set of all operators in a software base type component, while \( OPS(qc) \) represents the set of all operators in a query type component. \( Tot\_In(sbc) \) denotes the entire set of input parameter identifier names over all operators of a software base type component, while \( Tot\_In(qc) \) denotes the entire set of input parameter identifier names over all operators of a query type component. Similarly \( Tot\_Out(sbc) \) and \( Tot\_Out(qc) \) are defined for output operators.

E. SYNTACTIC MATCHING RULES FOR TYPE COMPONENTS

The following rules for component matching are again taken from McDowell [Ref. 9] and Dolgoff [Ref. 5]. \( NUM(X) \) is used to represent the cardinality of the set \( X \).

1. Rules for Operator Components

Initial filtering for operator components is based upon comparing numbers of parameters between two operators. these are listed below:

- \( NUM(In(sbc)) = NUM(In(qc)) \)
- \( NUM(Out(sbc)) \geq NUM(Out(qc)) \)
- \( ST(sbc) = ST(qc) \)

The number of software base component input parameters must be the same as those of the query component. The number of software base component output parameters must be
greater than or equal to those of the query component. Both components must either be state machines, or both components must not be state machines.

Extended filtering rules for operator components were specified by Dolgoff and follow below [Ref. 5]:

\textbf{a. Property 1}

There must exist a bijective function \( f \) from the set \( \text{In}(qc) \) to the set \( \text{In}(sbc) \) for which the following holds:

\[
(\forall p \in \text{In}(qc)) \ (\text{input_type}(p, qc) \subseteq \text{input_type}(f(p), sbc))
\]

where the subset operator denotes “is a subtype of”.

\textbf{b. Property 2}

There must exist a one-to-one function \( f \) from the set \( \text{Out}(qc) \) to the set \( \text{Out}(sbc) \) for which the following holds:

\[
(\forall p \in \text{Out}(qc)) \ (\text{output_type}(f(p), sbc) \subseteq \text{output_type}(p, qc))
\]

These two rules enforce signature matching for operator components.

\textbf{2. Rules for Type Components}

PSDL \textit{type} components contain one or more abstract data type declarations and zero or more operator. Initial filtering of software base \textit{type} components is based upon aggregate signatures composed from the type’s operator components. The basic rules for types are [Ref. 9]:

- \( \text{NUM}(\text{ADT}(sbc)) \geq \text{NUM}(\text{ADT}(qc)) \)
- \( \text{NUM}(\text{Tot}_\text{In}(sbc)) \geq \text{NUM}(\text{Tot}_\text{In}(qc)) \)
- \( \text{NUM}(\text{Tot}_\text{Out}(sbc)) \geq \text{NUM}(\text{Tot}_\text{Out}(qc)) \)
- \( \text{NUM}(\text{OPS}(sbc)) \geq \text{NUM}(\text{OPS}(qc)) \)

The number of ADTs, operators, total operator inputs and total operator outputs within the software base \textit{type} component must all be greater than or equal to those of the query \textit{type} component.
Extended filtering for type components as specified by Dolgoff, consists of a single rule which states that there must exist a one-to-one function $f$ from the set $OPS(qc)$ to the set $OPS(sbc)$ such that $\forall OP_{qc} \in OPS(qc)$ properties one and two above, for operators, hold true [Ref. 5]. In addition to the rules specified by Dolgoff, properties one and two for operators must be satisfied by type components as well, in order to match the aggregate signatures for component inputs and outputs.

F. SEARCHING FOR TYPE COMPONENTS

The process for matching type components is shown in Figure 21. This is a slightly modified version of the process presented by Dolgoff [Ref. 5]. The Array Check step has been removed, as it is performed within the Operator Mapping sub-process. All filtering prior to Operator Mapping is accomplished by database queries to the CAPS software base, as described by Dolgoff. Operator Mapping and the Instantiation Check are discussed in the following sections.

Figure 21. Matching Process for Type Components
1. Operator Mapping

Operator Mapping is the process that determines whether a one-to-one function can be found that maps OPS(qc) to OPS(sbc). In order to do this, each operator component, within the query type component, is formulated into a query operator component, and used as input to existing operator matching functions. Each of these query operator components is matched against a set derived from OPS(sbc) in search of a match. Should a match be found for an operator component, it becomes part of the Operator Mapping function, and the matching software base operator component is added to a set Used_OPS(sbc). Used_OPS(sbc) is initialized to the empty set. The set of available software base operator components input to the process is, OPS(sbc) - Used_OPS(sbc). This assures that a one-to-one mapping will be generated should the process succeed. If no match can be found for an operator component, backtracking is utilized in order to achieve an exhaustive search for a suitable one-to-one mapping. If no one-to-one mapping can be generated, then the entire Operator Mapping step fails for that particular software base type component, and it is eliminated from consideration as a match for PS(sbc). The Operator Mapping process is shown in Figure 22 and is derived from Dolgoff's filter process for operators. The Is

![Figure 22. Operator Mapping Sub-Process](image-url)
Generic step is a modified version of the same step used by Dolgoff. Is Generic branches in the following manner:

- **Yes** - The Yes branch can be taken for either of the following two reasons. First, the software base *operator* component is generic. Second, if the software base *type* component which contains the software base *operator* component is generic, and one or more of the *operator’s* input or output parameter types matches one of the *type* component’s declared generic parameters (see example Figure 23).

- **No** - The No branch is taken if the *operator* component is non-generic, and none of its input or output parameters match any generic parameters which belong to the *type* component in which it is contained.

The Array Check step performed across the set of $\text{OPS(qc)}$, throughout the process of Operator Mapping, removes the need to perform Array Check separately in the *type* matching process.

2. **Type Instantiation Check**

At the Instantiation Check stage, generic *type* components are evaluated to determine if a proper set of actual Ada type parameters can be found to properly instantiate the *type* component to match the query *type* component. If no set of Ada types can be found to perform the instantiation, the software base *type* component is removed from consideration as a possible match for the query component. For non-generic software base *type* components, this stage is simply a pass through filter, and the previous stages have demonstrated that the component is a syntactic match for the query component. For generic software base *type* components, the Operator Mapping stage has shown that at the *operator* level, generic instantiations exist which match all *operators* contained in the query *type* component. A problem exists in the fact that the generic parameters are defined at the *type* level rather than at the *operator* in generic *type* components. Figure 23 shows an example of a generic *type* component. Potential problems lie in the manner in which the Operator Mapping phase assigns to actual Ada types to the parameter Discrete1. Figure 24 shows an example query *type* component which can be matched by the *type* component in Figure 23. In this example, the Operator Mapping process would assign Discrete1 to Ada type Positive when instantiating Op1, and it would assign Discrete1 to Ada type Integer when instantiating Op2. The problem is to find a single type to assign to Discrete1 which will
create suitable instantiations for both Op1 and Op2. The solution is to utilize the subtype

```ada
type GenericExample
specification
  generic
    Discrete1 : Discrete_Type
  operator Op1
    specification
      input
        Op1Parm1 : Discrete1
    end
  operator Op2
    specification
      input
        Op2Parm1 : Discrete1
      output
        Op2Parm2 : Float
    end
end
```

**Figure 23. Example Generic Type Specification**

hierarchies specified by Dolgoff [Ref. 5]. In cases where two or more operator components utilize the same generic parameter in declaring input or output parameters, the concept of least upper bound is used to determine a proper instantiation of the type component. Considering the Ada types selected for instantiation of operators within the software base type component as a set, a proper instantiation is possible only if the least upper bound of the set is a specific Ada type, as opposed to the types which may only appear in generic formal parameter declarations (shown in Figure 11). In attempting to match the query component in Figure 24 to the software base component of Figure 23, the set of possibilities for the Discrete1 parameter is found to be \{Integer, Positive\}. Referring to Figure 20, the Ada type Integer is found to be the least upper bound. Integer is a specific Ada type, and is therefore selected as the appropriate type with which to instantiate the software base component. In Figure 25, an example of an unsuccessful instantiation is shown. The set of
possibilities for Discrete1 in this case is \{Positive, String\}, and the least upper bound of this set is the Ada type \textit{Private}. Private is not a specific Ada type, and the conclusion is that GenericExample can not be instantiated in such a way as to match QueryExample2. There is no single Ada type with which GenericExample can be instantiated which contains both a String and an Integer. GenericExample is then removed from consideration as a matching \textit{type} component.

```ada
type QueryExample
  specification
    operator Op1
      specification
        input
          Op1Parm1 : Positive
      end
    operator Op2
      specification
        input
          Op2Parm1 : Positive
        output
          Op2Parm2 : Float
      end
  end
```

\textbf{Figure 24. Example Query Type Component (Successful Match)}
type QueryExample2
    specification
        operator Op1
            specification
                input
                    Op1Parm1 : Positive
            end
        end
    operator Op2
        specification
            input
                Op2Parm1 : String
            output
                Op2Parm2 : Float
        end
    end

Figure 25. Example Query Type Component (Unsuccessful Match)
VI. CONCLUSIONS AND FUTURE RESEARCH

A. CONCLUSIONS

The primary goal of this thesis was to produce a tool with the capability to automatically produce a PSDL specification when given an Ada package specification as input. This translation tool was produced utilizing the Synthesizer Generator, and has demonstrated its effectiveness by successfully translating several complex components from the Common Ada Missile Packages library. Additional accomplishments include the extension of PSDL constructs to allow the use of Ada procedures and functions as generic formal parameters, and extended considerations for the retrieval of PSDL type components from the CAPS software base. The following sections discuss areas in which further work may be accomplished to build upon the work of this thesis.

B. POPULATE THE CAPS SOFTWARE BASE

The completion of the translation tool presented in this thesis provides the opportunity to populate the CAPS software base by bringing in components from a variety of DOD Ada software libraries. These libraries include, but are not limited to, the CAMP, RAPID, ASSET, and CRSS libraries. Population of the software base will greatly enhance the ability of CAPS users to build significant, useful, prototype systems.

C. EXTEND THE CAPABILITIES OF CURRENT COMPONENT RETRIEVAL TOOLS

The current retrieval tool utilized by CAPS is capable of retrieving PSDL operator components only. Two major restrictions were imposed on these retrieval operations due to Ada to PSDL mapping limitations which existed at the time the retrieval tool was created. The first restriction prevented the use of in out parameters as procedure arguments. The second restriction prevented the use of functions and procedures as generic formal parameters. Updated translation tools and further review of PSDL have removed these two restrictions. First, in out parameters are now allowed as formal arguments within procedures, and second, mapping schemes have been created to allow functions and
procedures to be used as generic formal parameters. Dolgoff's retrieval tool needs updating to handle these two new cases. Additionally, Dolgoff's tool was created to handle Ada 83 packages, and with the introduction of Ada 95, further research will be required to determine how derived types, and generic derived types can be made to fit into Dolgoff's type hierarchy.

D. IMPLEMENT PSDL TYPE COMPONENT RETRIEVAL

Scott Dolgoff's work created a tool which is used to retrieve PSDL operator components from the CAPS Software Base. This thesis extends the discussion on methods for the retrieval of PSDL type components from the software base. These methods require some further refinement followed by an actual implementation and integration with the operator retrieval tool to provide a complete PSDL component retrieval suite.
LIST OF REFERENCES


APPENDIX A. SSL SOURCE CODE: ABSTRACT SYNTAX

The source code below comprises two files which specify the abstract syntax for Ada 95 package specifications and for PSDL.

/* File: abstract.ada9x.ssl */
/* Date: 3 March, 1995 */
/* Author: Chris Eagle */
/* System: Sun SPARCstation */
/* Description: This file contains the abstract grammar for that */
/* portion of the Ada9x language which is required for */
/* package specifications. It was derived from the YACC */
/* grammar noted below. */
/* ************************************************************************* */
/* This grammar is thought to be correct as of May 1, 1994 */
/* but as usual there is *no warranty* to that effect. */
/* ************************************************************************* */

/* Lexemes for concrete syntax. This specification accounts for */
/* Ada's type insensitivity. */

QUOTED_STRING : <"(([^\"\"]|\\")\")\" >;
CHAR_LIT : <'(.[n])'>;
TIC : <"\"" >;
DOT_DOT : <".." >;
BOX : <"<>" >;
LT_EQ : <"<=" >;
LT_LT : <"<" >;
EXPON : <"**" >;
NE : <"/=" >;
GE : <">=" >;
GT_GT : <">>" >;
IS_ASSIGNED : <"==" >;
RIGHT_SHAFT : <"=>" >;
ABORT : <[aA][bB][oO][rR][tT] >;
ABS : <[aA][bB][sS] >;
ABSTRACT : <[aA][bB][sS][tT][rR][aA][cC] >;
ACCEPT : <[aA][cC][eE][pP] >;
ACCESS : <[aA][cC][eE][sS][sS] >;
ALIASED : <[aA][iI][A][sS][eE][dD] >;
ALL  :< [aA][IL][IL] >;
AND  :< [aA][nN][dD] >;
ARRAY :< [aA][rR][rR][aA][yY] >;
AT   :< [aA][tT] >;
BEGIN :< [bB][eE][gG] >;
BoDY :< [bB][oO][yY] >;
CASE  :< [cC][aA][sS][E] >;
CONSTANT :< [cC][nN][sS][tT][aA][nN] >;
DECLARE :< [dD][eE][cC][aA][rR][eE] >;
DELAY :< [dD][eE][I][aA][yY] >;
DELTA :< [dD][eE][I][tT][aA] >;
DIGITS :< [dD][iI][gG][iI][T][S][S][S] >;
DO   :< [dD][oO] >;
ELSE  :< [eE][I][L][S][I][S][E] >;
ELSIF :< [eE][I][L][S][I][I][F] >;
END  :< [eE][nN][D] >;
ENTRY :< [eE][xX][iI][T][S][T][T] >;
EXCEPTION :< [eE][xX][cC][eE][pP][tT][iI][oO][nN] >;
EXECUTION :< [eE][xX][eE][cC][uU][T][T][I][O][O][N] >;
EXIT :< [eE][xX][iI][T] >;
FOR   :< [fF][oO][rR] >;
FUNCTION :< [fF][uU][nN][cC][T][T][I][O][O][N] >;
GENERIC :< [gG][eE][nN][eE][rR][iI][cC] >;
GOTO  :< [gG][oO][T][O] >;
HRS :< [hH][oO][U] >;
LIMITED :< [lL][iI][M][M][I][T][T] >;
LOOP :< [lL][oO][oO][O][P] >;
MAXIMUM :< [mM][iI][M][M][U] >;
MIN  :< [mM][iI][N] >;
MOD  :< [mM][O][D][D] >;
MS   :< [mM][S][S] >;
NEW  :< [nN][E][W] >;
NOT :< [nN][O][R][T] >;
NuLL :< [nN][U] >;
OF   :< [O][F] >;
OR   :< [O][R] >;
OTHERS :< [O][T][H][E][R][S] >;
PACKAGE :< [P][P][A][C][K][A][G][E] >;
PrAGMA :< [P][P][R][A][G][M][A] >;
PRIVATE :< [P][P][R][I][V][A][T][E] >;
PROCEDURE :< [P][P][R][O][C][E][D][U][R][E] >;
PROTECTED :< [P][P][R][O][T][E][C][T][E][D] >;
RAISE :< [R][A][I][S][E] >;
RaNGE :< [R][A][N][G][E] >;
RECORD :< [R][E][C][O][R][D] >;
REM :< [R][E][M] >;
ReNAMES :< [R][E][N][A][M][E][S] >;
REQUEUE :< [r][e][[q][u][e][u] >;
RETURN :< [r][e][i][n][r][n] >;
REVERSE :< [r][e][v][e][r][s][e] >;
SEC :< <PSDL_STATE> [s][e][c] <INITIAL> >;
SELECT :< [s][e][l][e][c] >;
SEPARATE :< [s][e][p][a][r][a][t] >;
SUBTYPE :< [s][u][b][t][y][p] >;
TAGGED :< [t][a][g][g][e] >;
TASK :< [t][a][s][k] >;
TERMINATE :< [t][e][r][m][i][n][a][t] >;
THEN :< [t][h][e] >;
TIME :< <PSDL_STATE> [t][i][m][e] >;
TYPE :< [t][y][p][e] >;
UNTIL :< [u][n][t][i][l] >;
USE :< [u][s][e] >;
USEC :< <PSDL_STATE> [m][i][l][i][l][m][e] >;
WHEN :< [w][h][e][n] >;
WH-ILE :< [w][h][i][l] >;
WHITESPACE :< [\t\n] >;
WITH :< [w][i][t][h] >;
XOR :< [x][o][r] >;

PSDL_COMMENT : < "--PSDL " <PSDL_STATE> >;
ADA_COMMENT : < "-" >;
INTEGER : <[0-9]-?[0-9]* >;
REAL_CS : <[0-9]-?[0-9]*\.[0-9]-?[0-9]* >;
ID : <[a-zA-Z]?[a-zA-Z0-9]-?[a-zA-Z0-9]* >;

/* precedence declarations */

left ',', ';',
left AND, OR, XOR,
left '!', NE, LT, LT_EQ, GT, GE,
left '+', '-', '&',
left '*', '/', MOD, REM,
left EXPON,
left '.';

root compilation;

comp_unit, pkg_decl, pkg_spec, private_part, decl_item_s, decl_item, decl
{inh INT nesting_level;};

compilation : CompilationNil()
    | Compilation(pragmas comp_unit_list)
    :

optional list comp_unit_list;
comp_unit_list : CUListNil()
    | CUList(comp_unit comp_unit_list) {
      comp_unit.nesting_level = 0;
    }
    |

comp_unit : CompUnit(context_spec_opt private_opt pkg_decl pragma_s) {
  pkg_decl.nesting_level = $$.nesting_level;
}
|

optional private_opt;
private_opt : PrivateOptNull()
  | PrivateOptPrompt()
  | PrivateOpt()
  |

optional context_spec_opt;
context_spec_opt : ContextSpecNull()
  | ContextSpecPrompt()
  | ContextSpec(context_spec)
  |

custom spec : EmptyContextSpec()
  | ContextWithUse(context_spec_opt with_clause use_clause_opt)
  | ContextPragma(context_spec pragma)
  |

with_clause : WithClause(c_name_list)
  |

optional list use_clause_opt;
use_clause_opt : UseClauseOptNil()
  | UseClause(use_clause use_clause_opt)
  |

use_clause : EmptyUseC()
  | Use(name_s)
  | UseType(name_s)
  |

list name_s;
name_s : NameNil()
  | nameList(name name_s)
  |

name : EmptyName()
  | SimpleName(identifier)
  | IndexComp(name value_s)
  | SelectedComp(selected_comp)
  | Attribute(name attribute_id)
OperatorSymbol(QUOTED_STRING)

identifier : IdNull()
  | Ident(ID)

list value_s:
  value_s : ValueNil()
    | ValueList(value value_s)

value : EmptyValue()
  | ValueExpr(expression)
    | ValueCompAssoc(comp_assoc)
    | ValueDiscWithRange(discrete_with_range)

selected_comp : EmptySelComp()
  | DotId(name identifier)
  | DotUsedChar(name CHAR_LIT)
  | DotString(name QUOTED_STRING)
  | DotAll(name)

attribute_id : EmptyAttribId()
  | AttribId(identifier)
  | AttribDigits()
  | AttribDelta()
  | AttribAccess()

expression : EmptyExpression()
  | Relation(relation)
    | And, Or, Xor, AndThen, OrElse(expression relation)

relation : EmptyRelation()
  | SimpleExpr(simple_expression)
    | Equal, NotEqual, LessThan, LessThanEq,
    | GreaterThan, GreaterThanEq(simple_expression simple_expression)
    | RangeMember(simple_expression membership range)
    | NameMember(simple_expression membership name)

membership : EmptyMembr()
  | In()
  | NotIn()

simple_expression : EmptySimple()
optional unary:
unary : UnaryNull()
    | UnaryPrompt()
    | Plus()
    | Minus()
    ;

term : EmptyTerm()
    | Factor(factor)
    | Mult, Divide, Mod, Rem(term factor)
    ;

factor : EmptyFactor()
    | Primary(primary)
    | NotPrimary(primary)
    | AbsPrimary(primary)
    | Expon(primary primary)
    ;

primary : EmptyPrimary()
    | Literal(literal)
    | PrimaryName(name)
    | Allocator(allocator)
    | Qualified(qualified)
    | Parens(expression)
    | PrimaryAgg(aggregate)
    ;

list compound_name;
compound_name : EmptyCompound()
    | DotCompound(identifier compound_name)
    ;

c_name_list : CompoundNameNil()
    | CompoundList(compound_name c_name_list)
    ;

numeric_lit : IntLit(integer)
    | RealLit(REAL_CS)
    ;

literal : EmptyLiteral()
    | NumLit(numeric_lit)
    | UsedChar(CHAR_LIT)
    | NilLit()
    ;
aggregate : EmptyAggregate()
    | AggCompAssoc(comp_assoc)
    | AggValues2(value_s_2)
    | AggExprValue(expression value_s)
    | AggExprWithNull(expression)
    | AggExprNullRec()

value_s_2 : Value2Pair(value value)
    | Value2List(value_s_2 value)

comp_assoc : CompAssoc(choice_s expression)
    ;

list choice_s;
choice_s : ChoiceNil()
    | ChoiceList(choice choice_s)
    ;

choice : EmptyChoice()
    | ChoiceExpr(expression)
    | ChoiceRange(discrete_with_range)
    | ChoiceOthers()
    ;

discrete_with_range : DiscreteNameRange(name range_constraint)
    | DiscreteWithRange(range)
    ;

range_constraint : Range(range)
    ;

range : EmptyRange()
    | SimpleRange(simple_expression simple_expression)
    | NameTicRange(name)
    | NameTicRangeExp(name expression)
    ;

qualified : EmptyQual()
    | NameTicAgg(name aggregate)
    | NameTicExpr(name expression)
    ;

allocator : newName(name)
    | NewQualified(qualified)
    ;

pragma : EmptyPragma()
    | PragmaId(identifier)

53
PragmaSimple(identifier pragma_arg_s)

list pragma_arg_s;
pragma_arg_s : PragmaArgNil()
   | PragmaSargList(pragma_arg pragma_arg_s)

pragma_arg : EmptyPragmaArg()
   |PragmaExp(expression)
   |PragmaNameExp(identifier expression)

optional list pragma_s;
pragma_s : PragmasNil()
   | PragmasList(pragma pragma_s)

pkg_decl : EmptyPkgDecl()
   |PkgSpec(generic_hdr pkg_spec) {
pkg_spec.nesting_level = $$.nesting_level;
}
   |GenPkglnst(compound_name generic_inst)

pkg_spec : Package(compound_name decl_item_s private_part) {
decl_item_s.nesting_level = $$.nesting_level + 1;
private_part.nesting_level = $$.nesting_level;
}

optional private_part;
private_part : PrivatePartNull()
   |PrivatePartPrompt()
   |Private(decl_item_s) {
decl_item_s.nesting_level = $$.nesting_level + 1;
}

optional list decl_item_s;
decl_item_s : DeclListNil()
   |DeclList(decl_item decl_item_s) {
decl_item_s$2.nesting_level = $$.nesting_level;
decl_item.nesting_level = $$.nesting_level;
}

decl_item : EmptyDeclItem()
   |Decl(decl) {
decl.nesting_level = $$.nesting_level;
}
IUseClauseDecl(use_clause)
| DeclRepSpec(rep_spec)
| DeclPragma(pragmas)
|
rep_spec : EmptyRepSpec()
| AttribDef(mark expression)
| RecordTypeSpec(mark align_opt comp_loc_s)
| AddressSpec(mark expression)
|
optional align_opt;
align_opt : AlignOptNull()
| AlignOptPrompt()
| AlignOpt(expression)
|
optional comp_loc_s;
comp_loc_s : CompLocNull()
| CompLocPrompt()
| CompLocList(comp_loc_s mark expression range)
|
mark : EmptyMark()
| Mark(identifier marklist)
|
optional tiedot;
tiedot : TicDotNil()
| TicDotPH()
| TicOpt(attribute_id)
| DotOpt(identifier)
|
optional list marklist;
marklist : MarkListNil()
| MarkList(tiedot marklist)
|
decl : EmptyDecl()
| ObjDecl(def_id_s object_qualifier_opt object_subtype_def init_opt)
| NumDecl(def_id_s expression)
| TypeDecl(identifier discrim_part_opt type_completion)
| SubTypeDecl(identifier subtype_ind)
| SubProgDecl(subprog_decl)
| PkgDecl(pkg_decl) {
pkg_decl.nesting_level = $$.nesting_level;
}
| TaskDecl(task_spec)
| ProtDecl(prot_spec)
| ExcDecl(def_id_s)

55
list def_id_s;
def_id_s : DefIdNil()
    | DefIdList(identifier def_id_s)

optional object_qualifier_opt;
object_qualifier_opt : ObjQualOptNull()
    | ObjQualOptPrompt()
    | Aliased()
    | Constant()
    | AliasedConst()

object_subtype_def : EmptySubtypeDef()
    | SubtypeInd(subtype_ind)
    | ArrayType(array_type)

optional init_opt;
init_opt : InitOptNull()
    | InitOptPrompt()
    | ExprInitOpt(expression)

subtype_ind : EmptySubInd()
    | SubtypeIndConstraint(name constraint)
    | SubTypeIndName(name)

constraint : EmptyConstraint()
    | RangeConstraint(range_constraint)
    | DecDigConstraint(expression range_constr_opt)

range_constr_opt : EmptyRangeConstrOpt()
    | RangeConstr(range_constraint)

array_type : EmptyArrayType()
    | UnconstrArray(index_s component_subtype_def)
    | ConstrArray(iter_discrete_range_s component_subtype_def)

cOMPONENT_subtype_def : CompSubtypeDef(aliased_opt subtype_ind)

optional aliased_opt;
aliased_opt : AliasedOptNull()
               | AliasedOptPrompt()
               | AliasedOpt()
               
list index_s,
index_s : IndexNil()
               | IndexList(name index_s)
               
list iter_discrete_range_s,
iter_discrete_range_s : DiscreteRangeNil()
               | DiscreteRangeList(discrete_range iter_discrete_range_s)
               
discrete_range : EmptyDiscRng()
               | DiscRangeName(name range_constr_opt)
               | DiscRangeRange(range)
               
optional discrim_part_opt;
discrim_part_opt : DiscrimPartNull()
               | DiscrimPartPrompt()
               | DiscrimPart(discrim_spec_s)
               | Box()
               
list discrim_spec_s,
discrim_spec_s : DiscrimSpecNil()
               | DiscrimSpecList(discrim_spec discrim_spec_s)
               
discrim_spec : DiscrimSpecDef(def_id_s access_opt mark init_opt)
               
optional access_opt;
access_opt : AccessOptNull()
               | AccessOptPrompt()
               | AccessOpt()
               
optional type_completion;
type_completion : TypeComplNull()
               | TypeComplPrompt()
               | TypeDefCompl(type_def)
               
type_def : EmptyTypeDef()
               | EnumTypeDef(enum_id_s)
               | IntTypeDef(integer_type)
               | RealTypeDef(real_type)
| ArrayTypeDef(array_type) |
| RecordTypeDef(tagged_opt limited_opt record_def) |
| AccessTypeDef(access_type) |
| DerivedTypeDef(derived_type) |
| PrivateTypeDef(private_type) |

\[
\text{derived_type : EmptyDerivedType()}
| \text{NewDerivedType(subtype_ind)}
| \text{NewDerivedWithPrivate(subtype_ind)}
| \text{NewDerivedWithRecord(subtype_ind record_def)}
| \text{AbsNewDerivedWithPrivate(subtype_ind)}
| \text{AbsNewDerivedWithRecord(subtype_ind record_def)}
\]

\[
\text{list enum_id_s;}
\text{enum_id_s : EnumIdNil()}
| \text{EnumIdList(enum_id enum_id_s)}
\]

\[
\text{enum_id : EmptyEnumId()}
| \text{Id(identifier)}
| \text{CharLit(CHAR_LIT)}
\]

\[
\text{integer_type : EmptyIntType()}
| \text{RangeSpec(range_spec)}
| \text{ModExpr(expression)}
\]

\[
\text{range_spec : RangeSpecConstr(range_constraint)}
\]

\[
\text{record_def : EmptyRecordDef()}
| \text{Record(pragma_s comp_list)}
| \text{NullRecord()}
\]

\[
\text{comp_list : EmptyCompList()}
| \text{CompListWithVariant(comp_decl_s variant_part_opt)}
| \text{CompListWithPragma(variant_part pragma_s)}
| \text{NullWithPragma(pragma_s)}
\]

\[
\text{comp_decl_s : CompDeclNil()}
| \text{CompDecl(comp_decl)}
| \text{CompDeclList(comp_decl_s pragma_s comp_decl)}
\]

\[
\text{variant_part_opt : EmptyVariantPart()}
| \text{VariantPartOptPragma(pragma_s)}
\]

58
| VariantPartOpt(pragma_s variant_part pragma_s) |
| comp_decl : CompDeclDefs(def_id_s component_subtype_def init_opt) |
| variant_part : VariantPart(identifier pragma_s variant_s) |
| list variant_s; |
| variant_s : VariantNil() |
| variant : VariantChoice(choice_s pragma_s comp_list) |
| optional tagged_opt; |
| tagged_opt : TaggedOptNull() |
|          | TaggedOptPrompt() |
|          | Tagged() |
|          | AbstractTagged() |
| optional range_spec_opt; |
| range_spec_opt : RangeSpecOptNull() |
|          | RangeSpecOptPrompt() |
|          | RangeSpecOpt(range_spec) |
| real_type : EmptyRealType() |
|          | FloatType(expression range_spec_opt) |
|          | FixedType(fixed_type) |
| fixed_type : EmptyFixedType() |
|          | FixedDelta(expression range_spec) |
|          | FixedDeltaDigits(expression expression range_spec_opt) |
| private_type : PrivateType(tagged_opt limited_opt) |
| optional limited_opt; |
| limited_opt : LimitedOptNull() |
|          | LimitedOptPrompt() |
|          | Limited() |
| subprog_decl : EmptySubpDecl() |
|          | SubprogSpec(generic_hdr subprog_spec psdl_met_opt) |
| GenericSubprogInst(subprog_spec generic_inst psdl_met_opt) |
| AbstractSubprogSpec(subprog_spec psdl_met_opt) |

optional psdl_met_opt;
psdl_met_opt : MetNull() |
| MetPrompt() |
| MetUsec, MetMs, MetSec, MetMin, MetHrs(integer) |

subprog_spec : EmptySubpSpec() |
| SubProgProc(compound_name formal_part_opt) |
| SubProgFuncReturn/designator formal_part_opt name |
| SubProgFunc/designator |

designator : EmptyDesignator() |
| DesigCompound(compound_name) |
| DesigString(QUOTED_STRING) |

optional formal_part_opt;
formal_part_opt : FormalPartOptNull() |
| FormalPartOptPrompt() |
| FormalPart(param_s) |

list param_s;
param_s : ParamNil() |
| ParamList(param param_s) |

param : ParamId(def_id_s mode mark init_opt) |
| EmptyParam() |

optional mode;
mode : ModeNull() |
| ModePrompt() |
| InMode() |
| OutMode() |
| InOutMode() |
| AccessMode() |

task_spec : EmptyTaskSpec() |
| SimpleTask(identifier task_def) |
| TaskType(identifier discrim_part_opt task_def) |

optional task_def;
task_def : TaskDefNull()
    | TaskDefPrompt()
    | TaskDef(entry_decl_s rep_spec_s task_private_opt)
    ;

optional task_private_opt;
task_private_opt : TaskPvtOptNull()
    | TaskPvtOptPrompt()
    | TaskPvtOpt(entry_decl_s rep_spec_s)
    ;

entry_decl_s : EntryDeclPragma(pragma_s)
    | EntryDeclPragmaList(entry_decl_s entry_decl pragma_s)
    ;

entry_decl : EmptyEntryDecl()
    | EntryDeclId(identifier formal_part_opt)
    | EntryRange(identifier discrete_range formal_part_opt)
    ;

optional rep_spec_s;
rep_spec_s : RepSpecNull()
    | RepSpecPrompt()
    | RepSpecList(rep_spec_s rep_spec pragma_s)
    ;

prot_spec : EmptyProtSpec()
    | Prot(identifier prot_def)
    | ProtType(identifier discrim_part_opt prot_def)
    ;

prot_def : ProtDef(prot_op_decl_s prot_private_opt)
    ;

optional prot_private_opt;
prot_private_opt : ProtPvtOptNull()
    | ProtPvtOptPrompt()
    | ProtPvtOpt(prot_elem_decl_s)
    ;

optional list prot_op_decl_s;
prot_op_decl_s : ProtOptDeclListNil()
    | ProtOptDeclList(prot_op_decl prot_op_decl_s)
    ;

prot_op_decl : EmptyProtOpDecl()
    | EntryDecl(entry_decl)
    | ProtOptSubprog(subprog_spec)
    | RepSpec(rep_spec)
    | ProtOptPragma(pragma)
    ;
optional list prot_elem_decl_s;
prot_elem_decl_s : ProtElemDeclNil()
                 | ProtElemDeclList(prot_elem_decl prot_elem_decl_s)
                 ;

prot_elem_decl : EmptyProtElem()
                 | ProtOptDecl(prot_op_decl)
                 | ProtElemCompDecl(comp_decl)
                 ;

rename_decl : EmptyRenameDecl()
             | RenameDeclSub(def_id_s object_qualifier_opt subtype_ind renames)
             | RenameExec(def_id_s renames)
             | RenameUnitDecl(rename_unit)
             ;

rename_unit : EmptyRenameUnit()
             | RenamePkg(generic_hdr compound_name renames)
             | RenameSubprog(generic_hdr subprog_spec renames)
             ;

renames : Renames(name)
         ;

optional generic_hdr;
generic_hdr : GenericHdrNil()
            | GenericHdrPrompt()
            | GenericHdr(generic_formal_part)
            ;

optional list generic_formal_part;
generic_formal_part : GenericNil()
                    | GenFormalList(generic_formal generic_formal_part)
                    ;

generic_formal : EmptyGenFormal()
                | GenParm(param)
                | GenTypeParm(identifier generic_discrim_part_opt generic_type_def)
                | GenProcParm(identifier formal_part_opt subp_default)
                | GenFuncParm(designator formal_part_opt name subp_default)
                | GenPkgParmBox(identifier name)
                | GenPkgParm(identifier name)
                | GenUseparm(use_clause)
                ;

optional generic_discrim_part_opt;
generic_discrim_part_opt : GenDiscOptNull()
                        | GenDiscOptPrompt()
                        | GenDisc(discrim_spec_s)
                        | GenBox()
optional subp_default;
subp_default : SubpDefaultNull()
  | SubpDefaultPrompt()
  | SubpDefName(name)
  | SubpDefBox()
;
generic_type_def : EmptyGenTypeDef()
  | GenTypeBox()
  | GenTypeRangeBox()
  | GenTypeModBox()
  | GenTypeDeltaBox()
  | GenTypeDeltaDigBox()
  | GenTypeDigitsBox()
  | GenTypeArray(array_type)
  | GenTypeAccess(access_type)
  | GenTypePriv(private_type)
  | GenTypeDerived(generic Derived_type)
;
generic Derived_type : EmptyGenDerType()
  | GenDerivedSubt(subtype_ind)
  | GenDerivedSubtPriv(subtype_ind)
  | GenDerivedAbst(subtype_ind)
;
integer : IntNull()
  | Integer(INTEGER)
;
access_type : EmptyAccessType()
  | AccessSubtype(subtype_ind)
  | AccessConstSubtype(subtype_ind)
  | AccessAllSubtype(subtype_ind)
  | AccessProcedure(prot_opt formal_part_opt)
  | AccessFunction(prot_opt formal_part_opt mark)
;
optional prot_opt;
prot_opt : ProtOptNull()
  | ProtOptPrompt()
  | Protected()
;
body_stub : EmptyBodyStub()
  | TaskStub(identifier)
  | PkgStub(compound_name)
  | SubprogStub(subprog_spec)
  | ProtStub(identifier)
optional psdl;
psdl : EmptyPsdlO
    | PsdlPH0
    | Component(component)
    |
    list component_s;
component_s : ComponentNil0
    | ComponentList(component component_s)
    |
    component : CompDataType(data_type)
    | CompOperator(operator_imp)
    |
data_type : DataType(compound_name type_spec type_impl)
    |
type_spec : TypeSpec(generic_type_decl type_decl_opt op_list_opt
    | functionality)
    |
    optional generic_type_decl:
generic_type_decl : GTDNil0
    | GTD_PH0
    | GTD(type_decl_s)
    |
    optional type_decl_opt;
type_decl_opt : TDO_Nil()
optional reqmts_trace;
reqmts_trace : RqmtsNil()

optional functionality;
functionality : FuncNil()

optional keywords;
keywords : KW_Nil()

optional informal_desc;
informal_desc : ID_Nil()

optional formal_desc;
formal_desc : FD_Nil()

type_impl : AdaTypeImp(compound_name)

operator_impl : AdaOplmp(compound_name)

list initial_expression_list;
initial_expression_list : InitExpNil()

initial_expression : ExpTrue, ExpFalse()
optional opt_init_exp_list;
opt_init_exp_list : optListNil()
                  | optListPrompt()
                  | optList(initial_expression_list)
                  ;

binary_op : PsdlAnd, PsdlOr, PsdlXor, PsdlLT, PsdlLT,
          PsdlEQ, PsdlGTEQ, PsdlLTEQ, PsdlNE, PsdlAdd,
          PsdlSub, PsdlCat, PsdlMul, PsdlDiv, PsdlMod,
          PsdlRem, PsdlExp()
          ;

unary_op : PsdlNot, PsdlAbs, PsdlNeg, PsdlPos()
          ;

time_unit : TimeuSec, TimeMs, TimeSec, TimeMin, TimeHrs(integer)
          ;
APPENDIX B. SSL SOURCE CODE: UNPARSING RULES

The source code below comprises three files which specify the unparsing rules for
Ada 95 package specifications and for PSDL.

/* ***************************************************/
/* File: unparse.ada9x.ssl */
/* Date: 3 March, 1995 */
/* Author: Chris Eagle */
/* System: Sun SPARCstation */
/* Description: This file contains the unparsing rules for that */
/* package specifications. It was derived from the YACC */
/* grammar noted below. */
/* ***************************************************/

/****************** A YACC grammar for Ada 9X **********************/
/* Copyright (C) Intermetrics, Inc. 1994 Cambridge, MA USA */
/* Copying permitted if accompanied by this statement. */
/* Derivative works are permitted if accompanied by this statement. */
/* This grammar is thought to be correct as of May 1, 1994 */
/* but as usual there is *no warranty* to that effect. */
/* ***************************************************/

style Keyword, Placeholder;

identifier : IdNull[@ ::= "<%S(Placeholder:identifier%S)>"]
  [Ident[^ ::= ^]]

integer : IntNull[@ ::= "<%S(Placeholder:integer%S)>"]
  [Integer[^ ::= ^]]

compilation : CompilationNil[@ ::= "<%S(Placeholder:compilation%S)>"]
  [Compilation[@ ::= @]]

comp_unit_list : CUListNil[@ :
  [CUList[@ ::= ^ @]]

pragma : EmptyPragma[^ ::= "<%S(Placeholder:pragma%S)>>"]
  [PragmaIdl[@ ::= "%S(Keyword:PRAGMA%S)" @
    ";\n    --TRANSLATION_ERROR: pragmas do not translate to PSDL\n"
  ]
  [PragmaSimplel[@ ::= "%S(Keyword:PRAGMA%S)" @ "( "
    ");\n    --TRANSLATION_ERROR: pragmas do not translate to PSDL\n"
  ]


pragma_arg_s : PragmaArgNil[@ :]
    | PragmaSargList[@ ::= ^[",",] @]

pragma_arg : EmptyPragmaArg[^: "<%S(Placeholder:pragma arg%S)>>"]
    | PragmaExp[@ ::= @]
    | PragmaNameExp[@ ::= @] => " @]

pragma_s : PragmasNil[@ :]
    | PragmasList[@ ::= ^"%n" @]

decl : EmptyDecl[^: "<%S(Placeholder:declaration%S)>>"]
    | ObjDecl[^: @ " : " @ @ @ "]
    | NumDecl[^: @ " : " %S(Keyword:CONSTANT%S) := " @."]
    | TypeDecl[^: "%S(Keyword:TYPE%S) " @ @ @ "]
    | SubTypeDecl[^: "%S(Keyword:SUBTYPE%S) " @ IS " @."]
    | SubProgDecl[^: @]
    | PkgDecl[^: @]
    | TaskDecl[^: @ "]
    | ProtDecl[^: @ "]
    | ExcDecl[^: @ " : " %S(Keyword:EXCEPTION%S)."
    | RenameDecl[^: @]
    | BodyStubDecl[^: @]

def_id_s : DefIdNil[@ :]
    | DefIdList[@ ::= ^[",",] @]

object_qualifier_opt : ObjQualOptNull[@ :]
    | ObjQualOptPrompt[@ ::= "<%S(Placeholder:qualifier%S)>>"]
    | Aliased[^: "%S(Keyword:ALIASED%S) "]
    | Constant[^: "%S(Keyword:CONSTANT%S) "]
    | AliasedConst[^: "%S(Keyword:ALIASED CONSTANT%S) "]

object_subtype_def : EmptySubtypeDef[@ ::= "<%S(Placeholder:object subtype def%S)>>"]
    | SubtypeInd[@ ::= @]
    | ArrayType[@ ::= @]

init_opt : InitOptNull[@ :]
    | InitOptPrompt[@ ::= "<%S(Placeholder:initializer%S)>>"]
    | ExprInitOpt[@ ::= " := " @]

discrim_part_opt : DiscrimPartNull[@ :]
    | DiscrimPartPrompt[@ ::= "<%S(Placeholder:discriminant%S)>>"]
DiscrimPart[ @ := “ “ @ ]
Box[ @ := “(<>)” ]

TypeCompletion[ @ := TypeComplNull[ @ ] ]
   TypeComplProm[ ^ := “<%S(Placeholder:type%S)” ]
   TypeDefCompl[ @ := “%S(Keyword:IS%S)” @ ]

TypeDef[ ^ := “<%S(Placeholder:type%S)” ]
   EnumTypeDef[ ^ := (“@”) ]
   IntTypeDef[ ^ := @ ]
   RealTypeDef[ ^ := @ ]
   ArrayTypeDef[ ^ := @ ]
   RecordTypeDef[ ^ := @ ]
   AccessTypeDef[ ^ := @ ]
   DerivedTypeDef[ ^ := @ ]
   PrivateTypeDef[ ^ := @ ]

TypeDefInd[ @ := “<%S(Placeholder:subtype%S)” ]
   SubtypeIndConstraint[ @ := @ ]
   SubtypeIndName[ @ := @ ]

Constraint[ ^ := “<%S(Placeholder:constraint%S)” ]
   RangeConstraint[ @ := @ ]
   DecDigConstraint[ @ := “%S(Keyword:DIGITS%S)” @ ]

DerivedType[ ^ := “<%S(Placeholder:derived%S)” ]
   NewDerivedType[ ^ := “%S(Keyword:NEW%S)” ]
   NewDerivedWithPrivate[ ^ := “%S(Keyword:NEW%S)” @
      “%S(Keyword:WITH PRIVATE%S)” ]
   NewDerivedWithRecord[ ^ := “%S(Keyword:NEW%S)” @
      “%S(Keyword:WITH%S)” ]
   AbsNewDerivedWithPrivate[ ^ := “%S(Keyword:ABSTRACT NEW%S)” @
      “%S(Keyword:WITH PRIVATE%S)” ]
   AbsNewDerivedWithRecord[ ^ := “%S(Keyword:ABSTRACT NEW%S)” @
      “%S(Keyword:WITH%S)” ]

RangeConstraint[ @ := @ ]
   Range[ @ := “%S(Keyword:RANGE%S)” ]

Range[ ^ := “<%S(Placeholder:range%S)” ]
   SimpleRange[ ^ := “ “ @ ]
   NameTicRange[ ^ := “%S(Keyword:RANGE%S)” ]
   NameTicRangeExp[ ^ := “%S(Keyword:RANGE%S)” ]
enum_id_s : EnumIdNil[@ :]
   | EnumIdList[@ := ^ [",",] @]
   |

enum_id : EmptyEnumId[@ := "<%S(Placeholder:enumeration id%S)>"]
   | Id[@ := @]
   | CharLit[@ := @]
   |

integer_type : EmptyIntType[@ := "<%S(Placeholder:int type%S)>"]
   | RangeSpec[@ := @]
   | ModExpr[@ := " %S(Placeholder:MOD%S) @"]
   |

range_spec : RangeSpecConstr[@ := @]
   |

range_spec_opt : RangeSpecOptNull[@ :]
   | RangeSpecOptPrompt[@ := "<%S(Placeholder:range specifier%S)>"]
   | RangeSpecOpt[@ := " @"]
   |

real_type : EmptyRealType[@ := "<%S(Placeholder:real type%S)>"]
   | FloatType[@ := "%S(Placeholder:DIGITS%S) @ @"]
   | FixedType[@ := @]
   |

fixed_type : EmptyFixedType[@ := "<%S(Placeholder:fixed type%S)>"]
   | FixedDelta[@ := "%S(Placeholder:DELTA%S) @ @"]
   | FixedDeltaDigits[@ := "%S(Placeholder:DELTA%S) @ @ %S(Placeholder:DIGITS%S) @ @"]
   |

array_type : EmptyArrayType[@ := "<%S(Placeholder:array type%S)>"]
   | UnconstrArray[@ := "%S(Placeholder:ARRAY%S) (" @ " %S(Placeholder:OF%S) @ " ]
   | ConstrArray[@ := "%S(Placeholder:ARRAY%S) (" @ " %S(Placeholder:OF%S) @ " ]
   |

component_subtype_def : CompSubtypeDef[@ := @ @]
   |

aliased_opt : AliasedOptNull[@ :]
   | AliasedOptPrompt[@ := "<%S(Placeholder:aliased%S)>"]
   | AliasedOpt[@ := " %S(Placeholder:ALIASED%S) @"]
   |

index_s : IndexNil[@ :]
   | IndexList[@ := ^ " %S(Placeholder:RANGE%S) @]" [","] @]
   |
iter_discrete_range_s : DiscreteRangeNil[@ ]
| DiscreteRangeList[@ ::= ^ ["","] @ ]

; 

discrete_range : EmptyDiscRng[@ ::= "<%S(Placeholder: discrete range%)>" ]
| DiscRangeName[@ ::= @ @ ]
| DiscRangeRange[@ ::= @ ]

; 

range_constr_opt : EmptyRangeConstrOpt[@ ::= "<%S(Placeholder: range constr %)>" ]
| RangeConstr[@ ::= " @ ]

; 

record_def : EmptyRecordDef[@ ::= "<%S(Placeholder: record def%)>" ]
| Record[@ ::= "%S(Keyword: RECORD%) @ @ 
%b%n%S(Keyword: END RECORD%)"]
| NullRecord[" : "%S(Keyword: NULL RECORD%)"]

; 

tagged_opt : TaggedOptNull[@ ]
| TaggedOptPrompt[@ ::= "<%S(Placeholder: tagged%)>" ]
| Tagged[^ : "%S(Keyword: TAGGED%) 
AbstractTagged[^ : "%S(Keyword: ABSTRACT TAGGED%) 

; 

comp_list : EmptyCompList[@ ::= "<%S(Placeholder: comp list%)>" ]
| CompListWithVariant[@ ::= @ @ ]
| CompListWithPragma[@ ::= @ @ ]
| NullWithPragma[@ ::= "%S(Keyword: NULL%) @ @ 

; 

comp_decl_s : CompDeclNil[@ ]
| CompDecl[@ ::= @ ]
| CompDeclList[@ ::= @ @ @ ]

; 

variant_part_opt : EmptyVariantPart[@ ::= "<%S(Placeholder: variant part%)>" ]
| VariantPartOptPragma[@ ::= @ ]
| VariantPartOpt[@ ::= @ @ @ ]

; 

comp_decl : CompDeclDefs[@ ::= @ " @ @ 
:n"]

; 

discrim_spec_s : DiscrimSpecNil[@ ]
| DiscrimSpecList[@ ::= ^ [":","] @ ]

; 

discrim_spec : DiscrimSpecDef[@ ::= @ " @ @ ]
access_opt : AccessOptNull[@ :]
   [ AccessOptPrompt[@ ::= "<%S(Placeholder:access%S)>"]
   [ AccessOpt[^ : "%S(Keyword:ACCESS%S)"

variant_part : VariantPart[@ ::= "%S(Keyword:CASE%S)" @
   " %S(Keyword:IS%S)@%n" @ @
   "%b@%n%S(Keyword:END CASE%S),%n"

variant_s : VariantNil[@ :
   [ VariantList[@ ::= ^ ["%n"] @]

variant : VariantChoice[@ ::= "%S(Keyword:WHEN%S)" @ @ %=%t%n" @ @ "%b"

choice_s : ChoiceNil[@ :
   [ ChoiceList[@ ::= ^ ["""] @]

choice : EmptyChoice[@ ::= "<%S(Placeholder:choice%S)>"]
   [ ChoiceExpr[@ ::= @]
   [ ChoiceRange[@ ::= @]
   [ ChoiceOthers[^ : "%S(Keyword:OTHERS%S)"

discrete_with_range : DiscreteNameRange[@ ::= @ " " @]
   [ DiscreteWithRange[@ ::= @]

access_type : EmptyAccessType[^ : "%S(Placeholder:access type%S)>"]
   [ AccessSubtype[^ : "%S(Keyword:ACCESS%S)" @]
   [ AccessConstSubtype[^ : "%S(Keyword:ACCESS CONSTANT%S)" @]
   [ AccessAllSubtype[^ : "%S(Keyword:ACCESS ALL%S)" @]
   [ AccessProcedure[^ : "%S(Keyword:ACCESS%S)" @
      " %S(Keyword:PROCEDURE%S)" @]
   [ AccessFunction[^ : "%S(Keyword:ACCESS%S)" @
      " %S(Keyword:FUNCTION%S)" @
      " %S(Keyword:RETURN%S)" @ "%n"

prot_opt : ProtOptNull[@ :]
   [ ProtOptPrompt[@ ::= "<%S(Placeholder:protected%S)>"]
   [ Protected[^ : "%S(Keyword:PROTECTED%S)"

decl_item_s : DeclListNil[@ :]


```plaintext
| DeclList[@ ::= ^["%n%n"] @] |

decl_item : EmptyDeclItem[@ ::= "<%S(Placeholder:decl item%S)>" ]
  | Decl[@ ::= @] |
  | UseClauseDecl[@ ::= @] |
  | DeclRepSpec[@ ::= @] |
  | DeclPragma[@ ::= @] |

name : EmptyName[@ ::= "<%S(Placeholder:name%S)>"]
  | SimpleName[@ ::= @] |
  | IndexComp[@ ::= @ "(" @ ")"] |
  | SelectedComp[@ ::= @] |
  | Attribute[@ ::= @ "\" "] |
  | OperatorSymbol[@ ::= @] |

mark : EmptyMark[@ ::= "<%S(Placeholder:mark%S)>"]
  | Mark[@ ::= @ @] |

	|

ticdot : TicDotNil[@ ::= ] |
  | TicDotPH[@ ::= "<%S(Placeholder:ATTR or \ID%S)>" ] |
  | TicOpt[@ ::= "\" @] |
  | DotOpt[@ ::= "\" @] |

marklist : MarkListNil[@ ::= ] |
  | MarkList[@ ::= ^[" @] |

compound_name : EmptyCompound[@ :]
  | DotCompound[@ ::= ^["." @] |

c_name_list : CompoundNameNil[@ :]
  | CompoundList[@ ::= ^["," @] |

value_s : ValueNil[@ :]
  | ValueList[@ ::= ^["," @] |

value : EmptyValue[@ ::= "<%S(Placeholder:value%S)>"]
  | ValueExpr[@ ::= @] |
  | ValueCompAssoc[@ ::= @] |
  | ValueDiscWithRange[@ ::= @] |

selected_comp : EmptySelComp[@ ::= "<%S(Placeholder:selected component%S)>"] |
```

75
I
DotUsedChar[@ := @ ""]
| DotUsedChar[@ := @ "]( " @]
| DotString[@ := @ ""]
| DotAll[@ := @ "%.S(Keyword:ALL%)"]

attribute_id : EmptyAttribId[@ := "<%(Placeholder:attribute id%)>" ]
| AttribId[@ := @ ]
| AttribDigits[^ : "%.S(Keyword:DIGITS%)" ]
| AttribDelta[^ : "%.S(Keyword:DELTA%)" ]
| AttribAccess[^ : "%.S(Keyword:ACCESS%)" ]

numeric_lit : IntLit[@ := @ ]
| RealLit[@ := @ ]

literal : EmptyLiteral[@ := "<%(Placeholder:literal%)>" ]
| NumLit[@ := @ ]
| UsedChar[@ := @ ]
| NilLit[^ : "%.S(Keyword:NULL%)"]

aggregate : EmptyAggregate[@ := "<%(Placeholder:aggregate%)>" ]
| AggCompAssoc[@ := "(" @ ")"]
| AggValues2[& := "(" @ ")"]
| AggExprValue[& := "(" @ ")"]
| AggExprWithNull[& := "(" @ ")"]
| AggExpNullRecf[& := "(" @ ")"]

value_s_2 : ValueS2Pair[@ := @ ", " @] 
| ValueS2List[@ := @ ", " @]

comp_assoc : CompAssoc[@ := @ => " @]

expression : EmptyExpression[@ := "<%(Placeholder:expression%)>" ]
| Relation[@ := @ ]
| And[@ := @ " %.S(Keyword:AND%)" @] 
| Or[@ := @ " %.S(Keyword:OR%)" @] 
| Xor[@ := @ " %.S(Keyword:XOR%)" @] 
| AndThen[@ := @ " %.S(Keyword:AND THEN%)" @] 
| OrElse[ @ := @ " %.S(Keyword:OR ELSE%)" @] 

relation : EmptyRelation[@ := "<%(Placeholder:relation%)>" ]
| SimpleExpr[@ := @ ]
| Equal[@ := @ ", = @] 
| NotEqual[@ := @ ", /= @] 

76
| LessThan[@ ::= @ "<" @] |
| LessThanEq[@ ::= @ "<=" @] |
| GreaterThan[@ ::= @ ">" @] |
| GreaterThanEq[@ ::= @ ">= " @] |
| RangeMember[@ ::= @ @ @] |
| NameMember[@ ::= @ @] |

; membership : EmptyMembr[@ ::= "<%S(Placeholder:mbr op%S)>"] |
| In[@ ::= "%S(Keyword:IN%S)"] |
| NotIn[@ ::= "%S(Keyword:NOT IN%S)"] |

; simple_expression : EmptySimple[@ ::= "<%S(Placeholder:simple expr%S)>"] |
| Term[@ ::= @ @] |
| Addition[@ ::= @ " + " @] |
| Subtraction[@ ::= @ " - " @] |
| Concat[@ ::= @ " & " @] |

; unary : UnaryNull[@ ] |
| UnaryPrompt[@ ::= "<%S(Placeholder:unary op%S)>"] |
| Plus[@ ::= "+"] |
| Minus[@ ::= "-"] |

; term : EmptyTerm[@ ::= "<%S(Placeholder:term%S)>"] |
| Factor[@ ::= @] |
| Mult[@ ::= @ " * " @] |
| Divide[@ ::= @ "/" @] |
| Mod[@ ::= @ "%S(Keyword:MOD%S) " @] |
| Rem[@ ::= @ "%S(Keyword:REM%S) " @] |

; factor : EmptyFactor[@ ::= "<%S(Placeholder:factor%S)>"] |
| Primary[@ ::= @] |
| NotPrimary[@ ::= "%S(Keyword:NOT%S) " @] |
| AbsPrimary[@ ::= "%S(Keyword:ABS%S) " @] |
| Expon[@ ::= @ " ** " @] |

; primary : EmptyPrimary[@ ::= "<%S(Placeholder:primary%S)>"] |
| Literal[@ ::= @] |
| PrimaryName[@ ::= @] |
| Allocator[@ ::= @] |
| Qualified[@ ::= @] |
| Parens[@ ::= "(" @ ")"] |
| PrimaryAgg[@ ::= @] |

; qualified : EmptyQual[@ ::= "<%S(Placeholder:qualified%S)>"]

77
| NameTicAgg[@ := @ "" @] |
| NameTicExpr[@ := @ "" ( @ ")"] |

allocator : newName[@ := "%S(Keyword:NEW%)" @] 
| NewQualified[@ := "%S(Keyword:NEW%)" @] |

subprog_decl : EmptySubpDecl[@ := "<%S(Placeholder:subprog decl%)>"] 
| SubpSpec[@ := @ @ "]"] |
| GenericSubpInst[@ := @ "%S(Keyword:IS%)" @ ",;" @] |
| AbstractSubpSpec[@ := @ "%S(Keyword:IS ABSTRACT%)";" @] |

psdl_met_opt : MetNull[@ ] 
| MetPrompt[@ := "%n<%S(Placeholder:psdl met%)>"] |
| MetUsec[@ := "%n --PSDL MAXIMUM EXECUTION TIME" @ " MICROSEC"] |
| MetMs[@ := "%n --PSDL MAXIMUM EXECUTION TIME" @ " MS"] |
| MetSec[@ := "%n --PSDL MAXIMUM EXECUTION TIME" @ " SEC"] |
| MetMin[@ := "%n --PSDL MAXIMUM EXECUTION TIME" @ " MIN"] |
| MetHrs[@ := "%n --PSDL MAXIMUM EXECUTION TIME" @ " HOURS"] |

subprog_spec : EmptySubpSpec[@ := "<%S(Placeholder:subprog spec%)>"] 
| SubpProc[@ := "%S(Keyword:PROCEDURE%)" @ @] |
| SubpFuncReturn[@ := "%S(Keyword:FUNCTION%)" @ @]. |
| SubpFunc[@ := "%S(Keyword:FUNCTION%)" @ @ "TRANSLATION_ERROR: Functions do not translate to PSDL"] |

designator : EmptyDesignator[@ := "<%S(Placeholder:designator%)>"] 
| DesigCompound[@ := @] |
| DesigString[@ := @] |

formal_part_opt : FormalPartOptNull[@ ] 
| FormalPartOptPrompt[@ := "<%S(Placeholder:formals%)>"] |
| FormalPart[@ := "(" @ ")"] |

param_s : ParamNil[@ ] 
| ParamList[@ := ^ @ " @ @ @] |

param : ParamId[@ := @ " @ @ @] |

mode : ModeNull[@ ] 
| ModePrompt[@ := "<%S(Placeholder:mode%)>"] |
InMode["%^"^"%S(Keyword:IN%S)""]
OutMode["%^"^"%S(Keyword:OUT%S)""]
InOutMode["%^"^"%S(Keyword:IN OUT%S)""]
AccessMode["%^"^"%S(Keyword:ACCESS%S)""]

pkg_decls : EmptyPkgDecl[@ ::= "%^"^"%S(Placeholder:pkdecl%S)>"]
   | PkgSpec[@ ::= @ @ "%^",n^"%S"]
   | GenPkgInst[@ ::= "%^"^"%S(Keyword:PACKAGE%S)" @ "%^%S(Keyword:IS%S)" @]

pkg_spec : Package[@ ::= "%^"^"%S(Keyword:PACKAGE%S)" @
"%^%S(Keyword:IS%S)"^"%n^"@ @ "%^"^"%b%n%S(Keyword:END%S)""compound_name]

private_part : PrivatePartNull[@ ]
   | PrivatePartPrompt[@ ::= "%^"^"%n%^"^"%S(Placeholder:private part%S)>"]
   | Private[@ ::= "%^"^"%n%S(Keyword:PRIVATE%S)"^"%n^"@

private_type : PrivateType[@ ::= @ @ "%^"^"%S(Keyword:PRIVATE%S)""]

limited_opt : LimitedOptNull[@ ]
   | LimitedOptPrompt[@ ::= "%^"^"%S(Placeholder:limited%S)>"]
   | Limited[@ ::= "%^"^"%S(Keyword:LIMITED%S)""]

use_clause : EmptyUse[@ ::= "%^"^"%S(Placeholder:use clause%S)>"]
   | Use[@ ::= "%^"^"%S(Keyword:USE%S)"@ "%^",n^"%n^"
   | UseType[@ ::= "%^"^"%S(Keyword:USE TYPE%S)"@ "%^",n^"

name_s : NameNil[@ ]
   | nameList[@ ::= ^"[", ]^"@]

rename_decl : EmptyRenameDecl[@ ::= "%^"^"%S(Placeholder:rename decl%S)>"]
   | RenameDeclSub[@ ::= @ @ "%^", @ @ "%^", @ @
   | RenameExc[@ ::= @ @ "%^"^"%S(Keyword:EXCEPTION%S)"@ "%^", @]
   | RenameUnitDecl[@ ::= @ @

rename_unit : EmptyRenameUnit[@ ::= "%^"^"%S(Placeholder:rename unit%S)>"]
   | RenamePkg[@ ::= @ @ "%^"^"%S(Keyword:PACKAGE%S)"@ "%^", @]
   | RenameSubprog[@ ::= @ @ "%^", @ @

renames : Renames[@ ::= "%^"^"%S(Keyword:RENAME%S)"@]
task_def {inh identifier idopt;};

  task_spec : EmptyTaskSpec[@ ::= "<%S(Placeholder:task spec%S)>"]
  [ SimpleTask[@ ::= "%S(Keyword:TASK%S)" @ @] {
    task_def.idopt = identifier;
  }
  TaskType[@ ::= "%S(Keyword:TASK TYPE%S)" @ @] {
    task_def.idopt = identifier;
  }

  task_def : TaskDefNull[@ :]
  [ TaskDefPrompt[@ ::= "<%S(Placeholder:task def%S)>"]
  TaskDef[@ ::= "%S(Keyword:IS%S)%t%n" @ @
    "%b%n%S(Keyword:END%S)"
    task_def.idopt]

  task_private_opt : TaskPvtOptNull[@ :]
  [ TaskPvtOptPrompt[@ ::= "<%S(Placeholder:task private%S)>"]
  TaskPvtOpt[@ ::= "%S(Keyword:PRIVATE%S)%t%n" @ @]

  prot_def {inh identifier idopt;};

  prot_spec : EmptyProtSpec[@ ::= "<%S(Placeholder:protected spec%S)>"]
  [ Prot[@ ::= "%S(Keyword:PROTECTED%S)" @ @] {
    prot_def.idopt = identifier;
  }
  ProtType[@ ::= "%S(Keyword:PROTECTED TYPE%S)%t%n" @ @
    "%b%n%S(Keyword:END%S)"
    prot_def.idopt]

  prot_private_opt : ProtPvtOptNull[@ :]
  [ ProtPvtOptPrompt[@ ::= "<%S(Placeholder:protected private%S)>"]
  ProtPvtOpt[@ ::= "%S(Keyword:PRIVATE%S)%t%n" @ @]

  prot_op_decl_s : ProtOptDeclListNil[@ :]
  [ ProtOptDeclList[@ ::= ^["%n"] @ @]

  prot_op_decl : EmptyProtOptDecl[@ ::= "<%S(Placeholder:prot op%S)>"]
  [ EntryDecl[@ ::= @]
prot\_elem\_decl\_s : ProtElemDeclNil[@ :]
  | ProtElemDeclList[@ ::= ^ ['%n'] @]

prot\_elem\_decl : EmptyProtElem[@ ::= "<\%(Placeholder:prot elem%)>" ]
  | ProtOptDecll[@ ::= @]
  | ProtElemCompDecl[@ ::= @]

entry\_decl\_s : EntryDeclPragma[@ ::= @]
  | EntryDeclPragmaList[@ ::= @ @ @]

entry\_decl : EmptyEntryDecl[@ ::= "<\%(Placeholder:entry decl%)>" ]
  | EntryDeclIdl[@ ::= "%S(Keyword:ENTRY%) " @ " ;%n"]
  | EntryRange[@ ::= "%S(Keyword:ENTRY%) " @ "(" @ ")" @ " ;%n"]

rep\_spec\_s : RepSpecNull[@ :]
  | RepSpecPromptl[@ ::= "<\%(Placeholder:representation specs%)>" ]
  | RepSpecList[@ ::= @ @ @]

comp\_unit : CompUnit[@ ::= @ @ "%n" @ "%n" @]

private\_opt : PrivateOptNull[@ :]
  | PrivateOptPromptl[@ ::= "<\%(Placeholder:private%)>" ]
  | PrivateOptl[@ ::= "%S(Keyword:PRIVATE%)"]

collection\_spec\_opt : CollectionSpecNull[@ :]
  | CollectionSpecPromptl[@ ::= "<\%(Placeholder:collection spec opt%)>%n"]
  | CollectionSpec[@ ::= @ @ "%n"]

collection\_spec : EmptyCollectionSpec[@ ::= "<\%(Placeholder:collection spec%)>" ]
  | CollectionWithUse[@ ::= @ @ @]
  | CollectionPragma[@ ::= @ @ @]

with\_clause : WithClause[@ ::= "%S(Keyword:WITH%) " @ ",%n"]

use\_clause\_opt : UseClauseOptNil[@ :]
  | UseClause[@ ::= ^ @]
body_stub := EmptyBodyStub[@ ::= "$S(Placeholder:body stub%)"]
| TaskStub[@ ::= "$S(Keyword:TASK BODY%)" @ "%S(Keyword:IS SEPARATE%)%n"]
| PkgStub[@ ::= "$S(Keyword:PACKAGE BODY%)" @ "%S(Keyword:IS SEPARATE%)%n"]
| SubprogStub[@ ::= "$S(Keyword:IS SEPARATE%)%n"]
| ProtStub[@ ::= "$S(Keyword:PROTECTED BODY%)" @ "%S(Keyword:IS SEPARATE%)%n"]

generic_hdr := GenericHdrNil[@ ]
| GenericHdrPrompt[@ ::= "$S(Placeholder:generic header%)%n"]
| GenericHdr[@ ::= "$S(Keyword:GENERIC%)%t%n"
| Generic_hdr[@ ::= "%S(Placeholder:generic header%)%n"]

generic_formal_part := GenericNil[@ ]
| GenFormalList[@ ::= ^ ["%n"] ]

generic_formal := EmptyGenFormal[@ ::= "$S(Placeholder:generic formal%)"]
| GenParm[@ ::= @ ""; %n--TRANSLATION_ERROR: Generic value parameters do not "" translate to PSDL"
| GenTypeParm[@ ::= "$S(Keyword:TYPE%)" @ @ "%S(Keyword:IS%)%n" @ @ "%S(Keyword:IS%)%n" @ @ "%S(Keyword:IS%)%n"
| GenProcParm[@ ::= "$S(Keyword:WITH PROCEDURE%)" @ @ "%S(Keyword:RETURN%)%n" @ @ "%S(Keyword:RETURN%)%n"
| GenFuncParm[@ ::= "$S(Keyword:WITH FUNCTION%)" @ @ "%S(Keyword:RETURN%)%n" @ @ "%S(Keyword:RETURN%)%n"
| GenPkmpParmBox[@ ::= "$S(Keyword:WITH PACKAGE%)" @ @ "%S(Keyword:IS NEW%)%n" @ @ "%S(Keyword:IS NEW%)%n"
| GenPkmpParm[@ ::= "$S(Keyword:WITH PACKAGE%)" @ @ "%S(Keyword:IS NEW%)%n" @ @ "%S(Keyword:IS NEW%)%n"
| GenUseparm[@ ::= @ "%n--TRANSLATION_ERROR: Generic Use clauses do not "" translate to PSDL"

generic_discrim_part_opt := GenDiscOptNull[@ ]
| GenDiscOptPrompt[@ ::= "$S(Placeholder:discriminant%)"
| GenDisc[@ ::= "( " @ ")"
| GenBox[^ := "(<>)"

subp_default := SubpDefaultNull[@ ]
| SubpDefaultPrompt[@ ::= "$S(Placeholder:default%)"]
generic_type_def : EmptyGenTypeDef[@ ::="<%S(Placeholder:generic type def%S)>"]
   [ GenTypeDefBox[@ :=="<" @]
   [ GenTypeDefRangeBox[@ :=="<%S(Placeholder:generic type delY/oS)>"]
   [ GenTypeDefModBox[@ :=="<%S(Placeholder:MOD%S)>"]
   [ GenTypeDefDeltaBox[@ :=="<%S(Placeholder:DELTA%S)>"]
   [ GenTypeDefDeltaDigBox[@ :=="<%S(Placeholder:DELTA%S)DIGITS%S)>"]
   [ GenTypeDefDigitsBox[@ :=="<%S(Placeholder:DIGITS%S)>"]
   [ GenTypeDefArray[@ ::="<"
   [ GenTypeDefAccess[@ :=="<"
   [ GenTypeDefPriv[@ :=="<"
   [ GenTypeDefDerived[@ :=="<"
   
   generic_derived_type : EmptyGenDerType[@ ::="<%S(Placeholder:generic derived type%S)>"]
   [ GenDerivedSubt[@ :=="<%S(Placeholder:NEW%S)"
   [ GenDerivedSubtPriv[@ :=="<%S(Placeholder:NEW%S)"
   [ GenDerivedAbst[@ :=="<%S(Placeholder:ABSTRACT NEW%S)"
   
   generic_inst : GenInst[@ ::="<%S(Placeholder:NEW%S)"
   
   rep_spec : EmptyRepSpec[@ ::="<%S(Placeholder:representation spec%S)>"]
   [ AttrbDef[@ :=="<%S(Placeholder:FOR%S)"
   [ RecordTypeSpec[@ :=="<%S(Placeholder:FOR%S)"
   [ AddressSpec[@ :=="<%S(Placeholder:FOR%S)"
   
   align_opt : AlignOptNull[@ :]
   [ AlignOptPrompt[@ :=="<%S(Placeholder:align%S)>"]
   [ AlignOpt[@ :=="<%S(Placeholder:AT MOD%S)"
   
   comp_loc_s : CompLocNull[@ :]
   [ CompLocPrompt[@ :=="<%S(Placeholder:locations%S)>"]
   [ CompLocList[@ :=="<%S(Placeholder:AT%S)"
   
83
view PSDL_VIEW;

identifier : IdNull[PSDL_VIEW ^::"<identifier>" ]
    | Ident[PSDL_VIEW ^::]

integer : IntNull[PSDL_VIEW ^::"<integer>" ]
    | Integer[PSDL_VIEW ^::]

compilation : CompilationNil[PSDL_VIEW @:: ]
    | Compilation[PSDL_VIEW @:: .. ^]

comp_unit_list : CUListNil[PSDL_VIEW @:: ]
    | CUList[PSDL_VIEW @:: := ^::"%n" ^::]

pragma : EmptyPragma[PSDL_VIEW ^::]
    |PragmaId[PSDL_VIEW ^:: ..]
    |PragmaSimple[PSDL_VIEW ^:: .. .. ]

pragma_arg_s : PragmaArgNil[PSDL_VIEW ^::]
    |PragmaSargList[PSDL_VIEW ^:: .. .. ]

pragma_arg : EmptyPragmaArg[PSDL_VIEW ^::]
    |PragmaExp[PSDL_VIEW ^:: ..]
    |PragmaNameExp[PSDL_VIEW ^:: .. .. ]
pragma_s : PragmasNil[PSDL_VIEW ^:]
    | PragmasList[PSDL_VIEW ^: ..]
    
decl : EmptyDecl[PSDL_VIEW ^:]
    | ObjDecl[PSDL_VIEW ^: .. ..]
    | NumDecl[PSDL_VIEW ^: ..]
    | TypeDecl[PSDL_VIEW ^: ..]
    | SubTypeDecl[PSDL_VIEW ^: ..]
    | SubProgDecl[PSDL_VIEW ^: ^"\%n"]
    | PkgDecl[PSDL_VIEW ^: ^"\%n"]
    | TaskDecl[PSDL_VIEW ^: ..]
    | ProtDecl[PSDL_VIEW ^: ..]
    | ExcDecl[PSDL_VIEW ^: ..]
    | RenameDecl[PSDL_VIEW ^: ..]
    | BodyStubDecl[PSDL_VIEW ^: ..]
    
def_id_s : DefIdNil[PSDL_VIEW ^:]
    | DefIdList[PSDL_VIEW ^: ^[",",] ^]
    
object_qualifier_opt : ObjQualOptNull[PSDL_VIEW ^:]
    | ObjQualOptPrompt[PSDL_VIEW ^:]
    | Aliased[PSDL_VIEW ^:]
    | Constant[PSDL_VIEW ^:]
    | AliasedConst[PSDL_VIEW ^:]
    
object_subtype_def : EmptySubtypeDef[PSDL_VIEW ^:]
    | SubtypeInd[PSDL_VIEW ^: ..]
    | ArrayType[PSDL_VIEW ^: ..]
    
init_opt : InitOptNull[PSDL_VIEW ^:]
    | InitOptPrompt[PSDL_VIEW ^:]
    | ExprInitOpt[PSDL_VIEW ^: ..]
    
discrim_part_opt : DiscrimPartNull[PSDL_VIEW ^:]
    | DiscrimPartPrompt[PSDL_VIEW ^:]
    | DiscrimPart[PSDL_VIEW ^: ..]
    | Box[PSDL_VIEW ^:]
    
type_completion : TypeComplNull[PSDL_VIEW ^:]
    | TypeComplPrompt[PSDL_VIEW ^:]
    | TypeDefCompl[PSDL_VIEW ^: ..]
"type def" : EmptyTypeDef[PSDL_VIEW ^ : ]
| EnumTypeDef[PSDL_VIEW ^ : ]
| IntTypeDef[PSDL_VIEW ^ : ]
| RealTypeDef[PSDL_VIEW ^ : ]
| ArrayTypeDef[PSDL_VIEW ^ : ]
| RecordTypeDef[PSDL_VIEW ^ : ]
| AccessTypeDef[PSDL_VIEW ^ : ]
| DerivedTypeDef[PSDL_VIEW ^ : ]
| PrivateTypeDef[PSDL_VIEW ^ : ]

"subtype ind" : EmptySubInd[PSDL_VIEW ^ : ]
| SubtypeIndConstraint[PSDL_VIEW ^ : ]
| SubtypeIndName[PSDL_VIEW ^ : ]

"constraint" : EmptyConstraint[PSDL_VIEW ^ : ]
| RangeConstraint[PSDL_VIEW ^ : ]
| DecDigConstraint[PSDL_VIEW ^ : ]

"derived type" : EmptyDerivedType[PSDL_VIEW ^ : ]
| NewDerivedType[PSDL_VIEW ^ : ]
| NewDerivedWithPrivate[PSDL_VIEW ^ : ]
| NewDerivedWithRecord[PSDL_VIEW ^ : ]
| AbsNewDerivedWithPrivate[PSDL_VIEW ^ : ]
| AbsNewDerivedWithRecord[PSDL_VIEW ^ : ]

"range constraint" : Range[PSDL_VIEW ^ : ]

"range" : EmptyRange[PSDL_VIEW ^ : ]
| SimpleRange[PSDL_VIEW ^ : ]
| NameTicRange[PSDL_VIEW ^ : ]
| NameTicRangeExp[PSDL_VIEW ^ : ]

"enum_id_s" : EnumIdNil[PSDL_VIEW ^ : ]
| EnumIdList[PSDL_VIEW ^ : ]

"enum_id" : EmptyEnumId[PSDL_VIEW ^ : ]
| Id[PSDL_VIEW ^ : ]
| CharLit[PSDL_VIEW ^ : ]

"integer_type" : EmptyIntType[PSDL_VIEW ^ : ]
| RangeSpec[PSDL_VIEW ^ : ]
| ModExpr[PSDL_VIEW ^ : ]

86
range_spec : RangeSpecConstr[PSDL_VIEW ^: ..]

range_spec_opt : RangeSpecOptNull[PSDL_VIEW ^: ]
  | RangeSpecOptPrompt[PSDL_VIEW ^: ]
  | RangeSpecOpt[PSDL_VIEW ^: ..]

real_type : EmptyRealType[PSDL_VIEW ^: ]
  | FloatType[PSDL_VIEW ^: ..]
  | FixedType[PSDL_VIEW ^: ..]

fixed_type : EmptyFixedType[PSDL_VIEW ^: ]
  | FixedDelta[PSDL_VIEW ^: ..]
  | FixedDeltaDigits[PSDL_VIEW ^: ..]

array_type : EmptyArrayType[PSDL_VIEW ^: ]
  | UnconstrArray[PSDL_VIEW ^: ..]
  | ConstrArray[PSDL_VIEW ^: ..]

component_subtype_def : Comp_subtypeDef[PSDL_VIEW ^: ..]

aliased_opt : AliasedOptNull[PSDL_VIEW ^: ]
  | AliasedOptPrompt[PSDL_VIEW ^: ]
  | AliasedOpt[PSDL_VIEW ^: ]

index_s : IndexNil[PSDL_VIEW ^: ]
  | IndexList[PSDL_VIEW ^: ..]

iter_discrete_range_s : DiscreteRangeNil[PSDL_VIEW ^: ]
  | DiscreteRangeList[PSDL_VIEW ^: ..]

discrete_range : EmptyDiscRng[PSDL_VIEW ^: ]
  | DiscRangeName[PSDL_VIEW ^: ..]
  | DiscRangeRange[PSDL_VIEW ^: ..]

range_constr_opt : EmptyRangeConstrOpt[PSDL_VIEW ^: ]
  | RangeConstr[PSDL_VIEW ^: ..]

record_def : EmptyRecordDef[PSDL_VIEW ^: ]
tagged_opt : TaggedOptNull[PSDL_VIEW ^:]
  | TaggedOptPrompt[PSDL_VIEW ^:]
  | Tagged[PSDL_VIEW ^:]
  | AbstractTagged[PSDL_VIEW ^:]

comp_list : EmptyCompList[PSDL_VIEW ^:]
  | CompListWithVariant[PSDL_VIEW ^:...
  | CompListWithPragma[PSDL_VIEW ^:...
  | NullWithPragma[PSDL_VIEW ^:...]

comp_decl : CompDecl[PSDL_VIEW ^:...]
  | CompDeclList[PSDL_VIEW ^:... ...]

variant_part_opt : EmptyVariantPart[PSDL_VIEW ^:]
  | VariantPartOptPragma[PSDL_VIEW ^:...]
  | VariantPartOpt[PSDL_VIEW ^:... ...]

comp_decl : CompDeclDefs[PSDL_VIEW ^:... ...]

discrim_spec_s : DiscrimSpecNil[PSDL_VIEW ^:...]
  | DiscrimSpecList[PSDL_VIEW ^:... ...]

discrim_spec : DiscrimSpecDef[PSDL_VIEW ^:... ...]

access_opt : AccessOptNull[PSDL_VIEW ^:]
  | AccessOptPrompt[PSDL_VIEW ^:]
  | AccessOpt[PSDL_VIEW ^:]

variant_part : VariantPart[PSDL_VIEW ^:... ...]

variant_s : VariantNil[PSDL_VIEW ^:]
  | VariantList[PSDL_VIEW ^: ... ...]

variant : VariantChoice[PSDL_VIEW ^: ... ...]
choice_s : ChoiceNil[PSDL_VIEW ^:]
  | ChoiceList[PSDL_VIEW ^: ...]
  ;

choice : EmptyChoice[PSDL_VIEW ^:]
  | ChoiceExpr[PSDL_VIEW ^: ...]
  | ChoiceRange[PSDL_VIEW ^: ...]
  | ChoiceOthers[PSDL_VIEW ^: ...]
  ;

discrete_with_range : DiscreteNameRange[PSDL_VIEW ^: ...]
  | DiscreteWithRange[PSDL_VIEW ^: ...]
  ;

access_type : EmptyAccessType[PSDL_VIEW ^:]
  | AccessSubtype[PSDL_VIEW ^: ...]
  | AccessConstSubtype[PSDL_VIEW ^: ...]
  | AccessAllSubtype[PSDL_VIEW ^: ...]
  | AccessProcedure[PSDL_VIEW ^: ...]
  | AccessFunction[PSDL_VIEW ^: ...]
  ;

prot_opt : ProtOptNull[PSDL_VIEW ^:]
  | ProtOptPrompt[PSDL_VIEW ^:]
  | Protected[PSDL_VIEW ^:]
  ;

decl_item_s : DeclListNil[PSDL_VIEW ^:]
  /* [DeclList[PSDL_VIEW ^: ^: ["%n"] ^: */
  | DeclList[PSDL_VIEW ^: ^: ["%n"] ^: ]*
  ;

decl_item : EmptyDeclItem[PSDL_VIEW ^:]
  | Decl[PSDL_VIEW ^: ^:]
  | UseClauseDecl[PSDL_VIEW ^: ...]
  | DeclRepSpec[PSDL_VIEW ^: ...]
  | DeclPragma[PSDL_VIEW ^: ...]
  ;

name : EmptyName[PSDL_VIEW ^:]
  | SimpleName[PSDL_VIEW ^: ...]
  | IndexComp[PSDL_VIEW ^: ...]
  | SelectedComp[PSDL_VIEW ^: ...]
  | Attribute[PSDL_VIEW ^: ...]
  | OperatorSymbol[PSDL_VIEW ^: ...]
  ;

mark : EmptyMark[PSDL_VIEW ^:]
  | Mark[PSDL_VIEW ^: ^:]
  ;
ticdot : TicDotNil[PSDL_VIEW ^ : ]
  | TicOpt[PSDL_VIEW ^ := "\"\" ^]
  | DotOpt[PSDL_VIEW ^ := "." ^]

marklist : MarkListNil[PSDL_VIEW ^ : ]
  | MarkList[PSDL_VIEW ^ : ^]

compound_name : EmptyCompound[PSDL_VIEW ^ : ]
  | DotCompound[PSDL_VIEW ^ : ^ "\"\" ^]

name_list : CompoundNameNil[PSDL_VIEW ^ : ]
  | CompoundList[PSDL_VIEW ^ : ...]

value_s : ValueNil[PSDL_VIEW ^ : ]
  | ValueList[PSDL_VIEW ^ : ...]

value : EmptyValue[PSDL_VIEW ^ : ]
  | ValueExpr[PSDL_VIEW ^ : ...]
  | ValueCompAssoc[PSDL_VIEW ^ : ...]
  | ValueDiscWithRange[PSDL_VIEW ^ : ...]

selected_comp : EmptySelComp[PSDL_VIEW ^ : ]
  | DotId[PSDL_VIEW ^ : ...]
  | DotUsedChar[PSDL_VIEW ^ : ...]
  | DotString[PSDL_VIEW ^ : ...]
  | DotAll[PSDL_VIEW ^ : ...]

attribute_id : EmptyAttribId[PSDL_VIEW ^ : ]
  | AttribId[PSDL_VIEW ^ : ...]
  | AttribDigits[PSDL_VIEW ^ : ...]
  | AttribDelta[PSDL_VIEW ^ : ...]
  | AttribAccess[PSDL_VIEW ^ : ...]

numeric_lit : IntLit[PSDL_VIEW ^ : ^]
  | RealLit[PSDL_VIEW ^ : ^]

literal : EmptyLiteral[PSDL_VIEW ^ : ]
  | NumLit[PSDL_VIEW ^ : ...]
  | UsedChar[PSDL_VIEW ^ : ...]
  | NilLit[PSDL_VIEW ^ : ]
aggregate : EmptyAggregate[PSDL_VIEW ^ : ]
    | AggCompAssoc[PSDL_VIEW ^ : ..]
    | AggValues2[PSDL_VIEW ^ : ..]
    | AggExprValue[PSDL_VIEW ^ : ..]
    | AggExprWithNull[PSDL_VIEW ^ : ..]
    | AggExprNullRec[PSDL_VIEW ^ : ]

value_s_2 : ValueS2Pair[PSDL_VIEW ^ : ..]
    | ValueS2List[PSDL_VIEW ^ : ..]

comp_assoc : CompAssoc[PSDL_VIEW ^ : ..]

expression : EmptyExpression[PSDL_VIEW ^ : ]
    | Relation[PSDL_VIEW ^ : ..]
    | And[PSDL_VIEW ^ : ..]
    | Or[PSDL_VIEW ^ : ..]
    | Xor[PSDL_VIEW ^ : ..]
    | AndThen[PSDL_VIEW ^ : ..]
    | OrElse[ ^ : ..]

relation : EmptyRelation[PSDL_VIEW ^ : ]
    | SimpleExpr[PSDL_VIEW ^ : ..]
    | Equal[PSDL_VIEW ^ : ..]
    | NotEqual[PSDL_VIEW ^ : ..]
    | LessThan[PSDL_VIEW ^ : ..]
    | LessThanEq[PSDL_VIEW ^ : ..]
    | GreaterThan[PSDL_VIEW ^ : ..]
    | GreaterThanEq[PSDL_VIEW ^ : ..]
    | RangeMember[PSDL_VIEW ^ : ..]
    | NameMember[PSDL_VIEW ^ : ..]

membership : EmptyMembr[PSDL_VIEW ^ : ]
    | In[PSDL_VIEW ^ : ]
    | NotIn[PSDL_VIEW ^ : ]

simple_expression : EmptySimple[PSDL_VIEW ^ : ]
    | Term[PSDL_VIEW ^ : ..]
    | Addition[PSDL_VIEW ^ : ..]
    | Subtraction[PSDL_VIEW ^ : ..]
    | Concat[PSDL_VIEW ^ : ..]

unary : UnaryNull[PSDL_VIEW ^ : ]
    | UnaryPrompt[PSDL_VIEW ^ : ]
    | Plus[PSDL_VIEW ^ : ]
term : EmptyTerm[PSDL_VIEW ^ :]
| Factor[PSDL_VIEW ^ : ..]
| Mul[PSDL_VIEW ^ : ...]
| Div[PSDL_VIEW ^ : ....]
| Mod[PSDL_VIEW ^ : .....]
| Rem[PSDL_VIEW ^ : ...]

factor : EmptyFactor[PSDL_VIEW ^ :]
| Primary[PSDL_VIEW ^ : ..]
| NotPrimary[PSDL_VIEW ^ : ..]
| AbsPrimary[PSDL_VIEW ^ : ..]
| Expon[PSDL_VIEW ^ : ..]

primary : EmptyPrimary[PSDL_VIEW ^ :]
| Literal[PSDL_VIEW ^ : ..]
| PrimaryName[PSDL_VIEW ^ : ..]
| Allocator[PSDL_VIEW ^ : ..]
| Qualified[PSDL_VIEW ^ : ..]
| Parens[PSDL_VIEW ^ : ..]

qualified : EmptyQual[PSDL_VIEW ^ :]
| NameTicAgg[PSDL_VIEW ^ : ..]
| NameTicExpr[PSDL_VIEW ^ : ..]

allocator : newName[PSDL_VIEW ^ :]
| NewQualified[PSDL_VIEW ^ : ..]

subprog_decl : EmptySubpDecl[PSDL_VIEW ^ :]
| SubprogSpec[PSDL_VIEW ^ : ^ ..]
| GenericSubprogInst[PSDL_VIEW ^ : ......]
| AbstractSubprogSpec[PSDL_VIEW ^ : .... ..]

psdl_met_opt : MetNull[PSDL_VIEW ^ :]
| MetPrompt[PSDL_VIEW ^ :]
| MetUseq[PSDL_VIEW ^ : ..]
| MetMs[PSDL_VIEW ^ : ...]
| MetSec[PSDL_VIEW ^ : ..]
| MetMin[PSDL_VIEW ^ : ..]
| MetHrs[PSDL_VIEW ^ : ..]

subprog_spec : EmptySubpSpec[PSDL_VIEW ^ : ]
designator : EmptyDesignator[PSDL_VIEW ^ : 
] | DesignCompound[PSDL_VIEW ^ : ]
| DesignString[PSDL_VIEW ^ : ]
;

formal_part_opt : FormalPartOptNull[PSDL_VIEW ^ : ]
| FormalPartOptPrompt[PSDL_VIEW ^ : ]
| FormalPart[PSDL_VIEW ^ : ]
;

param_s : ParamNil[PSDL_VIEW ^ : ]
| ParamList[PSDL_VIEW ^ : ["", "] ^ ]
;

param : ParamId[PSDL_VIEW ^ : ["", "] ^ ]
;

mode : ModeNull[PSDL_VIEW ^ : ]
| ModePrompt[PSDL_VIEW ^ : ]
| InMode[PSDL_VIEW ^ : ]
| OutMode[PSDL_VIEW ^ : ]
| InOutMode[PSDL_VIEW ^ : ]
| AccessMode[PSDL_VIEW ^ : ]
;

pkg_decl : EmptyPkgDecl[PSDL_VIEW ^ : ]
| PkgSpec[PSDL_VIEW ^ : ...
| GenPkgInst[PSDL_VIEW ^ : ...
;

pkg_spec : Package[PSDL_VIEW ^ : ...
;

private_part : PrivatePartNull[PSDL_VIEW ^ : ]
| PrivatePartPrompt[PSDL_VIEW ^ : ]
| Private[PSDLerview ^ : ..]
;

private_type : PrivateType[PSDL_VIEW ^ : ..]
;

limited_opt : LimitedOptNull[PSDL_VIEW ^ : ]
| LimitedOptPrompt[PSDL_VIEW ^ : ]
| Limited[PSDL_VIEW ^ : ]
use_clause : EmptyUseC[PSDL_VIEW ^:]
    | Use[PSDL_VIEW ^: ..]
    | UseType[PSDL_VIEW ^: ..]
    
name_s : NameNil[PSDL_VIEW ^:]
    | nameList[PSDL_VIEW ^: ...]
    
rename_decl : EmptyRenameDecl[PSDL_VIEW ^:]
    | RenameDeclSub[PSDL_VIEW ^: .. .. ..]
    | RenameExc[PSDL_VIEW ^: .. ..]
    | RenameUnitDecl[PSDL_VIEW ^: ^]
    
rename_unit : EmptyRenameUnit[PSDL_VIEW ^:]
    | RenamePkg[PSDL_VIEW ^: .. .. ..]
    | RenameSubprog[PSDL_VIEW ^: ^ ^ ..]
    
renames : Renames[PSDL_VIEW ^: ..]
    
task_spec : EmptyTaskSpec[PSDL_VIEW ^:]
    | SimpleTask[PSDL_VIEW ^: .. ..]
    | TaskType[PSDL_VIEW ^: .. ..]
    
task_def : TaskDefNull[PSDL_VIEW ^:]
    | TaskDefPrompt[PSDL_VIEW ^: .. ..]
    | TaskDef[PSDL_VIEW ^: .. .. ..]
    
task_private_opt : TaskPvtOptNull[PSDL_VIEW ^:]
    | TaskPvtOptPrompt[PSDL_VIEW ^: .. ..]
    | TaskPvtOpt[PSDL_VIEW ^: .. .. ..]
    
prot_spec : EmptyProtSpec[PSDL_VIEW ^:]
    | Prot[PSDL_VIEW ^: .. ..]
    | ProtType[PSDL_VIEW ^: .. .. ..]
    
prot_def : ProtDef[PSDL_VIEW ^: .. ..]
    
prot_private_opt : ProtPvtOptNull[PSDL_VIEW ^:]
    | ProtPvtOptPrompt[PSDL_VIEW ^: .. ..]
    | ProtPvtOpt[PSDL_VIEW ^: .. .. ..]
    
94
prot_op_decl_s : ProtOptDeclListNil[PSDL_VIEW ^:]
    [ProtOptDeclList[PSDL_VIEW ^: ...]]
    ;

prot_op_decl : EmptyProtOpDecl[PSDL_VIEW ^:]
    [EntryDecl[PSDL_VIEW ^: ...]]
    [ProtOptSubprog[PSDL_VIEW ^: ...]]
    [RepSpec[PSDL_VIEW ^: ...]]
    [ProtOptPragma[PSDL_VIEW ^: ...]]
    ;

prot_elem_decl_s : ProtElemDeclNil[PSDL_VIEW ^:]
    [ProtElemDeclList[PSDL_VIEW ^: ...]]
    ;

prot_elem_decl : EmptyProtElem[PSDL_VIEW ^:]
    [ProtOptDecl[PSDL_VIEW ^: ...]]
    [ProtElemCompDecl[PSDL_VIEW ^: ...]]
    ;

entry_decl_s : EntryDeclPragma[PSDL_VIEW ^: ...]
    [EntryDeclPragmaList[PSDL_VIEW ^: ...]]
    ;

entry_decl : EmptyEntryDecl[PSDL_VIEW ^:]
    [EntryDeclId[PSDL_VIEW ^: ...]]
    [EntryRange[PSDL_VIEW ^: ...]]
    ;

rep_spec_s : RepSpecNull[PSDL_VIEW ^:]
    [RepSpecPrompt[PSDL_VIEW ^:]]
    [RepSpecList[PSDL_VIEW ^: ...]]
    ;

comp_unit : CompUnit[PSDL_VIEW ^: ...^:]
    ;

private_opt : PrivateOptNull[PSDL_VIEW ^:]
    [PrivateOptPrompt[PSDL_VIEW ^:]]
    [PrivateOpt[PSDL_VIEW ^:]]
    ;

context_spec_opt : ContextSpecNull[PSDL_VIEW ^:]
    [ContextSpecPrompt[PSDL_VIEW ^:]]
    [ContextSpec[PSDL_VIEW ^: ...]]
    ;

context_spec : EmptyContextSpec[PSDL_VIEW ^:]
    [ContextWithUse[PSDL_VIEW ^: ...]]
    [ContextPragma[PSDL_VIEW ^: ...]]
    95
with_clause : WithClause[PSDL_VIEW ^: ..]
;
use_clause_opt : UseClauseOptNil[PSDL_VIEW ^:]
    | UseClause[PSDL_VIEW ^: ...]
;
body_stub : EmptyBodyStub[PSDL_VIEW ^:]
    | TaskStub[PSDL_VIEW ^: ..]
    | PkgStub[PSDL_VIEW ^: ..]
    | SubprogStub[PSDL_VIEW ^: ..]
    | ProtStub[PSDL_VIEW ^: ..]
;
generic_hdr : GenericHdrNil[PSDL_VIEW ^:]
    | GenericHdrPrompt[PSDL_VIEW ^:]
    | GenericHdr[PSDL_VIEW ^: ..]
;
generic_formal_part : GenericNil[PSDL_VIEW ^: “%b”]
    | GenFormalList[PSDL_VIEW ^: “[“%n”]” “%n”]
;
generic_formal : EmptyGenFormal[PSDL_VIEW ^:]
    | GenParm[PSDL_VIEW ^: ... “: GenericValue”]
    | GenTypeParm[PSDL_VIEW ^: ... “: GenericType”]
    | GenProcParm[PSDL_VIEW ^: ... “: GenericProcedure”]
    | GenFuncParm[PSDL_VIEW ^: ... “: GenericFunction”]
    | GenPkgParmBox[PSDL_VIEW ^: ...]
    | GenPkgParm[PSDL_VIEW ^: ...]
    | GenUseParm[PSDL_VIEW ^: ...]
;
generic_discrim_part_opt : GenDiscOptNull[PSDL_VIEW ^:]
    | GenDiscOptPrompt[PSDL_VIEW ^:]
    | GenDisc[PSDL_VIEW ^: ...]
    | GenBox[PSDL_VIEW ^:]
;
subp_default : SubpDefaultNull[PSDL_VIEW ^:]
    | SubpDefaultPrompt[PSDL_VIEW ^:]
    | SubpDefName[PSDL_VIEW ^: ...]
    | SubpDefBox[PSDL_VIEW ^:]
;
generic_type_def : EmptyGenTypeDef[PSDL_VIEW ^:]
    | GenTypeBox[PSDL_VIEW ^:]
    | GenTypeRangeBox[PSDL_VIEW ^:]
    | GenTypeModBox[PSDL_VIEW ^:]

96
generic_derived_type : EmptyGenDerType[PSDL_VIEW^:]
| GenDerivedSubt[PSDL_VIEW^:..]
| GenDerivedSubtPriv[PSDL_VIEW^:..]
| GenDerivedAbst[PSDL_VIEW^:..]

generic_inst : GenInst[PSDL_VIEW^:..]

rep_spec : EmptyRepSpec[PSDL_VIEW^:]
| AttribDef[PSDL_VIEW^:..]
| RecordTypeSpec[PSDL_VIEW^:..]
| AddressSpec[PSDL_VIEW^:..]

align_opt : AlignOptNull[PSDL_VIEW^:]
| AlignOptPrompt[PSDL_VIEW^:]
| AlignOpt[PSDL_VIEW^:..]

comp_loc_s : CompLocNull[PSDL_VIEW^:]
| CompLocPrompt[PSDL_VIEW^:]
| CompLocList[PSDL_VIEW^:..]

/* **********************************************/
/* **********************************************/
/* **********************************************/

/* **********************************************/
/*           unparse.psl.ssl                      */
/* Date:    3 March, 1995                        */
/* Author:  Chris Eagle                          */
/* System:  Sun SPARCstation                     */
/* Description: This file contains the unparsing rules for PSDL */
/* productions.                                  */
/* **********************************************/

psdl : EmptyPsdl[PSDL_VIEW^:]
| PsdlPH[PSDL_VIEW^:]
| Component[PSDL_VIEW^:..]
component_s : ComponentNil[PSDL_VIEW ^ : ]
    | ComponentList[PSDL_VIEW ^ : ["%n%n"] ^ ]
    ;
component : CompDataType[PSDL_VIEW ^ : ^ ]
    | CompOperator[PSDL_VIEW ^ : ^ ]
    ;
data_type : DataType[PSDL_VIEW ^ : "%S(Keyword:TYPE%S)" ^ "%n" ^ ]
    ;
type_spec : TypeSpec[PSDL_VIEW ^ : "%S(Keyword:SPECIFICATION%S)%t%n"
    ^^^ "%S(Keyword:%b%nEND%S)%n"]
    ;
generic_type_decl : GTDNil[PSDL_VIEW ^ : ]
    | GTD_PH[PSDL_VIEW ^ : ]
    | GTD[PSDL_VIEW ^ : "%S(Keyword:GENERIC%S)%t%n" ^ "%~"]
    ;
type_decl_opt : TDO.Nil[PSDL_VIEW ^ : ]
    | TDO_PH[PSDL_VIEW ^ : ]
    | TDO[PSDL_VIEW ^ : "%n"]
    ;
op_list_opt : OLO.Nil[PSDL_VIEW ^ : ]
    | OLO_Cons[PSDL_VIEW ^ : [%"n"] ^ ]
    ;
operator : EmptyOperator[PSDL_VIEW ^ : ]
    | PslOp[PSDL_VIEW ^ : "%S(Keyword:OPERATOR%S)" ^ "%n"]
    ;
operator_imp : OperatorImp[PSDL_VIEW ^ : ^ "%n"]
    ;
operator_spec : OpSpec[PSDL_VIEW ^ : "%S(Keyword:SPECIFICATION%S)%t%n"
    ^ ^ "%S(Keyword:%b%nEND%S)%n"]
    ;
interface_s : Interface_s.Nil[PSDL_VIEW ^ : ]
    | InterfaceList[PSDL_VIEW ^ : ^ ]
    ;
interface : EmptyInterface[PSDL_VIEW ^ : ]
    | Interface[PSDL_VIEW ^ : "%n"]
    ;
attribute : Generics[PSDL_VIEW ^ : "%S(Keyword:GENERIC%S)%t%n" ^ "%b"]
    | Inputs[PSDL_VIEW ^ : "%S(Keyword:INPUT%S)%t%n" ^ "%b"]
    ;
type decl_s : TypeDeclNil[PSDL_VIEW @:] 
    | TypeDeclList[PSDL_VIEW @::= ^[",%n"] @]

    type decl_s2 : TypeDeclNil2[PSDL_VIEW @:]
    | TypeDeclList2[PSDL_VIEW @::= ^[", "] @]

type decl : TypeDeclPSDL[PSDL_VIEW @::= @ @]
    | EmptyTypeDecl[PSDL_VIEW @:]

    type name : EmptyTypeName[PSDL_VIEW @:]
    | TN_Id[PSDL_VIEW ^: ^]
    | TN_Array[PSDL_VIEW ^: ^[" "] ]

reqmts_trace : RqmtsNil[PSDL_VIEW ^:]
    | Rqmts_PH[PSDL_VIEW ^: ]
    /* | Rqmts[PSDL_VIEW ^: "%S(Keyword:BY REQUIREMENTS%S) " ]*/

    functionality : FuncNil[PSDL_VIEW ^:]
    | FuncPH[PSDL_VIEW ^:]
    /* | Functionality[PSDL_VIEW ^: .. .. ]*/

    /*
    keywords : KW_Nil[PSDL_VIEW ^: ]
    | KW_PH[PSDL_VIEW ^: ]
    | Keywords[PSDL_VIEW ^: "%S(Keyword:KEYWORDS%S) " ]

    informal_desc : ID_Nil[PSDL_VIEW ^: ]
    | ID_PH[PSDL_VIEW ^: ]
    | InfDesc[PSDL_VIEW ^: "%S(Keyword:DESCRIPTION%S)%t%n" "%b%n]

    formal_desc : FD_Nil[PSDL_VIEW ^: ]
    | FD_PH[PSDL_VIEW ^: ]
    | FormalDesc[PSDL_VIEW ^: "%S(Keyword:AXIOMS%S)%t%n" "%b%n"]

    */
type_impl : AdaTypelmpl[PSDL_VIEW ^ :
    "%S(Keyword:IMPLEMENTATION ADA%S)"
    ^ "%S(Keyword:%nEND%S)%n"]
/* | TypeImpl[PSDL_VIEW ^ :
    "%S(Keyword:IMPLEMENTATION%S)"
    ..
    .. "%S(Keyword:%nEND%S)%n"]*/

operator_impl : AdaOplmp[PSDL_VIEW ^ :
    "%S(Keyword:IMPLEMENTATION ADA%S)"
    ^ "%S(Keyword:%nEND%S)%n"]
    | EmptyImpl[PSDL_VIEW ^ ]
/* | PslOplmp[PSDL_VIEW ^ :
    "%S(Keyword:IMPLEMENTATION%S)"
    .. "%S(Keyword:%nEND%S)%n"]*/

initial_expression_list : InitExpNil[PSDL_VIEW ^ :]
    | InitExpList[PSDL_VIEW ^ : [",", ] ^ ]

initial_expression : ExpTrue[PSDL_VIEW ^ : "True"]
    | ExpFalse[PSDL_VIEW ^ : "False"]
    | ExpInt[PSDL_VIEW ^ : ]
    | ExpReal[PSDL_VIEW ^ : ]
    | ExpStr[PSDL_VIEW ^ : ]
    | Expld[PSDL_VIEW ^ : ]
    | ExpType[PSDL_VIEW ^ : "^^"]
    | ExpInitExp[PSDL_VIEW ^ : "(" ^ ")"]
    | ExpBinOp[PSDL_VIEW ^ : "^^"]
    | ExpUnary[PSDL_VIEW ^ : ]

opt_init_exp_list : optListNil[PSDL_VIEW ^ :]
    | optListPrompt[PSDL_VIEW ^ :]
    | optList[PSDL_VIEW ^ : "(" ^ ")"]

binary_op : PslAnd[PSDL_VIEW ^ : "%S(Keyword: AND%S)"
    | PslOr[PSDL_VIEW ^ : "%S(Keyword: OR%S)"
    | PslXor[PSDL_VIEW ^ : "%S(Keyword: XOR%S)"
    | PslLT[PSDL_VIEW ^ : "<"]
    | PslGT[PSDL_VIEW ^ : ">"]
    | PslEQ[PSDL_VIEW ^ : "="]
    | PslGTEQ[PSDL_VIEW ^ : ">="]
    | PslLT[PSDL_VIEW ^ : "<"]
    | PslNE[PSDL_VIEW ^ : "/="]
    | PslAdd[PSDL_VIEW ^ : "+"]
    | PslSub[PSDL_VIEW ^ : "-"]
    | PslCat[PSDL_VIEW ^ : "&"]
    | PslMul[PSDL_VIEW ^ : "*"]
    | PslDiv[PSDL_VIEW ^ : "/"]
    | PslMod[PSDL_VIEW ^ : "%S(Keyword: MOD%S)"]
unary_op : PSDL_Not

<table>
<thead>
<tr>
<th>unary_op</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSDL_Not</td>
<td>NOT</td>
</tr>
<tr>
<td>PSDL_Abs</td>
<td>ABS</td>
</tr>
<tr>
<td>PSDL_Pos</td>
<td>+</td>
</tr>
</tbody>
</table>

time_unit : TimeSec

<table>
<thead>
<tr>
<th>time_unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeSec</td>
<td>MS</td>
</tr>
<tr>
<td>TimeMin</td>
<td>MIN</td>
</tr>
<tr>
<td>TimeHrs</td>
<td>HOURS</td>
</tr>
</tbody>
</table>

```plaintext
[PSDL Rem[PSDL_VIEW ^ : "%S(Keyword: REM%S)""]
[PSDL Exp[PSDL_VIEW ^ : " ** "]

unary_op : PSDL_Not[PSDL_VIEW ^ : "%S(Keyword: NOT%S)""]
[PSDL_Abs[PSDL_VIEW ^ : "%S(Keyword: ABS%S)""]
[PSDL_Pos[PSDL_VIEW ^ : "+"]

time_unit : TimeSec[PSDL_VIEW ^ : "%S(Keyword: MICROSEC%S)""]
[TimeMs[PSDL_VIEW ^ : "%S(Keyword: MS%S)""]
[TimeSec[PSDL_VIEW ^ : "%S(Keyword: SEC%S)""]
[TimeMin[PSDL_VIEW ^ : "%S(Keyword: MIN%S)""]
[TimeHrs[PSDL_VIEW ^ : "%S(Keyword: HOURS%S)""]
```
APPENDIX C. SSL SOURCE CODE: ATTRIBUTE FUNCTIONS

The source code below is used to compute the attributes of Ada 95 productions. These attributes are specified as productions within the PSDL language. Once computed, each attribute is displayed according to the unparsing rules for PSDL productions.

```plaintext
subprogspec {
    inh psdl_met_opt met;
};

INT IsProcSpec(subprogspec ss) {
    with (ss) (  
        SubProgProc(*,*) : 1,
        default : 0
    );
}

INT IsProcDecl(decl d) {
    with (d) (  
        SubProgDecl(sd) : with (sd) (  
            SubprogSpec(*,ss,*) : IsProcSpec(ss),
            default : 0
        ),
    );
}
```

/* File: attrib.ada9x.ssl */
/* Date: 3 March, 1995 */
/* Author: Chris Eagle */
/* System: Sun SPARCstation */
/* Description: This file contains functions which compute the */
/* attribute for Ada 9x productions. These attributes are */
/* in turn PSDL productions, which when displayed provide */
/* the Ada 9x to PSDL translation. */
/* */
/* function to count the number of declarations in the input */
/* decl_item_s list which will translate to PSDL operators */
/* Return value = number of decls that will become PSDL operators */

INT CountOps(decl_item_s dis) {
    with (dis) {
        DeclListNil : 0,
        DeclList(di, rest) : with (di) {
            Decl(d) : with (d) {
                SubProgDecl(*) : IsProcDecl(d) + CountOps(rest),
                PkgDecl(pd) : with (pd) {
                    PkgSpec(gh, ps) : with (ps) {
                        Package(cn, ds, pp) : (CountOps(ds) == 1 ? 1 : 0) + 
                        CountOps(rest)
                    },
                    default : CountOps(rest)
                },
                RenameDecl(rd) : with (rd) {
                    RenameUnitDecl(ru) : with (ru) {
                        RenameSubprog(*, ss, *) : IsProcSpec(ss) + CountOps(rest),
                        default : CountOps(rest)
                    },
                    default : CountOps(rest)
                },
                default : CountOps(rest)
            }
        }
    }
};

identifier MarkTold(mark m) {
    with (m) {
        EmptyMark : IdNull,
        Mark(i, *) : i
    }
};

type_name PSDLTypeName(mark m) {
    with (m) {
        EmptyMark : EmptyTypeName,
        Mark(i, *) : TN_Id(i)
    }
};

type_decl PSDLTypeDecl(param p) {
with (p) {
    EmptyParam : EmptyTypeDecl,
    ParamId(d, *, m, *) : TypeDeclPSDL(dis, PSDLTypeName(m))
}

/* function to return a parameter list containing only the parameters */
/* in p which are of type IN or IN OUT */

type_decl_s ExtractIns(param_s p) {
    with (p) {
        ParamNil : TypeDeclNil,
        ParamList(parm, ps) :
            with (parm) {
                ParamId(d, md, mk, *) :
                    with (md) {
                        OutMode : ExtractIns(ps),
                        AccessMode : ExtractIns(ps),
                        default : PSDLTypeDecl(parm)::ExtractIns(ps)
                    },
                    EmptyParam : ExtractIns(ps)
            }
    }
}

/* function to create the INPUTS portion of a PSDL operator */
/* specification given an input formal parameter list */

interface MakeInputs(formal_part_opt fp) {
    with (fp) {
        FormalPart(p) : with (ExtractIns(p)) {
            TypeDeclNil : EmptyInterface,
            TypeDeclList(*, *) : Interface(Inputs(ExtractIns(p)), RqmtsNil)
        },
        default : EmptyInterface,
    }
}

/* function to return a parameter list containing only the parameters */
/* in p which are of type OUT or IN OUT */

type_decl_s ExtractOuts(param_s p) {
    with (p) {
        ParamNil : TypeDeclNil,
        ParamList(parm, ps) :
            with (parm) {
                ParamId(d, md, mk, io) :
                    with (md) {
                        OutMode : PSDLTypeDecl(parm)::ExtractOuts(ps),
                    }
            }
    }
}

InOutMode : PSDL.TypeDecl(parm):.ExtractOuts(ps),
default : ExtractOuts(ps)
),
EmptyParam : ExtractOuts(ps)
)
)
);

/* function to create the OUTPUTS portion of a PSDL operator */
/* specification given an input formal parameter list */

interface MakeOutputs(formal_part_opt fp) {
    with (fp) {
        FormalPart(p) : with (ExtractOuts(p)) {
            TypeDeclNil : EmptyInterface,
            TypeDeclList(,* , *) : Interface(Outputs(ExtractOuts(p)),RqmtsNil)
        ),
        default : EmptyInterface,
    }
}

/* function to create the MET portion of a PSDL operator given */
/* an input psdl_met_opt from an Ada program */

interface MakeMet(psdl_met_opt pmo) {
    with (pmo) {
        MetUsec(i) : Interface(MET(Timeusec(i)),RqmtsNil),
        MetMs(i) : Interface(MET(TimeMs(i)),RqmtsNil),
        MetSec(i) : Interface(MET(TimeSec(i)),RqmtsNil),
        MetMin(i) : Interface(MET(TimeMin(i)),RqmtsNil),
        MetHrs(i) : Interface(MET(TimeHrs(i)),RqmtsNil),
        default : EmptyInterface
    }
}

identifier ModeTold(mode m) {
    with (m) {
        OutMode : Ident("out"),
        InOutMode : Ident("in_out"),
        AccessMode : Ident("access"),
        default : Ident("in")
    }
}

type_decl PSDL.ProcParm(param p) {
    with (p) {
        ParamId(dis, md, mk, *) : TypeDeclPSDL(dis,TN_Array(ModeTold(md)), TypeDeclPSDL(Ident("t")::DefIdNil,
type decl_s2 ProcParmsToTypeDeclS(param_s p) {
    with (p) {
        ParamNil : TypeDeclNil2,
        ParamList(parm, ps) :
            with (parm) {
                ParamId(d, md, mk, *) :
                    PSDLProcParm(parm): ProcParmsToTypeDeclS(ps),
                EmptyParam : ProcParmsToTypeDeclS(ps)
            }
    }
}

type name ProcTypeName(formal_part_opt fpo) {
    with (fpo) {
        FormalPart(p):
            TN_Array(Ident("PROCEDURE"),ProcParmsToTypeDeclS(p)),
        default : TN_Id(Ident("PROCEDURE"))
    }
}

identifier NameToIdent(name n) {
    with (n) {
        EmptyName : IdNull,
        SimpleName(i) : i,
        OperatorSymbol(os) : Ident(os),
        default : Ident("DefaultId")
    }
}

type decl_s2 ReturnDecl(name n) {
    TypeDeclPSDL(Ident("RETURN"): :DefIdNil,TN_Id(NameToIdent(n))): :TypeDeclNil2
}

type decl_s2 FuncParmsToTypeDeclS(param_s p) {
    with (p) {
        ParamNil : TypeDeclNil2,
        ParamList(parm, ps) :
            with (parm) {
                ParamId(d, md, mk, io) :
                    PSDL TypeDecl(parm): FuncParmsToTypeDeclS(ps),
                EmptyParam : FuncParmsToTypeDeclS(ps)
            }
    }
}

type name FuncTypeName(formal_part_opt fpo, name n) {

107
with (fpo) {
    FormalPart(p) : 
    TN_Array(Ident("FUNCTION"), FuncParmsToTypeDecl(p)@ReturnDecl(n)),
    default : TN_Array(Ident("FUNCTION"), ReturnDecl(n))
}

identifier CompoundTold(compound_name cn) {
    Ident(CompoundToStr(cn, "_"))
}

identifier DesigTold(designator d) {
    with (d) {
        DesigCompound(cn) : CompoundTold(cn),
        DesigString(s) : Ident("func_#s"),
        EmptyDesignator : Ident(""")
    }
}

type decl MakeFuncDecl(designator d, formal_part_opt fpo, name n) {
    TypeDeclPSDL(DesigTold(d) : DefIdNil, FuncTypeName(fpo, n))
}

type decl MakePSDLTypeDecl(identifier I, STR s) {
    TypeDeclPSDL(i : DefIdNil, TN_Id(Ident(s)))
}

type name CompSubDefToTN(component_subtype_def csd) {
    with (csd) {
        CompSubtypeDef(*, si) : with (si) {
            SubtypeIndConstraint(n, *) : TN_Id(NameToIdent(n)),
            SubTypeIndName(n) : TN_Id(NameToIdent(n)),
            default : TN_Id(Ident("UNDEF_TYPE"))
        }
    }
}
\begin{verbatim}

\end{verbatim}
GenTypePriv(*): MakePSDL.TypeDecl(i,"PRIVATE_TYPE"),
GenTypeDerived(gdt): MakePSDL.TypeDecl(i,
   "GENERIC_DERIVED_TYPE"),
default: Empty TypeDecl
);
/* PSDL does not currently allow value parameters as generic formals */
/* if that changes, the following lines must be uncommented to */
/* provide the translation */
/*
GenParm(p): with (p) (  
   ParamId(id,*,*,*): MakePSDL.TypeDecl(id,"GENERIC_VALUE"),  
   default: Empty TypeDecl
 ),
*/
default: Empty TypeDecl
);
/* function to build a list of generic parameters for a PSDL component */
/* specification given an input list of Ada generic formal parameters */
type _decl_s PSDLGeneric(generic_formal_part gfp) {  
   with (gfp) (  
      GenericNil : TypeDeclNil,  
      GenFormalList(gf, rest): with (MakeParam(gf)) (  
         Empty TypeDecl : PSDLGeneric(rest),  
         default: MakeParam(gf): PSDLGeneric(rest)
      )
   )
}
/* function to create the PSDL code for a list of Generic */
/* formal parameters */
interface MakeGenerics(generic_formal_part gfp) {  
   with (gfp) (  
      GenericNil: Empty Interface,  
      GenFormalList(*, *): Interface(Generics(PSDLGeneric(gfp)),ReqtsNil)
   )
}
/* create a PSDL generic type declaration from an Ada generic formal */
/* parameter list */
generic_type_decl MakeGenericTypeDecl(generic_formal_part gfp) {  
   with (gfp) (  
      GenericNil: GTDNil,  
      GenFormalList(*, *): GTD(PSDLGeneric(gfp))
   )
};
/* build the interfaces portion of a PSDL component specification */
/* given input generic formal parameter list, formal parameters, */
/* an existing interface portion, and a psdl met from an Ada program */

interface BuildInterfaces(generic_formal_part gfp, formal_part_opt fpo, interface e, psdl_met_opt pmo) {
    MakeGenerics(gfp) :: MakeInputs(fpo) :: MakeOutputs(fpo) ::
    e :: MakeMet(pmo) :: Interface_s Nil
};

/* Extract all of the exceptions from a list of declarations */
def_id_s ExtractExceptions(decl_item_s dl) {
    with (dl) (  
        DeclListNil : DefIdNil,  
        DeclList(di, rest) :  
            with (di) (  
                Decl(dcl) :  
                    with (dcl) (  
                        RenameDecl(rd) : with (rd) (  
                            RenameExc(ds, *) :  
                                ds @ ExtractExceptions(rest),  
                                default : ExtractExceptions(rest)  
                            ),  
                        ExcDecl(dids) : dids @ ExtractExceptions(rest),  
                        default : ExtractExceptions(rest)  
                    ),  
            ),  
            default : ExtractExceptions(rest)
        )
    );
}

/* create the exceptions portion of a PSDL component specification */

interface MakeExcepts(decl_item_s d) {
    with (ExtractExceptions(d)) (  
        DefIdNil : EmptyInterface,  
        DefIdList(*, *) : Interface(Excpts(ExtractExceptions(d)), RqmtsNil)
    )
};

/* combine two lists of exceptions to form a single list */

interface JoinExcepts(interface i1, interface i2) {
    with (i1) (
        i1 @ JoinExcepts(i2)
    )

given an Ada package specification and its generic header

operator MakeOpFromPkg(generic_hdr gh, pkg_spec ps, interface e) {
    with (ps) {
        Package(cn, d, pp) : with (gh) {
            GenericHdr(gfp) : with (MakeOpFromPkg(GenericHdrNil, ps, EmptyInterface)) {
                PsdlOp(c, os) : with (os) {
                    OpSpec(is, f) : PsdlOp(c, OpSpec(MakeGenerics(gfp) :: is, f))
                },
                EmptyOperator : EmptyOperator
            },
            default : with (MakeOps(d, JoinExcepts(e, MakeExcepts(d)))) {
                OLO.Nil : EmptyOperator,
                OLO.Cons(o, rest) : o
            }
        }
    }
}

component MakeCompFromPkg(generic_hdr gh, pkg_spec ps) {
    with (ps) {
        Package(cn, d, p) : CountOps(d) == 1 ?
        CompOperator(OperatorImp(MakeOpFromPkg(gh, ps, EmptyInterface),
            AdaOpImp(with (MakeOpFromPkg(gh, ps, EmptyInterface))(PsdlOp(c, o) : c,
            EmptyOperator : EmptyCompound
        )))
    }
}
GenericHdr(grp):
    CompDataType(DataType(cn,MakeType(ps,grp),AdaTypeImp(cn))),
default:
    CompDataType(DataType(cn,MakeType(ps,GenericNil),AdaTypeImp(cn)))
}

component_s MakeCompListFromDeclItemS(decl_item_s dis, interface exc) {
    with (dis) {
        DeclListNil : ComponentNil,
        DeclList(di, rest) : with (di) {
            Decl(dcl) : with (dcl) {
                SubProgDecl(sd) : with (sd) {
                    SubprogSpec(gh, ss, pmo):
                    MakeCompOpFromSubprog(gh, ss, pmo, exc)@
                    MakeCompListFromDeclItemS(rest, exc),
                    default : MakeCompListFromDeclItemS(rest, exc)
                },
                PkgDecl(pd) : with (pd) {
                    PkgSpec(gh, ps) : MakeCompFromPkg(gh, ps)@
                    MakeCompListFromDeclItemS(rest, exc),
                    default : MakeCompListFromDeclItemS(rest, exc)
                },
                default : MakeCompListFromDeclItemS(rest, exc)
            },
            default : MakeCompListFromDeclItemS(rest, exc)
        },
        default : MakeCompListFromDeclItemS(rest, exc)
    },
    default : MakeCompListFromDeclItemS(rest, exc)
}

BOOL NestedTypes(decl_item_s dis) {
    with (dis) {
        DeclListNil : false,
        DeclList(di, rest) : with (di) {
            Decl(d) : with (d) {
                PkgDecl(pd) : with (pd) {
                    PkgSpec(gh, ps) : with (ps) {
                        Package(cn, ds, pp) :
                        CountOps(ds) != 1 ? true : NestedTypes(rest)
                    },
                    default : NestedTypes(rest)
                },
                default : NestedTypes(rest)
            },
            default : NestedTypes(rest)
        },
        default : NestedTypes(rest)
    },
    default : NestedTypes(rest)
}
component_s MakeCompListFromPkg(generic_hdr gh, pkg_spec ps) {
    with (ps) {
        Package(cn, dis, p) : NestedTypes(dis) ?
        MakeCompListFromDeclItemS(dis, MakeExcepts(dis)) :
        MakeCompFromPkg(gh, ps) : ComponentNil
    }
};

.operator MakeOpFromSubprog(generic_hdr gh, subprog_spec ss, 
    psdl_met_opt pmo, interface e) {
    with (gh) {
        GenericHdr(gfp) : with (ss) {
            SubProgProc(c, f) : PsdlOp(c, OpSpec(BuildInterfaces(gfp, 
                f, e, pmo), FuncNil)),
            default : EmptyOperator
        },
        default : with (ss) {
            SubProgProc(c, f) : PsdlOp(c, OpSpec(BuildInterfaces(GenericNil, 
                f, e, pmo), FuncNil)),
            default : EmptyOperator
        }
    }
};

component_s MakeCompOpFromSubprog(generic_hdr gh, subprog_spec ss, 
    psdl_met_opt pmo, interface e) {
    with (ss) {
        SubProgProc(c, f) : CompOperator(OperatorImp(
            MakeOpFromSubprog(gh, ss, pmo, e), AdaOpImp(c))) : ComponentNil,
        default : ComponentNil
    }
};

/* create a list of PSDL operators from a list of Ada declarations */
/* incorporating the exceptions specified in exc */

op_list_opt MakeOps(decl_item_s d, interface exc) {
    with (d) {
        DeclListNil : OLO.Nil,
        DeclList(di, rest) : with (di) {
            Decl(dcl) : with (dcl) {
                SubProgDecl(sd) : with (sd) {
                    SubprogSpec(gh, ss, pmo) : 
                        MakeOpFromSubprog(gh, ss, pmo, exc) : MakeOps(rest, exc),
                    default : MakeOps(rest, exc)
                }
            }
        }
    }
};
PkgDecl(pd) : with (pd) (  
PkgSpec(gh, ps) : with (ps) (   
  Package(cn, ds, pp) : CountOps(ds) == 1 ?   
    akeOpFromPkg(gh, ps, exc):: akeOps(rest,exc)   
  :   
    MakeOps(rest,exc)   
  ),   
  default : MakeOps(rest,exc)   
  ),   
  default : MakeOps(rest,exc)   
  ),   
  default : MakeOps(rest,exc)   
)  
);  

|type_spec MakeType(pkg_spec ps, generic_formal_part gfp) {  
  with (ps) (   
    Package(c, d, p) : TypeSpec(MakeGenericTypeDecl(gfp), TDO.Nil, MakeOps(d, MakeExcepts(d)), FuncNil)   
  )  
};  

|subprog_decl : SubprogSpec{  
  subprog_spec.met = psdl_met_opt;  
}  
| GenericSubprogInst{  
  subprog_spec.met = psdl_met_opt;  
}  
| AbstractSubprogSpec{  
  subprog_spec.met = psdl_met_opt;  
}  
|;  

pkg_decl :  
PkgSpec{  
  local component_s psdl_trans;  
  psdl_trans = ($$.nesting_level == 0) ?  
    MakeCompListFromPkg(generic_hdr, pkg_spec) :  
    ComponentNil;  
};  

rename_unit : RenameSubprog{  
  subprog_spec.met = MetNull;  
}  
|;  

prot_op_decl : ProtOptSubprog {  
  subprog_spec.met = MetNull;  
}
body_stub : SubprogStub{
    subprog_spec.met = MetNull;
}


APPENDIX D. SSL SOURCE CODE: CONCRETE SYNTAX

The source code below is used to specify the concrete syntax of Ada 95 package specifications. A complete concrete syntax for Ada 95 package specifications allows SynGen to construct a parser which will accept existing Ada 95 source code as a text file, and create the attributed abstract syntax tree which is required to complete a translation.

/* ***********************************************************************

COMPILATION { syn compilation abs; suggesting a structure for compilation }
compilation ~ COMPILATION.abs;

COMP_UNIT_LIST { syn comp_unit_list rev,
       inh comp_unit_list tail; suggesting a structure for a list of compilation units }
comp_unit_list ~ COMP_UNIT_LIST.rev
{COMP_UNIT_LIST.tail = CUListNil; suggesting the structure of the list tail };

PRAGMA { syn pragma abs; suggesting a structure for pragmas }
pragma ~ PRAGMA.abs;

PRAGMA_ARG_S { syn pragma_arg_s rev;
       inh pragma_arg_s tail; suggesting a structure for a list of pragmas arguments }
pragma_arg_s ~ PRAGMA_ARG_S.rev
{PRAGMA_ARG_S.tail = PragmaArgNil; suggesting the structure of the list尾 };

PRAGMA_ARG { syn pragma_arg abs; suggesting a structure for pragmas arguments }
pragma_arg ~ PRAGMA_ARG.abs;

PRAGMA_S { syn pragma_s rev;
       inh pragma_s tail; suggesting a structure for a list of pragmas }
pragma_s ~ PRAGMA_S.rev
{PRAGMA_S.tail = PragmasNil; suggesting the structure of the list tail };

DECL { syn decl abs; suggesting a structure for declarations }
decl ~ DECL.abs;

117
DEF_ID_S { syn def_id_s rev;  
    inh def_id_s tail; };  
def_id_s ~ DEF_ID_S.rev  
    {DEF_ID_S.tail = DefIdNil;};

OBJECT_QUALIFIER_OPT { syn object_qualifier_opt abs; };  
object_qualifier_opt ~ OBJECT_QUALIFIER_OPT.abs;

OBJECT_SUBTYPE_DEF { syn object_subtype_def abs; };  
object_subtype_def ~ OBJECT_SUBTYPE_DEF.abs;

INIT_OPT { syn init_opt abs; };  
init_opt ~ INIT_OPT.abs;

DISCRIM_PART_OPT { syn discrim_part_opt abs; };  
discrim_part_opt ~ DISCRIM_PART_OPT.abs;

TYPE_COMPLETION { syn type_completion abs; };  
type_completion ~ TYPE_COMPLETION.abs;

TYPE_DEF { syn type_def abs; };  
type_def ~ TYPE_DEF.abs;

SUBTYPE_IND { syn subtype_ind abs; };  
subtype_ind ~ SUBTYPE_IND.abs;

CONSTRAINT { syn constraint abs; };  
constraint ~ CONSTRAINT.abs;

DERIVED_TYPE { syn derived_type abs; };  
derived_type ~ DERIVED_TYPE.abs;

RANGE_CONSTRAINT { syn range_constraint abs; };  
range_constraint ~ RANGE_CONSTRAINT.abs;

RANGE { syn range abs; };  
range ~ RANGE.abs;

ENUM ID_S { syn enum_id_s rev;  
    inh enum_id_s tail; };  
enum_id_s ~ ENUM_ID_S.rev  
    {ENUM_ID_S.tail = EnumIdNil;};

ENUM_ID { syn enum_id abs; };  
enum_id ~ ENUM_ID.abs;

INTEGER_TYPE { syn integer_type abs; };  
integer_type ~ INTEGER_TYPE.abs;

RANGE_SPEC { syn range_spec abs; };
range_spec ~ RANGE_SPEC.abs;

RANGE_SPEC_OPT { syn range_spec_opt abs; }
range_spec_opt ~ RANGE_SPEC_OPT.abs;

REAL_TYPE { syn real_type abs; }
real_type ~ REAL_TYPE.abs;

FIXED_TYPE { syn fixed_type abs; }
fixed_type ~ FIXED_TYPE.abs;

ARRAY_TYPE { syn array_type abs; }
array_type ~ ARRAY_TYPE.abs;

COMPONENT_SUBTYPE_DEF { syn component_subtype_def abs; }
component_subtype_def ~ COMPONENT_SUBTYPE_DEF.abs;

ALIASED_OPT { syn aliased_opt abs; }
aliased_opt ~ ALIASED_OPT.abs;

INDEX_S { syn index_s rev;
    inh index_s_tail; }
index_s ~ INDEX_S.rev
    {INDEX_S.tail = IndexNil;};

ITER_DISCRETE_RANGE_S { syn iter_discrete_range_s rev;
    inh iter_discrete_range_s_tail; }
iter_discrete_range_s ~ ITER_DISCRETE_RANGE_S.rev
    {ITER_DISCRETE_RANGE_S.tail = DiscreteRangeNil;};

DISCRETE_RANGE { syn discrete_range abs; }
discrete_range ~ DISCRETE_RANGE.abs;

RANGE_CONSTR_OPT { syn range_constr_opt abs; }
rangle_constr_opt ~ RANGE_CONSTR_OPT.abs;

RECORD_DEF { syn record_def abs; }
record_def ~ RECORD_DEF.abs;

TAGGED_OPT { syn tagged_opt abs; }
tagged_opt ~ TAGGED_OPT.abs;

COMP_LIST { syn comp_list abs; }
comp_list ~ COMP_LIST.abs;

COMP_DECL_S { syn comp_decl_s abs; }
comp_decl_s ~ COMP_DECL_S.abs;

VARIANT_PART_OPT { syn variant_part_opt abs; }
variant_part_opt ~ VARIANT_PART_OPT.abs;
COMP_DECL { syn comp_decl abs; }
comp_decl ~ COMP_DECL.abs;

DISCRIM_SPEC_S { syn discrim_spec_s rev;
  inh discrim_spec_s tail; }
discrim_spec_s ~ DISCRIM_SPEC_S.rev
  {DISCRIM_SPEC_S.tail = DiscrimSpecNil;};

DISCRIM_SPEC { syn discrim_spec abs; }
discrim_spec ~ DISCRIM_SPEC.abs;

ACCESS_OPT { syn access_opt abs; }
access_opt ~ ACCESS_OPT.abs;

VARIANT_PART { syn variant_part abs; }
variant_part ~ VARIANT_PART.abs;

VARIANT_S { syn variant_s rev;
  inh variant_s tail; }
variant_s ~ VARIANT_S.rev
  {VARIANT_S.tail = VariantNil;};

VARIANT { syn variant abs; }
variant ~ VARIANT.abs;

CHOICE_S { syn choice_s rev;
  inh choice_s tail; }
choice_s ~ CHOICE_S.rev
  {CHOICE_S.tail = ChoiceNil;};

CHOICE { syn choice abs; }
choice ~ CHOICE.abs;

DISCRETE_WITH_RANGE { syn discrete_with_range abs; }
discrete_with_range ~ DISCRETE_WITH_RANGE.abs;

ACCESS_TYPE { syn access_type abs; }
access_type ~ ACCESS_TYPE.abs;

PROT_OPT { syn prot_opt abs; }
prot_opt ~ PROT_OPT.abs;

DECL_ITEM_S { syn decl_item_s rev;
  inh decl_item_s tail; }
decl_item_s ~ DECL_ITEM_S.rev
  {DECL_ITEM_S.tail = DeclListNil;};

DECL_ITEM { syn decl_item abs; }
decl_item ~ DECL_ITEM.abs;

NAME { syn name abs; }
name ~ NAME_abs;

MARK { syn mark abs; };
mark ~ MARK_abs;

TICDOT { syn ticdot abs; };
ticdot ~ TICDOT_abs;

MARKLIST { syn marklist abs; };
marklist ~ MARKLIST_abs;

COMPOUND_NAME { syn compound_name_rev; 
inh compound_name tail; };
compound_name ~ COMPOUND_NAME_rev 
{COMPOUND_NAME.tail = EmptyCompound;};

C_NAME_LIST { syn c_name_list rev; 
inh c_name_list tail; };
c_name_list ~ C_NAME_LIST_rev 
{C_NAME_LIST.tail = CompoundNameNil;};

VALUE_S { syn value_s_rev; 
inh value_s tail; };
value_s ~ VALUE_S_rev 
{VALUE_S.tail = ValueNil;};

VALUE { syn value abs; };
value ~ VALUE_abs;

SELECTED_COMP { syn selected_comp abs; };
selected_comp ~ SELECTED_COMP_abs;

ATTRIBUTE_ID { syn attribute_id abs; };
attribute_id ~ ATTRIBUTE_ID_abs;

NUMERIC_LIT { syn numeric_lit abs; };
numeric_lit ~ NUMERIC_LIT_abs;

LITERAL { syn literal abs; };
literal ~ LITERAL_abs;

AGGREGATE { syn aggregate abs; };
aggregate ~ AGGREGATE_abs;

VALUE_S_2 { syn value_s_2 abs; };
value_s_2 ~ VALUE_S_2_abs;

COMP_ASSOC { syn comp_assoc abs; };
comp_assoc ~ COMP_ASSOC_abs;

EXPRESSION { syn expression abs; };

121
expression ~ EXPRESSION.abs;

RELATION { syn relation abs; }
relation ~ RELATION.abs;

SIMPLE_EXPRESSION { syn simple_expression abs; }
simple_expression ~ SIMPLE_EXPRESSION.abs;

TERM { syn term abs; }
term ~ TERM.abs;

FACTOR { syn factor abs; }
factor ~ FACTOR.abs;

PRIMARY { syn primary abs; }
primary ~ PRIMARY.abs;

QUALIFIED { syn qualified abs; }
qualified ~ QUALIFIED.abs;

ALLOCATOR { syn allocator abs; }
allocator ~ ALLOCATOR.abs;

SUBPROG_DECL { syn subprog_decl abs; }
subprog_decl ~ SUBPROG_DECL.abs;

PSDL_MET_OPT { syn psdl_met_opt abs; }
psdl_met_opt ~ PSDL_MET_OPT.abs;

SUBPROG_SPEC { syn subprog_spec abs; }
subprog_spec ~ SUBPROG_SPEC.abs;

DESIGNATOR { syn designator abs; }
designator ~ DESIGNATOR.abs;

FORMAL_PART_OPT { syn formal_part_opt abs; }
formal_part_opt ~ FORMAL_PART_OPT.abs;

PARAM_S { syn param_s rev; 
    inh param_s tail; }
param_s ~ PARAM_S.rev 
    {PARAM_S.tail = ParamNil;};

PARAM { syn param abs; }
param ~ PARAM.abs;

MODE { syn mode abs; }
mode ~ MODE.abs;

PKGDECL { syn pkg_decl abs; }
pkg_decl ~ PKGDECL.abs;
PKG_SPEC \{ syn pkg_spec abs; \};
pkg_spec \sim PKG_SPEC.abs;

PRIVATE_PART \{ syn private_part abs; \};
private_part \sim PRIVATE_PART.abs;

PRIVATE_TYPE \{ syn private_type abs; \};
private_type \sim PRIVATE_TYPE.abs;

LIMITED_OPT \{ syn limited_opt abs; \};
limited_opt \sim LIMITED_OPT.abs;

USE_CLAUSE \{ syn use_clause abs; \};
use_clause \sim USE_CLAUSE.abs;

NAME_S \{ syn name_s rev;
    inh name_s tail; \};
name_s \sim NAME_S.rev
    \{ NAME_S.tail = NameNil; \};

RENAME_DECL \{ syn rename_decl abs; \};
rename_decl \sim RENAME_DECL.abs;

RENAME_UNIT \{ syn rename_unit abs; \};
rename_unit \sim RENAME_UNIT.abs;

RENAMEDECL \{ syn renames abs; \};
renames \sim RENAMES.abs;

TASK_SPEC \{ syn task_spec abs; \};
task_spec \sim TASK_SPEC.abs;

TASK_DEF \{ syn task_def abs; \};
task_def \sim TASK_DEF.abs;

TASK_PRIVATE_OPT \{ syn task_private_opt abs; \};
task_private_opt \sim TASK_PRIVATE_OPT.abs;

PROT_SPEC \{ syn prot_spec abs; \};
prot_spec \sim PROT_SPEC.abs;

PROT_DEF \{ syn prot_def abs; \};
prot_def \sim PROT_DEF.abs;

PROT_PRIVATE_OPT \{ syn prot_private_opt abs; \};
prot_private_opt \sim PROT_PRIVATE_OPT.abs;

PROTOPDECL \{ syn prot_op_decl_s rev;
    inh prot_op_decl_s tail; \};
prot_op_decl_s \sim PROTOPDECL_S.rev

123
{PROTOPDECLS.tail = ProtOptDeclListNil;}

PROTOPDECL { syn prot_op_decl abs; }
prot_op_decl ~ PROTOPDECL.abs;

PROTELEM DECL S { syn prot elem decl_s rev;
in prot elem decl_s tail; }
prot elem decl_s ~ PROTELEMDECL_S.rev
{PROTELEMDECL_S.tail = ProtElemDeclNil;}

PROTELEMDECL { syn prot elem decl abs; }
prot elem decl ~ PROTELEMDECL.abs;

ENTRYDECLS { syn entry decl_s abs; }
entry decl_s ~ ENTRYDECL_S.abs;

ENTRYDECL { syn entry decl abs; }
entry decl ~ ENTRYDECL.abs;

REPSPEC_S { syn rep spec_s abs; }
rep spec_s ~ REP_SPEC_S.abs;

COMP_UNIT { syn comp unit abs; }
comp_unit ~ COMP_UNIT.abs;

PRIVATE_OPT { syn private_opt abs; }
private_opt ~ PRIVATE_OPT.abs;

CONTEXTSPEC OPT { syn context-spec-opt abs; }
context spec opt ~ CONTEXTSPEC_OPT.abs;

CONTEXT_SPEC { syn context spec abs; }
context spec ~ CONTEXT_SPEC.abs;

WITH_CLAUSE { syn with clause abs; }
with clause ~ WITH_CLAUSE.abs;

USE_CLAUSEOPT { syn use clause-opt rev;
in use clause-opt tail; }
use clause-opt ~ USE_CLAUSEOPT.rev
{USE_CLAUSEOPT.tail = UseClauseOptNil;}

BODY_STUB { syn body stub abs; }
body stub ~ BODY_STUB.abs;

GENERICFORMALPART { syn generic formal part rev;
in generic formal part tail; }
generic formal part ~ GENERICFORMALPART.rev
{GENERICFORMALPART.tail = GenericNil;}

GENERICFORMAL { syn generic formal abs; }

generic_formal ~ GENERIC_FORMAL.abs;

GENERIC_DISCRIM_PART_OPT { syn generic_discrim_part_opt abs; }
generic_discrim_part_opt ~ GENERIC_DISCRIM_PART_OPT.abs;

SUBP_DEFAULT { syn subp_default abs; }
subp_default ~ SUBP_DEFAULT.abs;

GENERIC_TYPE_DEF { syn generic_type_def abs; }
generic_type_def ~ GENERIC_TYPE_DEF.abs;

GENERIC_DERIVED_TYPE { syn generic Derived_type abs; }
generic Derived_type ~ GENERIC_DERIVED_TYPE.abs;

GENERIC_INST { syn generic inst abs; }
generic_inst ~ GENERIC_INST.abs;

REP_SPEC { syn rep_spec abs; }
rep_spec ~ REP_SPEC.abs;

ALIGN_OPT { syn align opt abs; }
align_opt ~ ALIGN_OPT.abs;

COMP_LOC_S { syn comp_loc_s abs; }
comp_loc_s ~ COMP_LOC_S.abs;

INTEGER_CS { syn integer abs; }
integer ~ INTEGER_CS.abs;
INTEGER_CS :: (INTEGER) {$$.abs = Integer(INTEGER);} 

COMPILATION ::= () {$$.abs = CompilationNil; }
| (PRAGMA_S COMP_UNIT_LIST) {
PRAGMA_S.tail = PragmasNil;
COMP_UNIT_LIST.tail = CUListNil;
$$.abs = Compilation(PRAGMA_S.rev, COMP_UNIT_LIST.rev);} 

COMP_UNIT_LIST ::= (COMP_UNIT) {$$.rev = COMP_UNIT.abs::$$tail;} 
| (COMP_UNIT_LIST COMP_UNIT) {
COMP_UNIT_LIST$2.tail = COMP_UNIT.abs::$$tail;
$$.rev = COMP_UNIT_LIST$2.rev;
}

PRAGMA ::= (PRAGMA ID ';' ) { $$abs = Pragmas(Ident(ID)); } 
| (PRAGMA ID (' PRAGMA_ARG_S ') ';') {
PRAGMA_ARG_S.tail = PragmaArgNil;
$$abs = PragmaSimple(Ident(ID), PRAGMA_ARG_S.rev);
}
PRAGMA_ARG_S := (PRAGMA_ARG) { $$\cdot \text{rev} = \text{PRAGMA_ARG.abs}::$$\.tail; }
| (PRAGMA_ARG_S \', PRAGMA_ARG) {
PRAGMA_ARG_S$$2.tail = PRAGMA_ARG.abs::$$\cdot\text{tail};
$$\cdot\text{rev} = \text{PRAGMA_ARG_S$$2.rev};
}

PRAGMA_ARG := (EXPRESSION) { $$\cdot\text{abs} = \text{PragmaExp(EXPRESSION.abs)}; }
| (ID RIGHT SHAFT EXPRESSION) {
$$\cdot\text{abs} = \text{PragmaNameExp(Ident(ID), EXPRESSION.abs)};
}

PRAGMA_S := () { $$\cdot\text{rev} = $$\cdot\text{tail}; }
| (PRAGMA_S PRAGMA ) {
PRAGMA_S$$2.tail = PRAGMA.abs::$$\cdot\text{tail};
$$\cdot\text{rev} = \text{PRAGMA_S$$2.rev};
}

OBJECTDECL {syn dccl abs;};
NUMBERDECL {syn deci abs;};
TYPEDECL {syn decl abs;};
EXCEPTIONDECL {syn dccl abs;};
DECL := (OBJECTDECL) {$$\cdot\text{abs} = \text{OBJECTDECL.abs};}$
| (NUMBERDECL) {$$\cdot\text{abs} = \text{NUMBERDECL.abs};}$
| (TYPEDECL) {$$\cdot\text{abs} = \text{TYPEDECL.abs};}$
| (SUBTYPE ID IS SUBTYPE_IND \';') { $$\cdot\text{abs} = \text{SubTypeDecl(Ident(ID), SUBTYPE_IND.abs)}; }
| (SUBPROG_DECL) { $$\cdot\text{abs} = \text{SubProgDecl(SUBPROG_DECL.abs)}; }
| (PKG_DECL) { $$\cdot\text{abs} = \text{PkgDecl(PKG_DECL.abs)}; }
| (TASK_SPEC \';') { $$\cdot\text{abs} = \text{TaskDecl(TASK_SPEC.abs)}; }
| (PROT_SPEC \';') { $$\cdot\text{abs} = \text{ProtDecl(PROT_SPEC.abs)}; }
| (EXCEPTIONDECL) { $$\cdot\text{abs} = \text{EXCEPTIONDECL.abs}; }
| (RENAME DECL) { $$\cdot\text{abs} = \text{RenameDecl(RENAMEDECL.abs)}; }
| (BODY_STUB) { $$\cdot\text{abs} = \text{BodyStubDecl(BODYSTUB.abs)}; }


OBJECTDECL :=
| (DEF_ID S :: OBJECT_QUALIFIER_OPT OBJECT_SUBTYPE_DEF INIT_OPT \';') {
DEF_ID S.tail = DefIdNil;
$$\cdot\text{abs} = \text{ObjDecl(DEF_ID_S.rev, OBJECT_QUALIFIER_OPT.abs, OBJECT_SUBTYPE_DEF.abs, INIT_OPT.abs)};
}

NUMBERDECL := (DEF_ID S :: CONSTANT IS_ASSIGNED EXPRESSION \';') {
DEF_ID_S.tail = DefIdNil;

126
$$abs = \text{NumDecl}(\text{DEF_ID}\_S.rev, \text{EXPRESSION.abs};) \}
$$

\small

\begin{verbatim}
\textit{TYPE_DECL} ::= (\textit{TYPE ID DISCRIM\_PART\_OPT TYPE\_COMPLETION \";\"}) \{
\textit{abs} = \text{TypeDecl}(\text{Ident(ID), DISCRIM\_PART\_OPT.abs, TYPE\_COMPLETION.abs};)
\}
\}
\textit{EXCEPTION\_DECL} ::= (\textit{DEF_ID\_S \text{"\, EXCEPTION \";\"}) \{
\textit{DEF_ID\_S.tail} = \text{DefIdNil};
\textit{abs} = \text{ExecDecl}(\text{DEF_ID\_S.rev};)
\}
\}
\textit{DEF_ID\_S} ::= (\textit{ID} {\textit{rev} = \text{Ident(ID)}::\textit{tail};}
| (\textit{DEF_ID\_S \text{" ID}) \{ \textit{DEF_ID\_S.tail} = \text{Ident(ID)}::\textit{tail};
\textit{rev} = \text{DEF_ID\_S.rev};
\}
\}
\textit{OBJECT\_QUALIFIER\_OPT} ::= () \{ \textit{abs} = \text{ObjQualOptNull}; }
| (\textit{ALIASED}) \{ \textit{abs} = \text{Aliased}; }
| (\textit{CONSTANT}) \{ \textit{abs} = \text{Constant}; }
| (\textit{ALIASED CONSTANT}) \{ \textit{abs} = \text{AliasedConst}; }
\}
\textit{OBJECT\_SUBTYPE\_DEF} ::= (\textit{SUBTYPE\_IND}) \{ \textit{abs} = \text{Subtypelnd(SUBTYPE\_IND.abs); }
| (\textit{ARRAY\_TYPE}) \{ \textit{abs} = \text{ArrayType(ARRAY\_TYPE.abs); }
\}
\}
\textit{INIT\_OPT} ::= () \{ \textit{abs} = \text{InitOptNull(); }
| (\textit{IS\_ASSIGNED\_EXPRESSION}) \{ \textit{abs} = \text{ExprInitOpt(EXPRESSION.abs); }
\}
\}
\textit{DISCRIM\_PART\_OPT} ::= () \{ \textit{abs} = \text{DiscrimPartNull(); }
| (\textit{" DISCRIM\_SPEC\_S \"}) \{
\textit{DISCRIM\_SPEC\_S.tail} = \text{DiscrimSpecNil};
\textit{abs} = \text{DiscrimPart(DISCRIM\_SPEC\_S.rev); }
\}
| (\textit{" BOX \")} \{ \textit{abs} = \text{Box(); }
\}
\}
\textit{TYPE\_COMPLETION} ::= () \{ \textit{abs} = \text{TypeComplNull(); }
| (\textit{IS\_TYPE\_DEF}) \{ \textit{abs} = \text{TypeDefCompl(TYPE\_DEF.abs); }
\}
\}
\textit{TYPE\_DEF} ::= (\textit{" ENUM\_ID\_S \")} \{
\textit{ENUM\_ID\_S.tail} = \text{EnumIdNil};
\textit{abs} = \text{EnumTypeDef(ENUM\_ID\_S.rev); }
\}
| (\textit{INTEGER\_TYPE}) \{ \textit{abs} = \text{IntTypeDef(INTEGER\_TYPE.abs); }
| (\textit{REAL\_TYPE}) \{ \textit{abs} = \text{RealTypeDef(REAL\_TYPE.abs); }
| (\textit{ARRAY\_TYPE}) \{ \textit{abs} = \text{ArrayType(ARRAY\_TYPE.abs); }
| (\textit{TAGGED\_OPT LIMITED\_OPT RECORD\_DEF})
\end{verbatim}

\textbf{127}
$$\text{abs} = \text{RecordType}(\text{TAGGED\_OPT.abs, LIMITED\_OPT.abs, RECORD\_DEF.abs});$

\(\text{ACCESS\_TYPE}\) \{ \text{abs} = \text{AccessTypeDef(ACCESS\_TYPE.abs);} \}

\(\text{DERIVED\_TYPE}\) \{ \text{abs} = \text{DerivedTypeDef(DERIVED\_TYPE.abs);} \}

\(\text{PRIVATE\_TYPE}\) \{ \text{abs} = \text{PrivateTypeDef(PRIVATE\_TYPE.abs);} \}

\text{SUBTYPE\_IND} ::= (\text{NAME CONSTRAINT}) \{
\quad \text{abs} = \text{SubtypelndConstraint(NAME.abs, CONSTRAINT.abs);} \\
\quad \text{(NAME)} \{ \text{abs} = \text{SubTypelndName(NAME.abs);} \}
\}

\text{CONSTRAINT} ::= (\text{RANGE\_CONSTRAINT}) \{
\quad \text{abs} = \text{RangeConstraint(RANGE\_CONSTRAINT.abs);} \\
\}

\text{DERIVED\_TYPE} ::= 
\quad (\text{NEW SUBTYPE\_IND}) \{ \text{abs} = \text{NewDerivedType(SUBTYPE\_IND.abs);} \}
\quad (\text{NEW SUBTYPE\_IND WITH PRIVATE}) \{ 
\quad \text{abs} = \text{NewDerivedWithPrivate(SUBTYPE\_IND.abs);} \\
\quad \text{(NEW SUBTYPE\_IND WITH RECORD\_DEF)} \{ 
\quad \text{abs} = \text{NewDerivedWithRecord(SUBTYPE\_IND.abs, RECORD\_DEF.abs);} \\
\quad \text{(ABSTRACT NEW SUBTYPE\_IND WITH PRIVATE)} \{ 
\quad \text{abs} = \text{AbsNewDerivedWithPrivate(SUBTYPE\_IND.abs);} \\
\quad \text{(ABSTRACT NEW SUBTYPE\_IND WITH RECORD\_DEF)} \{ 
\quad \text{abs} = \text{AbsNewDerivedWithRecord(SUBTYPE\_IND.abs, RECORD\_DEF.abs);} \\
\}

\text{RANGE\_CONSTRAINT} ::= (\text{RANGE RANGE}) \{ \text{abs} = \text{Range(RANGE.abs);} \}

\text{RANGE} ::= (\text{SIMPLE\_EXPRESSION DOT\_DOT SIMPLE\_EXPRESSION}) \{
\quad \text{abs} = \text{SimpleRange(SIMPLE\_EXPRESSION$1.abs, SIMPLE\_EXPRESSION$2.abs);} \\
\quad \text{(NAME TIC Range)} \{ \text{abs} = \text{NameTicRange(NAME.abs);} \}
\quad \text{(NAME TIC Range \(\text{' EXPRESSION \'}\))} \{ 
\quad \text{abs} = \text{NameTicRangeExp(NAME.abs, EXPRESSION.abs);} \\
\}

\text{ENUM\_ID\_S} ::= (\text{ENUM\_ID}) \{ \text{rev} = \text{ENUM\_ID.abs::$$tail;} \}
\quad \text{(ENUM\_ID\_S \text{', ' ENUM\_ID})} \{ 

ENUM_ID ::= (ID) { $$.abs = Id(Ident(ID)); } |
(CHAR_LIT) { $$.abs = CharLit(CHAR_LIT); }

INTEGER_TYPE ::= (RANGE_SPEC) { $$.abs = RangeSpec(RANGE_SPEC.abs); }
| (MOD EXPRESSION) { $$.abs = ModExpr(EXPRESSION.abs); }

RANGE_SPEC ::= (RANGE_CONSTRAINT) {
    $$.abs = RangeSpecConstr(RANGE_CONSTRAINT.abs);
}

RANGE_SPEC_OPT ::= () { $$.abs = RangeSpecOptNull(); }
| (RANGE_SPEC) { $$.abs = RangeSpecOpt(RANGE_SPEC.abs); }

REAL_TYPE ::= (DIGITS EXPRESSION RANGE_SPEC_OPT) {
    $$.abs = FloatType(EXPRESSION.abs, RANGE_SPEC_OPT.abs); }
| (FIXED_TYPE) { $$.abs = FixedType(FIXED_TYPE.abs); }

FIXED_TYPE ::= (DELTA EXPRESSION RANGE_SPEC) {
    $$.abs = FixedDelta(EXPRESSION.abs, RANGE_SPEC.abs); }
| (DELTA EXPRESSION DIGITS EXPRESSION RANGE_SPEC_OPT) {
    $$.abs = FixedDeltaDigits(EXPRESSION$1.abs, EXPRESSION$2.abs,
        RANGE_SPEC_OPT.abs); }

ARRAY_TYPE ::= (ARRAY ['INDEX_S '] OF COMPONENT_SUBTYPE_DEF) {
    INDEX_S.tail = IndexNil;
    $$.abs = UnconstrArray(INDEX_S.rev,
        COMPONENT_SUBTYPE_DEF.abs); }
| (ARRAY ['ITER_DISCRETE_RANGE_S '] OF COMPONENT_SUBTYPE_DEF) {
    ITER_DISCRETE_RANGE_S.tail = DiscreteRangeNil;
    $$.abs = ConstrArray(ITER_DISCRETE_RANGE_S.rev,
        COMPONENT_SUBTYPE_DEF.abs); }

COMPONENT_SUBTYPE_DEF ::= (ALIASED_OPT SUBTYPE_IND) { $$.abs =
    CompSubtypeDef(ALIASED_OPT.abs, SUBTYPE_IND.abs); }

ALIASED_OPT ::= () { $$.abs = AliasedOptNull(); }
| (ALIASED) { $$.abs = AliasedOpt(); }

129
INDEX_S := (NAME RANGE BOX) { $$\text{.rev} = \text{NAME.abs.}$$; tail; }
| (INDEX_S \', NAME RANGE BOX) {
  INDEX_S.tail = NAME.abs.:$$\text{.tail};
  $$\text{.rev} = \text{INDEX_S.tail;}
}

ITER_DISCRETE_RANGE_S := (DISCRETE_RANGE) {
  $$\text{.rev} = \text{DISCRETE_RANGE.abs.}$$; tail;
| (ITER_DISCRETE_RANGE_S \', DISCRETE_RANGE) {
  ITER_DISCRETE_RANGE_S.tail = DISCRETE_RANGE.abs.;$$\text{.tail;}
  $$\text{.rev} = \text{ITER_DISCRETE_RANGE_S.tail;}
}

DISCRETE_RANGE ::= (NAME RANGE_CONSTRAINT_OPT) {
  $$\text{.abs} = \text{DiscRangeName}($$NAME.abs,$$\text{.RANGE_CONSTRAINT_OPT.abs;})
| (RANGE) { $$\text{.abs} = \text{DiscRangeRange}($$RANGE.abs;)
}

RANGE_CONSTRAINT_OPT ::= () { $$\text{.abs} = \text{EmptyRangeConstrOpt();}
| (RANGE_CONSTRAINT) { $$\text{.abs} = \text{RangeConstr}($$\text{RANGE_CONSTRAINT.abs;})

RECORD_DEF ::= (RECORD PRAGMA_S COMP_LIST END RECORD) {
  PRAGMA_S.tail = PragmasNil;
  $$\text{.abs} = \text{Record}($$PRAGMA_S.rev, COMP_LIST.abs;)
| (NULL RECORD) { $$\text{.abs} = \text{NullRecord();}

TAGGED_OPT ::= () { $$\text{.abs} = \text{TaggedOptNull();}
| (TAGGED) { $$\text{.abs} = \text{Tagged();}
| (ABSTRACT TAGGED) { $$\text{.abs} = \text{AbstractTagged();}

COMP_LIST ::= (COMP_DECL_S VARIANT_PART_OPT) {
  $$\text{.abs} = \text{CompListWithVariant}($$COMP_DECL_S.abs,
  VARIANT_PART_OPT.abs;)
| (VARIANT_PART PRAGMA_S) {
  PRAGMA_S.tail = PragmasNil;
  $$\text{.abs} = \text{CompListWithPragma}($$VARIANT_PART.abs,
  PRAGMA_S.rev;)
| (NULL \', PRAGMA_S) {
  PRAGMA_S.tail = PragmasNil;
  $$\text{.abs} = \text{NullWithPragma}($$PRAGMA_S.rev;)

COMP_DECL_S ::= (COMP_DECL) { $$\text{.abs} = \text{CompDecl}($$COMP_DECL.abs;)
| (COMP_DECL $PRAGMA_S COMP_DECL) {
  PRAGMA_S.tail = PragmasNil;
  $$\text{.abs} = \text{CompDeclList}($$COMP_DECL_S$$_2.abs, PRAGMA_S.rev,
COMPDECL::

VAR_INT_PART_OPT ::= (PRAGMA_S) {
    PRAGMA_S.tail = PragmasNil;
    $$\text{abs} = \text{VariantPartOptPragma}(\text{PRAGMA_S}.rev);
} 

| (PRAGMA_S VARIANT_PART PRAGMA_S) {
    PRAGMA_S$1.tail = PragmasNil;
    PRAGMA_S$2.tail = PragmasNil;
    $$\text{abs} = \text{VariantPartOpt}(\text{PRAGMA_S}$1.rev,
                        VARIANT_PART.abs, PRAGMA_S$2.rev); 
} 

COMPDECL ::= (DEF_ID ' ' COMPONENT_SUBTYPE_DEF INIT_OPT ' ')

DISCRIM_SPEC_S ::= (DISCRIM_SPEC) {$$.rev = DISCRIM_SPEC.abs::$$.tail;)

DISCRIM_SPEC ::= (DEF_ID ' ' ACCESS_OPT MARK INIT_OPT)

ACCESS_OPT ::= () { $$\text{abs} = \text{AccessOptNull}();
} | (ACCESS) { $$\text{abs} = \text{AccessOpt}();
}

VAR_INT_PART ::= (CASE ID IS PRAGMA_S VARIANT_S END CASE ' ')

VARIANT_S ::= (VARIANT) { $$\text{rev} = \text{VARIANT.abs}::$$\text{tail};
} | (VARIANT_S VARIANT) {
    VARIANT_S$2.tail = VARIANT.abs::$$\text{tail};
    $$\text{rev} = \text{VARIANT_S}$2.rev;
} 

VARIANT ::= (WHEN CHOICE_S RIGHT_SHAFT PRAGMA_S COMP_LIST) 

CHOICE_S.tail = ChoiceNil;
PRAGMA_S.tail = PragmasNil;
$$abs = VariantChoice(CHOICE_S.rev, PRAGMA_S.rev,
                    COMP_LIST.abs); }

CHOICE_S ::= (CHOICE) { $$rev = CHOICE.abs::$$tail; }
   | (CHOICE_S |'' CHOICE) {
       CHOICE_S$2.tail = CHOICE.abs::ChoiceNil;
       $$rev = CHOICE_S$2.rev;
   }

CHOICE ::= (EXPRESSION) { $$abs = ChoiceExpr(EXPRESSION.abs); }
   | (DISCRETE_WITH_RANGE) {
       $$abs = ChoiceRange(DISCRETE_WITH_RANGE.abs);
   }
   | (OTHERS) { $$abs = ChoiceOthers(); }

DISCRETE_WITH_RANGE ::= (NAME RANGE_CONSTRAINT) {
    $$abs = DiscreteNameRange(NAME.abs,
                             RANGE_CONSTRAINT.abs);
}
   | (RANGE) { $$abs = DiscreteWithRange(RANGE.abs); }

ACCESS_TYPE ::= (ACCESS SUBTYPE_IND) { $$abs = AccessSubtype(SUBTYPE_IND.abs); }
   | (ACCESS CONSTANT SUBTYPE_IND) {
       $$abs = AccessConstSubtype(SUBTYPE_IND.abs);
   }
   | (ACCESS ALL SUBTYPE_IND) { 
       $$abs = AccessAllSubtype(SUBTYPE_IND.abs);
   }
   | (ACCESS PROT_OPT PROCEDURE FORMAL_PART_OPT) {
       $$abs = AccessProcedure(PROT_OPT.abs, FORMAL_PART_OPT.abs);
   }
   | (ACCESS PROT_OPT FUNCTION FORMAL_PART_OPT RETURN MARK) {
       $$abs = AccessFunction(PROT_OPT.abs,
                               FORMAL_PART_OPT.abs, MARK.abs);
   }

PROT_OPT ::= () { $$abs = ProtOptNull(); }
   | (PROTECTED) { $$abs = Protected(); }

DECL_ITEM_S ::= () { $$rev = $$tail; }
   | (DECL_ITEM_S DECL_ITEM) {
DECLITEM$_2$.tail = DECLITEM.abs::$$.tail;
$$$.rev = DECLITEM$_2$.rev;
}

DECL_ITEM ::= (DECL) { $$.abs = Decl(DECL.abs); }
   | (USE_CLAUSE) { $$ abs = UseClauseDecl(USE_CLAUSE.abs); }
   | (REP_SPEC) { $$ abs = DeclRepSpec(REP_SPEC.abs); }
   | (PRAGMA) { $$ abs = DeclPragma(PRAGMA.abs); }

NAME ::= (ID) { $$ abs = SimpleName(Ident(ID)); }
   | (NAME '(' VALUE_S ')') { 
      VALUE_S.tail = ValueNil;
      $$ abs = IndexComp(NAME$_2$.abs, VALUE_S.rev);
   }
   | (SELECTED_COMP) { $$ abs = SelectedComp(SELECTED_COMP.abs); }
   | (NAME TIC ATTRIBUTE ID) {
      $$ abs = Attribute(NAME$_2$.abs, ATTRIBUTE ID.abs);
   }
   | (QUOTED_STRING) { $$ abs = OperatorSymbol(QUOTED_STRING); }

MARK ::= (ID MARKLIST) {
      $$ abs = Mark(Ident(ID), MARKLIST.abs); }

TICDOT ::= (TIC ATTRIBUTE ID) { $$ abs = TicOpt(ATTRIBUTE_ID.abs); }
   | (': ID) { $$ abs = DotOpt(Ident(ID)); }

MARKLIST ::= () { $$ abs = MarkListNil; }
   | (TICDOT MARKLIST) {
      $$ abs = TICDOT.abs::MARKLIST$_2$.abs;}

COMPOUND_NAME ::= (ID) { $$ .rev = Ident(ID)::$$ .tail; }
   | (COMPOUND_NAME ': ID) {
      COMPOUND_NAME$_2$.tail = Ident(ID)::$$ .tail;
      $$ .rev = COMPOUND_NAME$_2$.rev;
   }

C_NAME_LIST ::= (COMPOUND_NAME) {
      COMPOUND_NAME.tail = EmptyCompound;
      $$ .rev = COMPOUND_NAME.rev::$$ .tail;
   }
   | (C_NAME_LIST ': COMPOUND_NAME) {
      COMPOUND_NAME.tail = EmptyCompound;
      C_NAME_LIST$_2$.tail = COMPOUND_NAME.rev::$$ .tail;
      $$ .rev = C_NAME_LIST$_2$.rev;
VALUE_S ::= (VALUE) {$$ .rev = VALUE.abs::$$ .tail; }
    | (VALUE_S , VALUE) {
        VALUE_S2 .tail = VALUE.abs::$$ .tail;
        $$ .rev = VALUE_S2 .rev; }
    ;

VALUE ::= (EXPRESSION) { $$ .abs = ValueExpr(EXPRESSION.abs); }
    | (COMP_ASSOC) { $$ .abs = ValueCompAssoc(COMP_ASSOC.abs); }
    | (DISCRETE_WITH_RANGE) {
        $$ .abs = VALUE_DISCRETE_WITH_RANGE.abs;
    }
    ;

SELECTED_COMP ::= (NAME , ID) { $$ .abs = DotId(NAME.abs, Ident(ID)); }
    | (NAME , CHAR_LIT) {
        $$ .abs = DotUsedChar(NAME.abs, CHAR_LIT); }
    | (NAME , QUOTED_STRING) {
        $$ .abs = DotString(NAME.abs, QUOTED_STRING); }
    | (NAME , ALL) { $$ .abs = DotAll(NAME.abs); }
    ;

ATTRIBUTE_ID ::= (ID) { $$ .abs = AttribId(Ident(ID)); }
    | (DIGITS) { $$ .abs = AttribDigits(); }
    | (DELTA) { $$ .abs = AttribDelta(); }
    | (ACCESS) { $$ .abs = AttribAccess(); }
    ;

INTEGER_CS ::= (INTEGER) { $$ .abs = Integer(INTEGER); };

NUMERIC_LIT ::= (INTEGER_CS) { $$ .abs = IntLit(INTEGER_CS.abs); }
    | (REAL_CS) { $$ .abs = RealLit(REAL_CS); };

LITERAL ::= (NUMERIC_LIT) { $$ .abs = NumLit(NUMERIC_LIT.abs); }
    | (CHAR_LIT) { $$ .abs = UsedChar(CHAR_LIT); }
    | (NuL.L) { $$ .abs = NullLit(); }
    ;

AGGREGATE ::= 
    | (' COMP_ASSOC ') { $$ .abs = AggCompAssoc(COMP_ASSOC.abs); }
    | (' VALUE_S_2 ') { $$ .abs = AggValues2(VALUE_S_2.abs); }
    | (' EXPRESSION WITH VALUE_S ') {
        VALUE_S.tail = ValueNil;
        $$ .abs = AggExprValue(EXPRESSION.abs, VALUE_S.rev); }
    | (' EXPRESSION WITH NuL.L RECORD ') {
        $$ .abs = AggExprWithNull(EXPRESSION.abs); }
    ;
null RECORD 
{
$$\text{abs} = \text{AggExpNullRec}();
}

VALUE_S_2 ::= 
(VALUE ', VALUE) { $$\text{abs} = \text{Value2Pair}(\text{VALUE}_1\text{abs}, \text{VALUE}_2\text{abs}); }

| (VALUE_S_2 ', VALUE) 
| $$\text{abs} = \text{Value2List}(\text{VALUE}_S_2\text{abs}, \text{VALUE}\text{abs});
}

COMP_ASSOC ::= (CHOICE_S RIGHT SHAFT EXPRESSION) 
| CHOICE_S.tail = ChoiceNil; 
| $$\text{abs} = \text{CompAssoc}(\text{CHOICE}_S\text{rev}, \text{EXPRESSION}\text{abs});
}

LOGICAL {syn expression expOut; 
in expression expIn;};

SHORT_CIRCUIT {syn expression expOut; 
in expression expIn;};

EXPRESSION ::= (RELATION) { $$\text{abs} = \text{Relation}(\text{RELATION}\text{abs}); }

| (EXPRESSION LOGICAL) 
| LOGICAL.expIn = EXPRESSION$2.abs; 
| $$\text{abs} = \text{LOGICAL.expOut};
}

| (EXPRESSION SHORT_CIRCUIT) 
| SHORT_CIRCUIT.expIn = EXPRESSION$2.abs; 
| $$\text{abs} = \text{SHORT_CIRCUIT.expOut};
}

LOGICAL ::= (AND RELATION) 
| $$\text{expOut} = \text{And}($$\text{expIn},\text{RELATION}\text{abs});
}

| (OR RELATION) 
| $$\text{expOut} = \text{Or}($$\text{expIn},\text{RELATION}\text{abs});
}

| (XOR RELATION) 
| $$\text{expOut} = \text{Xor}($$\text{expIn},\text{RELATION}\text{abs});
}

SHORT_CIRCUIT ::= (AND THEN RELATION) 
| $$\text{expOut} = \text{AndThen}($$\text{expIn},\text{RELATION}\text{abs});
}

| (OR ELSE RELATION) 
| $$\text{expOut} = \text{OrElse}($$\text{expIn},\text{RELATION}\text{abs});
}

;
RELATIONAL {syn relation relOut;  
  inh simple_expression seIn;};

MEMBERSHIP {syn relation relOut;  
  inh simple_expression seIn;};

RELATION ::= (SIMPLE_EXPRESSION RELATIONAL) {  
  RELATIONAL.seIn = SIMPLE_EXPRESSION.abs;  
  $$'.abs = RELATIONAL.relOut;  
  }
  | (SIMPLE_EXPRESSION MEMBERSHIP) {  
  MEMBERSHIP.seIn = SIMPLE_EXPRESSION.abs;  
  $$'.abs = MEMBERSHIP.relOut;  
  }

RELAIONAL ::= () {$$.relOut = SimpleExpr($$.seIn);}
  | ('=' SIMPLE_EXPRESSION) {  
  $$'.relOut = Equal($$.seIn,SIMPLE_EXPRESSION.abs);  
  }
  | (NE SIMPLE_EXPRESSION) {  
  $$'.relOut = NotEqual($$.seIn,SIMPLE_EXPRESSION.abs);  
  }
  | (LT_LT SIMPLE_EXPRESSION) {  
  $$'.relOut = LessThan($$.seIn,SIMPLE_EXPRESSION.abs);  
  }
  | (LT_EQ SIMPLE_EXPRESSION) {  
  $$'.relOut = LessThanEq($$.seIn,SIMPLE_EXPRESSION.abs);  
  }
  | (GT_GT SIMPLE_EXPRESSION) {  
  $$'.relOut = GreaterThan($$.seIn,SIMPLE_EXPRESSION.abs);  
  }
  | (GE SIMPLE_EXPRESSION) {  
  $$'.relOut = GreaterThanEq($$.seIn,SIMPLE_EXPRESSION.abs);  
  }

MEMBERSHIP ::= (IN RANGE) {  
  $$'.relOut = RangeMember($$.seIn,In,RANGE.abs);  
  }
  | (NOT IN RANGE) {  
  $$'.relOut = RangeMember($$.seIn,NotIn,RANGE.abs);  
  }
  | (IN NAME) {  
  $$'.relOut = NameMember($$.seIn,In,NAME.abs);  
  }
  | (NOT IN NAME) {  
  $$'.relOut = NameMember($$.seIn,NotIn,NAME.abs);  
  }

ADDDLING {syn simple_expression seOut;  
  inh simple_expression seIn;  
  }

UNARY {syn simple_expression abs;};

SIMPLE_EXPRESSION ::= (UNARY) {$$.abs = UNARY.abs;}
  | ('-' TERM) {$$.abs = Term(Minus(),TERM.abs);}
  | (SIMPLE_EXPRESSION ADDING) {  
  ADDING.seIn = SIMPLE_EXPRESSION$2.abs;  
  $$'.abs = ADDING.seOut;  
  }
UNARY ::= (TERM) { $$\.abs = Term(UnaryNull,TERM\.abs); }
  | ('+' TERM) { $$\.abs = Term(Plus,TERM\.abs); }
  | ('-' TERM) { $$\.abs = Term(Minus,TERM\.abs); }

; 

ADDING ::= ('+' TERM) {
  $$\.seOut = Addition($$.seln,TERM\.abs);
}
  | ('-' TERM) {
  $$\.seOut = Subtraction($$.seln,TERM\.abs);
}
  | ('&' TERM) {
  $$\.seOut = Concat($$.seln,TERM\.abs);
}

; 

MULTIPLYING {syn term termOut;
        inh term termln;};

TERM ::= (FACTOR) { $$\.abs = Factor(FACTOR\.abs); }
  | (TERM MULTIPLYING) {
    MULTIPLYING\.termln = TERM$2\.abs;
    $$\.abs = MULTIPLYING\.termOut;
  }

; 

MULTIPLYING ::= ('*' FACTOR) {$$.termOut = Mult($$.termln,FACTOR\.abs);}
  | ('/' FACTOR) {$$.termOut = Divide($$.termln,FACTOR\.abs);}
  | (MOD FACTOR) {$$.termOut = Mod($$.termln,FACTOR\.abs);}
  | (REM FACTOR) {$$.termOut = Rem($$.termln,FACTOR\.abs);}

; 

FACTOR ::= (PRIMARY) { $$\.abs = Primary(PRIMARY\.abs); }
  | (NOT PRIMARY) { $$\.abs = NotPrimary(PRIMARY\.abs); }
  | (ABS PRIMARY) { $$\.abs = AbsPrimary(PRIMARY\.abs); }
  | (PRIMARY EXPON PRIMARY prec EXPON) {
    $$\.abs = Expon(PRIMARY$1\.abs, PRIMARY$1\.abs); }

; 

PRIMARY ::= (LITERAL) { $$\.abs = Literal(LITERAL\.abs); }
  | (NAME) { $$\.abs = PrimaryName(NAME\.abs); }
  | (ALLOCATOR) { $$\.abs = Allocator(ALLOCATOR\.abs); }
  | (QUALIFIED) { $$\.abs = Qualified(QUALIFIED\.abs); }
  | ("(" EXPRESSION ")") { $$\.abs = Paren(EXPRESSION\.abs); }
  | (AGGREGATE) { $$\.abs = PrimaryAgg(AGGREGATE\.abs); }

; 

QUALIFIED ::= (NAME TIC AGGREGATE) {
  $$\.abs = NameTicAgg(NAME\.abs, AGGREGATE\.abs); }
  | (NAME TIC "(" EXPRESSION ")") {
    $$\.abs = NameTicExpr(NAME\.abs, EXPRESSION\.abs); }

137
ALLOCATOR ::= (NEW NAME) { $$ \text{abs} = \text{newName} (\text{NAME} \text{abs}); } \\
| (NEW QUALIFIED) { $$ \text{abs} = \text{NewQualified} (\text{QUALIFIED} \text{abs}); } \\
| ;

SUBPROG_DECL ::= 
(GENERIC GENERIC_FORMAL_PART SUBPROG_SPEC ';'; PSDL_MET_OPT) { 
    GENERIC_FORMAL_PART.tail = GenericNil;
    $$ \text{abs} = \text{SubprogSpec} (\text{GenericHdr} (\text{GENERIC_FORMAL_PART} \text{rev}),
    \text{SUBPROG_SPEC} \text{abs}, \text{PSDL\_MET\_OPT} \text{abs});
} \\
| (SUBPROG_SPEC ';'; PSDL\_MET\_OPT) { 
    $$ \text{abs} = \text{SubprogSpec} (\text{GenericHdrNil},
    \text{SUBPROG\_SPEC} \text{abs}, \text{PSDL\_MET\_OPT} \text{abs});
} \\
| (SUBPROG\_SPEC IS GENERIC_INST ' '; PSDL\_COMMENT PSDL\_MET\_OPT) { 
    $$ \text{abs} = \text{GenericSubprogInst} (\text{SUBPROG\_SPEC} \text{abs},
    \text{GENERIC\_INST} \text{abs}, \text{PSDL\_MET\_OPT} \text{abs}); }
| (SUBPROG\_SPEC IS ABSTRACT ' '; PSDL\_MET\_OPT) { 
    $$ \text{abs} = \text{AbstractSubprogSpec} (\text{SUBPROG\_SPEC} \text{abs},
    \text{PSDL\_MET\_OPT} \text{abs}); }

PSDL\_MET\_OPT ::= () { $$ \text{abs} = \text{MetNull}(); }
| (PSDL\_COMMENT MAXIMUM EXECUTION TIME INTEGER CS USEC) { 
    $$ \text{abs} = \text{MetUsec} (\text{INTEGER-CS} \text{abs}); }
| (PSDL\_COMMENT MAXIMUM EXECUTION TIME INTEGER CS MS) { 
    $$ \text{abs} = \text{MetMs} (\text{INTEGER-CS} \text{abs}); }
| (PSDL\_COMMENT MAXIMUM EXECUTION TIME INTEGER CS SEC) { 
    $$ \text{abs} = \text{MetSec} (\text{INTEGER-CS} \text{abs}); }
| (PSDL\_COMMENT MAXIMUM EXECUTION TIME INTEGER CS MIN) { 
    $$ \text{abs} = \text{MetMin} (\text{INTEGER-CS} \text{abs}); }
| (PSDL\_COMMENT MAXIMUM EXECUTION TIME INTEGER CS HRS) { 
    $$ \text{abs} = \text{MetHrs} (\text{INTEGER-CS} \text{abs}); }

SUBPROG::= (PROCEDURE COMPOUND_NAME FORMAL\_PART\_OPT) { 
    COMPOUND_NAME.tail = EmptyCompound;
    $$ \text{abs} = \text{SubProgProc} (\text{COMPOUND\_NAME} \text{rev}, \text{FORMAL\_PART\_OPT} \text{abs}),
} \\
| (FUNCTION DESIGNATOR FORMAL\_PART\_OPT RETURN NAME) { 
    $$ \text{abs} = \text{SubProgFuncReturn} (\text{DESIGNATOR} \text{abs},
    \text{FORMAL\_PART\_OPT} \text{abs}, \text{NAME} \text{abs}); }
} \\
| (FUNCTION DESIGNATOR) { 
    $$ \text{abs} = \text{SubProgFunc} (\text{DESIGNATOR} \text{abs});
} /* for generic inst and generic rename */

DESIGNATOR ::= (COMPOUND\_NAME) { 
    COMPOUND\_NAME.tail = EmptyCompound;

138
$$.abs = \text{DesigCompound}(\text{COMPOUND\_NAME}.rev);$

| (\text{QUOTED\_STRING}) | $$.abs = \text{DesigString}(\text{QUOTED\_STRING}); |

\begin{align*}
\text{FORMAL\_PART\_OPT} & := () \quad \{ \text{FORMAL\_PART\_OPTNull();} \\
| (\text{PARAM\_S } \\ ') & \{ \\
\text{PARAM\_S.tail} & = \text{ParamNil}; \\
$$.abs & = \text{FormalPart}(\text{PARAM\_S}.rev); \\
\}
\end{align*}

\begin{align*}
\text{PARAM\_S} & := (\text{PARAM}) \quad \{$$.rev = \text{PARAM.abs}::$$\text{tail};} \\
| (\text{PARAM\_S } \\ ';') & \{ \\
\text{PARAM\_S.tail} & = \text{PARAM.abs}::$$\text{tail}; \\
$$.rev & = \text{PARAM\_S}.rev; \\
\}
\end{align*}

\begin{align*}
\text{PARAM} & := (\text{DEF\_ID\_S } \\ ';') \quad \text{MODE MARK INIT\_OPT} \quad \{ \\
\text{DEF\_ID\_S.tail} & = \text{DefIdNil}; \\
$$.abs & = \text{ParamId}(\text{DEF\_ID\_S}.rev, \text{MODE.abs}, \text{MARK.abs}, \text{INIT\_OPT.abs}); \\
\}
\end{align*}

\begin{align*}
\text{MODE} & := () \quad \{ \text{MODENull();} \\
| (\text{IN}) & \{ \text{INMode();} \\
| (\text{OUT}) & \{ \text{OUTMode();} \\
| (\text{IN OUT}) & \{ \text{INOutMode();} \\
| (\text{ACCESS}) & \{ \text{AccessMode();} \\
\}
\end{align*}

\begin{align*}
\text{PKG\_DECL} & := \quad \{ \text{SPEC(\text{GENERIC\_FORMAL\_PART PKG\_SPEC } \\ ';');} \\
\text{GENERIC\_FORMAL\_PART.tail} & = \text{GenericNil}; \\
$$.abs & = \text{PkgSpec}(\text{GenericHdr(\text{GENERIC\_FORMAL\_PART.rev),} \\
\text{PKG\_SPEC.abs});} \\
| (\text{PACKAGE COMPOUND\_NAME IS GENERIC\_INST } \\ ';') & \{ \\
\text{COMPOUND\_NAME.tail} & = \text{EmptyCompound}; \\
$$.abs & = \text{GenPkglnst(\text{COMPOUND\_NAME.rev, GENERIC\_INST.abs);} \\
\}
\end{align*}

\begin{align*}
| (\text{PKG\_SPEC } \\ ';') & \{ \\
$$.abs & = \text{PkgSpec(\text{GenericHdrNil, PKG\_SPEC.abs);} \\
\}
\end{align*}

\begin{align*}
\text{PKG\_SPEC} & := \quad \{ \text{SPEC(\text{PACKAGE COMPOUND\_NAME IS DECL\_ITEM\_S PRIVATE\_PART END} \\
\text{C\_ID\_OPT)} \\
\text{COMPOUND\_NAME.tail} & = \text{EmptyCompound;} \\
\text{DECL\_ITEM\_S.tail} & = \text{DecllstNil}; \\
$$.abs & = \text{Package(\text{COMPOUND\_NAME.rev, DECL\_ITEM\_S.rev,} \\
\text{PRIVATE\_PART.abs);} \\
\)
\end{align*}

139
PRIVATE_PART ::= () {$$.abs = PrivatePartNull();}
| (PRIVATE DECL_ITEM_S) {
   DECL_ITEM_S.tail = DeclListNil;
   $$abs = Private(DECL_ITEM_S.rev);}
|;

PRIVATE_TYPE ::= (TAGGED_OPT LIMITED_OPT PRIVATE) {
   $$abs = PrivateType(TAGGED_OPT.abs, LIMITED_OPT.abs);}
|;

LIMITED_OPT ::= () {$$.abs = LimitedOptNull();}
| (LIMITED) {$$.abs = Limited();}
|;

USE_CLAUSE ::= (USE NAME_S ';') {
   NAME_S.tail = NameNil;
   $$abs = Use(NAME_S.rev);}
| (USE TYPE NAME_S ';') {
   NAME_S.tail = NameNil;
   $$abs = UseType(NAME_S.rev);}
|;

NAME_S ::= (NAME) {$$.rev = NAME.abs; $$tail;}
| (NAME_S ',' NAME) {
   NAME_S.tail = NAME.abs; $$tail;
   $$rev = NAME_S$2.rev;}
|;

RENAMEDECL ::= 
   (DEF_ID_S ':' OBJECT_QUALIFIER_OPT SUBTYPE_IND RENAMES ';') {
   DEF_ID_S.tail = DefIdNil;
   $$abs = RenameDeclSub(DEF_ID_S.rev, OBJECT_QUALIFIER_OPT.abs, SUBTYPE_IND.abs, RENAMES.abs);
   }
| (DEF_ID_S ':' EXCEPTION RENAMES ';') {
   DEF_ID_S.tail = DefIdNil;
   $$abs = RenameExc(DEF_ID_S.rev, RENAMES.abs);
   }
| (RENAME_UNIT) {$$.abs = RenameUnitDecl(RENAME_UNIT.abs);}
|;

RENAME_UNIT ::= (GENERIC GENERIC_FORMAL_PART
   PACKAGE COMPOUND_NAME RENAMES ';') {
   GENERIC_FORMAL_PART.tail = GenericNil;
   COMPOUND_NAME.tail = EmptyCompound;
$$ \text{abs} = \text{RenamePkg}($\text{GenericHdr(GENERIC\_FORMAL\_PART.rev)}, \text{COMPOUND\_NAME.rev}, \text{RENNAMES.abs}) \};
\}

| \text{(PACKAGE COMPOUND\_NAME RENAMES ") \{}}
| \text{COMPOUND\_NAME.tail = EmptyCompound;}
| \text{\$$.abs = RenamePkg($\text{GenericHdrNil, COMPOUND\_NAME.rev, RENAMES.abs}) \};}

| \text{(GENERIC GENERIC\_FORMAL\_PART SUBPROG\_SPEC RENAMES ") \{}}
| \text{GENERIC\_FORMAL\_PART.tail = GenericNil;}
| \text{\$$.abs = RenameSubprog($\text{GenericHdr(GENERIC\_FORMAL\_PART.rev)}, \text{SUBPROG\_SPEC.abs}, \text{RENNAMES.abs}) \};}

| \text{(SUBPROG\_SPEC RENAMES ") \{}}
| \text{\$$.abs = RenameSubprog($\text{GenericHdrNil, SUBPROG\_SPEC.abs, RENAMES.abs}) \};}

\text{RENNAMES ::= } \text{(ReNAMES NAME) \{ \$$.abs = ReNames(NAME.abs) \};}

\text{TASK\_SPEC ::=}
\text{(TASK ID TASK\_DEF) \{ \$$.abs = SimpleTask($\text{Ident(ID)}, \text{TASK\_DEF.abs}) \};}
\text{(TASK TYPE ID DISCRIM\_PART\_OPT TASK\_DEF) \{}
\text{\$$.abs = TaskType($\text{Ident(ID)}, \text{DISCRIM\_PART\_OPT.abs, TASK\_DEF.abs}) \};}

\text{TASK\_DEF ::= () \{ \$$.abs = TaskDefNull();}
\text{\text{\( (\text{IS ENTRY\_DECL\_S REP\_SPEC\_S TASK\_PRIVATE\_OPT END ID\_OPT) \{}
\text{\$$.abs = TaskDef($\text{ENTRY\_DECL\_S.abs, REP\_SPEC\_S.abs, TASK\_PRIVATE\_OPT.abs}) \};}

\text{ID\_OPT ::= ()}
\text{\( (ID)\}

\text{TASK\_PRIVATE\_OPT ::= () \{ \$$.abs = TaskPvtOptNull();}
\text{\( (PRIVATE ENTRY\_DECL\_S REP\_SPEC\_S) \{}
\text{\$$.abs = TaskPvtOpt($\text{ENTRY\_DECL\_S.abs, REP\_SPEC\_S.abs}) \};}

\text{PROT\_SPEC ::=}
\text{(PROTECTED ID PROT\_DEF) \{ \$$.abs = Prot($\text{Ident(ID), PROT\_DEF.abs}) \};}
\text{(PROTECTED TYPE ID DISCRIM\_PART\_OPT PROT\_DEF) \{}
\text{\$$.abs = ProtType($\text{Ident(ID), DISCRIM\_PART\_OPT.abs, PROT\_DEF.abs}) \};}

\text{PROT\_DEF ::= (IS PROT\_OP\_DECL\_S PROT\_PRIVATE\_OPT END ID\_OPT) \{}
\text{\( \text{PROT\_OP\_DECL\_S.tail = ProtOptDeclListNil; \$$.abs = ProtDef($\text{PROT\_OP\_DECL\_S.rev, PROT\_PRIVATE\_OPT.abs}) \};}

141
PROT_PRIVATE_OPT := () { $$ abs = ProtOptNull(); }
| (PRIVATE PROT_ELEMDECL_S) { 
    PROT_ELEMDECL_S.tail = ProtElemDeclNil;
    $$ abs = ProtOptDecl(PRIVATE PROT_ELEMDECL_S.rev); }

PROT_OPDECL_S := () { $$ rev = $$ tail; }
| (PROT_OPDECL_S PROT_OPDECL) { 
    PROT_OPDECL_S.$$2.tail = PROT_OPDECL.abs.$$tail;
    $$ rev = PROT_OPDECL_S.$$2.rev; }

PROT_OPDECL := (ENTRY_DECL) { $$ abs = EntryDecl(ENTRY_DECL.abs); }
| (SUBPROG_SPEC ;') { $$ abs = ProtOptSubprog(SUBPROG_SPEC.abs); }
| (REP_SPEC) { $$ abs = RepSpec(REP_SPEC.abs); }
| (PRAGMA) { $$ abs = ProtOptPragma(PRAGMA.abs); }

PROT_ELEMDECL_S := () { $$ abs = ProtOptDecl(PROT_ELEMDECL.abs); }
| (PROT_ELEMDECL_S PROT_ELEMDECL) { 
    PROT_ELEMDECL_S.$$2.tail = PROT_ELEMDECL.abs.$$tail;
    $$ rev = PROT_ELEMDECL_S.$$2.rev; }

PROT_ELEMDECL :=
    (ENTRY_DECL) { $$ abs = ProtOptDecl(PROT_ELEMDECL.abs); }
| (COMP_DECL) { $$ abs = ProtElemCompDecl(COMP_DECL.abs); }

ENTRYDECL S := (PRAGMA S) {
    PRAGMA_S.tail = PragmasNil;
    $$ abs = EntryDeclPragma(PRAMA_S.rev); }
| (ENTRYDECL_S ENTRYDECL PRAGMA S) {
    PRAGMA_S.tail = PragmasNil;
    $$ abs = EntryDeclPragmaList(ENTRYDECL_S.$$2.abs,
    ENTRYDECL.abs, PRAGMA_S.rev); }

ENTRYDECL := (ENTRY ID FORMAL_PART OPT ';') {
    $$ abs = EntryDeclId(Ident(ID), FORMAL_PART_OPT.abs); }
| (ENTRY ID (' DISCRETE_RANGE ') FORMAL_PART OPT ';') {
    $$ abs = EntryRange(Ident(ID), DISCRETE_RANGE.abs,
    FORMAL_PART_OPT.abs); }

REP_SPEC_S := () { $$ abs = RepSpecNull(); }
| (REP_SPEC_S REP_SPEC PRAGMA S) {
    PRAGMA_S.tail = PragmasNil;
    $$ abs = RepSpecList(REP_SPEC_S.$$2.abs, REP_SPEC.abs,
    PRAGMA_S.rev); }
COMP_UNIT ::=  
  (CONTEXT_SPEC_OPT PRIVATE_OPT PKG_DECL PRAGMA_S) {  
    PRAGMA_S.tail = PragmasNil;  
    $$abs = CompUnit(CONTEXT_SPEC_OPT.abs, PRIVATE_OPT.abs,  
    PKG_DECL.abs, PRAGMA_S.rev);  
  }

PRIVATE_OPT ::= () { $$abs = PrivateOptNull(); }  
  | (PRIVATE) { $$abs = PrivateOpt(); }

CONTEXT_SPEC_OPT ::= () { $$abs = ContextSpecNull(); }  
  | (CONTEXT_SPEC) { $$abs = ContextSpec(CONTEXT_SPEC.abs); }

CONTEXT_SPEC ::=  
  (CONTEXT_SPEC_OPT WITH_CLAUSE USE_CLAUSE_OPT) {  
    USE_CLAUSE_OPT.tail = UseClauseOptNil;  
    $$abs = ContextWithUse(CONTEXT_SPEC_OPT.abs,  
    WITH_CLAUSE.abs, USE_CLAUSE_OPT.rev);  
  }  
  | (CONTEXT_SPEC PRAGMA) {  
    $$abs = ContextPragma(CONTEXT_SPEC.abs, PRAGMA.abs);  
  }

WITH_CLAUSE ::= (WITH C_NAME_LIST ';') {  
  C_NAME_LIST.tail = CompoundNameNil;  
  $$abs = WithClause(C_NAME_LIST.rev);  
}

USE_CLAUSE_OPT ::= () { $$rev = $$tail; }  
  | (USE_CLAUSE_OPT USE_CLAUSE) {  
    USE_CLAUSE_OPT.tail = USE_CLAUSE.abs.$$tail;  
    $$rev = USE_CLAUSE_OPT.$$.rev;  
  }

BODY_STUB ::= (TASK BODY ID IS SEPARATE ';') { $$abs = TaskStub(Ident(ID)); }  
  | (PACKAGE BODY COMPOUND_NAME IS SEPARATE ';') {  
    COMPOUND_NAME.tail = EmptyCompound;  
    $$abs = PkgStub(COMPOUND_NAME.rev);  
  }  
  | (SUBPROG_SPEC IS SEPARATE ';') {  
    $$abs = SubprogStub(SUBPROG_SPEC.abs);  
    $$abs = ProtStub(Ident(ID));  
  }

GENERIC_FORMAL_PART ::= () { $$rev = $$tail; }  
  | (GENERIC_FORMAL_PART GENERIC_FORMAL) {  
    GENERIC_FORMAL_PART.tail = GENERIC_FORMAL.abs.$$tail;  
    $$rev = GENERIC_FORMAL_PART.$$.rev;  
  }

143
GENERIC_FORMAL ::= (PARAM ';') { $$abs = GenParm(PARAM.abs); }

| (TYPE ID GENERIC_DISCRIM_PART_OPT IS GENERIC_TYPE_DEF ';') { $$abs = GenTypeParm(Ident(ID), GENERIC_DISCRIM_PART_OPT.abs, GENERIC_TYPE_DEF.abs); }
| (WITH PROCEDURE ID FORMAL_PART_OPT SUBP_DEFAULT ';') { $$abs = GenProcParm(Ident(ID), FORMAL_PART_OPT.abs, SUBP_DEFAULT.abs); }
| (WITH FUNCTION DESIGNATOR FORMAL_PART_OPT RETURN NAME SUBP_DEFAULT ';') { $$abs = GenFuncParm(DESIGNATOR.abs, FORMAL_PART_OPT.abs, NAME.abs, SUBP_DEFAULT.abs); }
| (WITH PACKAGE ID IS NEW NAME ' ( BOX ') ';') { $$abs = GenPkgParmBox(Ident(ID), NAME.abs); }
| (WITH PACKAGE ID IS NEW NAME ';') { $$abs = GenPkgParm(Ident(ID), NAME.abs); }
| (USE_CLAUSE) { $$abs = GenUseParm(USE_CLAUSE.abs); }

GENERIC_DISCRIM_PART_OPT ::= () { $$abs = GenDiscOptNull(); }
| (" DISCRIM_SPEC_S") { DISCRIM_SPEC_S.tail = DiscrimSpecNil; $$abs = GenDisc(DISCRIM_SPEC_S.rev); }
| (" BOX ") { $$abs = GenBox(); }

SUBP_DEFAULT ::= () { $$abs = SubpDefaultNull(); }
| (IS NAME) { $$abs = SubpDefName(NAME.abs); }
| (IS BOX) { $$abs = SubpDefBox(); }

GENERIC_TYPE_DEF ::= (" BOX ") { $$abs = GenTypeBox(); }
| (RANGE BOX) { $$abs = GenTypeRangeBox(); }
| (MOD BOX) { $$abs = GenTypeModBox(); }
| (DELTA BOX) { $$abs = GenTypeDeltaBox(); }
| (DELTA BOX DIGITS BOX) { $$abs = GenTypeDeltaDigBox(); }
| (DIGITS BOX) { $$abs = GenTypeDigBox(); }
| (ARRAY_TYPE) { $$abs = GenTypeArray(ARRAY_TYPE.abs); }
| (ACCESS_TYPE) { $$abs = GenTypeAccess(ACCESS_TYPE.abs); }
| (PRIVATE_TYPE) { $$abs = GenTypePriv(PRIVATE_TYPE.abs); }
| (GENERIC_DERIVED_TYPE) { $$abs = GenTypeDerived(GENERIC_DERIVED_TYPE.abs); }

GENERIC_DERIVED_TYPE ::= (NEW SUBTYPE_IND) { $$abs = GenDerivedSubt(SUBTYPE_IND.abs); }
| (NEW SUBTYPE_IND WITH PRIVATE) { $$abs = GenDerivedSubtPriv(SUBTYPE_IND.abs); }
| (ABSTRACT NEW SUBTYPE_IND WITH PRIVATE) { $$abs = GenDerivedAbst(SUBTYPE_IND.abs); }

144
GENERIC_INST ::= (NEW NAME) { $$abs = GenInst(NAME.abs); }

REP_SPEC ::= (FOR MARK USE EXPRESSION ';' ) {
  $$abs = AttribDef(MARK.abs, EXPRESSION.abs); }
  | (FOR MARK USE RECORD ALIGN_OPT COMP_LOC_S END RECORD ';' ) {
    $$abs = RecordTypeSpec(MARK.abs, ALIGN_OPT.abs,
                             COMP_LOC_S.abs); }
  | (FOR MARK USE AT EXPRESSION ';' ) {
    $$abs = AddressSpec(MARK.abs, EXPRESSION.abs); }
};

ALIGN_OPT ::= () { $$abs = AlignOptNull(); }
  | (AT MOD EXPRESSION ';' ) { $$abs = AlignOpt(EXPRSNSIO.abs); }
};

COMP_LOC_S ::= () { $$abs = CompLocNull(); }
  | (COMP_LOC_S MARK AT EXPRESSION RANGE RANGE ';' ) {
    $$abs = CompLocList(COMP_LOC_S$2.abs, MARK.abs,
                         EXPRESSION.abs, RANGE.abs); }
};
APPENDIX E. SSL SOURCE CODE: TRANSFORMATIONS

The source code below is used to specify the allowable transformations for Ada 95 productions. Transformation declarations specify the manner in which a user may manipulate the abstract syntax tree while using the translator in the interactive mode.

```plaintext
/* File: transforms.ada9x.ssl */
/* Date: 3 March, 1995 */
/* Author: Chris Eagle */
/* System: Sun SPARCstation */
/* Description: This file contains the transform rules which specify the ways in which users of the syntax directed may transform the syntax tree of an Ada 9x package. */

transform compilation
  on "PKG DECL"
    <compilation>: Compilation(<pragma_s>,
                         CUList(<comp_unit>, [comp_unit_list]))

transform comp_unit_list
  on "COMP UNIT"
    <comp_unit_list>: CUList(<comp_unit>, [comp_unit_list])

transform pragma
  on "ID"
    <pragma>: PragmalId(<identifier>),
  on "LIST"
    <pragma>: PragmalSimple(<identifier>, <pragma_arg_s>)

transform pragma_arg
  on "EXPR"
    <pragma_arg>: PragmaExp(<expression>),
  on "NAMED"
    <pragma_arg>: PragmaNameExp(<identifier>, <expression>)

transform decl
  on "OBJECT"
    <decl>: ObjDecl(<def_ids>,<object_qualifier_opt>,
                  <object_subtype_def>, <init_opt>),
  on "NUMERIC"
    <decl>: NumDecl(<def_id_s>, <expression>),
  on "TYPE"
    <decl>: TypeDecl(<identifier>, <discrim_part_opt>,
                   <type_completion>),
  on "SUBTYPE"
    <decl>: SubTypeDecl(<identifier>, <subtype_ind>),
  on "SUBPROG"
    <decl>: SubProgDecl(<subprog_decl>),
  on "PKG"
    <decl>: PkgDecl(<pkg_decl>),
  on "TASK"
    <decl>: TaskDecl(<task_spec>)
```

on "PROTECTED" <decl>: ProtDecl(<prot_spec>),
on "EXCEPTION" <decl>: ExcDecl(<def_id_s>),
on "RENAMES" <decl>: RenameDecl(<rename_deci>),
on "BODY_STUB" <decl>: BodyStubDecl(<body_stub>)

transform object_subtype_def
on "SUBTYPE" <object_subtype_def>: SubtypeInd(<subtype_ind>),
on "ARRAY" <object_subtype_def>: ArrayType(<array_type>)

transform type_def
on "ENUM" <type_def>: EnumTypeDef(<enum_id_s>),
on "INT" <type_def>: IntTypeDef(<integer_type>),
on "REAL" <type_def>: RealTypeDef(<real_type>),
on "ARRAY" <type_def>: ArrayTypeDef(<array_type>),
on "RECORD" <type_def>: RecordType(<tagged_opt>,
     <limited_opt>, <record_def>),
on "ACCESS" <type_def>: AccessTypeDef(<access_type>),
on "DERIVED" <type_def>: DerivedTypeDef(<derived_type>),
on "PRIVATE" <type_def>: PrivateTypeDef(<private_type>)

transform subtype_ind
on "CONSTRAINT" <subtype_ind>: SubtypeIndConstraint(<name>,
     <constraint>),
on "NAME" <subtype_ind>: SubTypelndName(<name>)

transform constraint
on "RANGE" <constraint>: RangeConstraint(<range_constraint>),
on "DIGITS" <constraint>: DecDigConstraint(<expression>,
     <range_constr_opt>)

transform derived_type
on "NEW" <derived_type>: NewDerivedType(<subtype_ind>),
on "NEW_PRIVATE" <derived_type>: NewDerivedWithPrivate(<subtype_ind>),
on "NEW_RECORD" <derived_type>: NewDerivedWithRecord(<subtype_ind>,
     <record_def>),
on "ABSTRACT_PRIVATE" <derived_type>
     : AbsNewDerivedWithPrivate(<subtype_ind>),
on "ABSTRACT_RECORD" <derived_type>: AbsNewDerivedWithRecord(<subtype_ind>,
     <record_def>)

transform range
on "..." <range>: SimpleRange(<simple_expression>,
     <simple_expression>),
on "RANGE"<range>: NameTicRange(<name>),
on "RANGE(EXPR)"<range>: NameTicRangeExp(<name>, <expression>)
;

transform enum_id
  on "ID"<enum_id>: Id(<identifier>),
on "CHAR_LIT"<enum_id>: CharLit(<CHAR_LIT>)
;

transform integer_type
  on "RANGE"<integer_type>: RangeSpec(<range_spec>),
on "MOD"<integer_type>: ModExpr(<expression>)
;

transform real_type
  on "FLOAT"<real_type>: FloatType(<expression>, <range_spec_opt>),
on "FIXED"<real_type>: FixedType(<fixed_type>)
;

transform fixed_type
  on "DELTA"<fixed_type>: FixedDelta(<expression>, <range_spec>),
on "DLETA_DIGITS"<fixed_type>: FixedDeltaDigits(<expression>,
    <expression>, <range_spec_opt>)
;

transform array_type
  on "UNCONSTRAINED"<array_type>: UnconstrArray(<index_s>,
    <component_subtype_def>),
on "CONSTRAINED"<array_type>: ConstrArray(<iter_discrete_range_s>,
    <component_subtype_def>)
;

transform discrete_range
  on "NAME"<discrete_range>: DiscRangeName(<name>,
    <range_constr_opt>),
on "RANGE"<discrete_range>: DiscRangeRange(<range>)
;

transform record_def
  on "RECORD"<record_def>: Record(<pragma_s>, <comp_list>),
on "NULL"<record_def>: NullRecord
;

transform comp_list
  on "VARIANT"<comp_list>: CompListWithVariant(<comp_decl_s>,
    <variant_part_opt>),
on "PRAGMA"<comp_list>: CompListWithPragma(<variant_part>,
    <pragma_s>),
on "NULL"<comp_list>: NullWithPragma(<pragma_s>)
;

149
transform variant_part_opt
  on "PRAGMA" <variant_part_opt>: VariantPartOptPragma(<pragma_s>),
  on "VARIANT" <variant_part_opt>: VariantPartOpt(<pragma_s>,
  on "" <variant_part>, <pragma_s>)
;
transform choice
  on "EXPR" <choice>: ChoiceExpr(<expression>),
  on "RANGE" <choice>: ChoiceRange(<discrete_with_range>),
  on "OTHERS" <choice>: ChoiceOthers
;
transform access_type
  on "SUBTYPE" <access_type>: AccessSubtype(<subtype_ind>),
  on "CONST_SUBTYPE" <access_type>: AccessConstSubtype(<subtype_ind>),
  on "All_SUBTYPE" <access_type>: AccessAllSubtype(<subtype_ind>),
  on "PROCEDURE" <access_type>: AccessProcedure(<prot_opt>,
  on "FUNCTION" <access_type>: AccessFunction(<prot_opt>,
  <formal_part_opt>, <mark>)
;
transform decl_item
  on "DECL" <decl_item>: Decl(<decl>),
  on "USE_CLAUSE" <decl_item>: UseClauseDecl(<use_clause>),
  on "REP_SPEC" <decl_item>: DeclRepSpec(<rep_spec>),
  on "PRAGMA" <decl_item>: DeclPragma(<pragma>)
;
transform name
  on "SIMPLE" <name>: SimpleName(<identifier>),
  on "INDEX" <name>: IndexComp(<name>, <value_s>),
  on "SELECTED" <name>: SelectedComp(<selected_comp>),
  on "ATTRIBUTE" <name>: Attribute(<name>, <attribute_id>),
  on "OPERATOR" <name>: OperatorSymbol(<QUOTED_STRING>)
;
transform mark
  on "MARK" <mark>: Mark(<identifier>, <marklist>)
;
transform ticdot
  on "ATTR" <ticdot>: TicOpt(<attribute_id>),
  on "ID" <ticdot>: DotOpt(<identifier>)
;
transform compound_name
  on "ID" <compound_name>: DotCompound(<identifier>,
  <compound_name>)
;
150
transform value
  on "EXPR" <value>: ValueExpr(<expression>),
on "COMP_ASSOC" <value>: ValueCompAssoc(<comp_assoc>),
on "DISC_WITH_RANGE" <value>: ValueDiscWithRange(<discrete_with_range>)
;
transform selected_comp
  on "DOT_ID" <selected_comp>: DotId(<name>, <identifier>),
on "DOT_CHAR" <selected_comp>: DotUsedChar(<name>, <CHAR_LIT>),
on "DOT_STRING" <selected_comp>: DotString(<name>,
    <QUOTED_STRING>),
on "DOT_ALL" <selected_comp>: DotAll(<name>)
;
transform attribute_id
  on "ID" <attribute_id>: AttribId(<identifier>),
on "DIGITS" <attribute_id>: AttribDigits,
on "DELTA" <attribute_id>: AttribDelta,
on "ACCESS" <attribute_id>: AttribAccess
;
transform literal
  on "NUMERIC" <literal>: NumLit(<numeric_lit>),
on "CHAR" <literal>: UsedChar(<CHAR_LIT>),
on "NIL" <literal>: NilLit
;
transform aggregate
  on "COMP_ASSOC" <aggregate>: AggCompAssoc(<comp_assoc>),
on "VALUES" <aggregate>: AggValues2(<value_s 2>),
on "EXPR_VALUE" <aggregate>: AggExprValue(<expression>, <value_s>),
on "EXPR" <aggregate>: AggExprWithNull(<expression>),
on "EXPR_NULL_REC" <aggregate>: AggExpNullRec
;
transform expression
  on "RELATION" <expression>: Relation(<relation>),
on "AND" <expression>: And(<expression>, <relation>),
on "OR" <expression>: Or(<expression>, <relation>),
on "XOR" <expression>: Xor(<expression>, <relation>),
on "AND_THEN" <expression>: AndThen(<expression>, <relation>),
on "ORELSE" <expression>: OrElse(<expression>, <relation>)
;
transform relation
  on "SIMPLE" <relation>: SimpleExpr(<simple_expression>),
on "=" <relation>: Equal(<simple_expression>,
    <simple_expression>),
on "/= " <relation>: NotEqual(<simple_expression>,
    <simple_expression>),
on "<" <relation>: LessThan(<simple_expression>,

151
on "<=" <simple_expression>,
    <relation>: LessThanEq(<simple_expression>,
        <simple_expression>),
on ">=" <simple_expression>,
    <relation>: GreaterThanEq(<simple_expression>,
        <simple_expression>),
on "RANGE_MBR" <simple_expression>,
    <relation>: RangeMember(<simple_expression>,
        <membership>, <range>),
on "NAME_MBR" <simple_expression>,
    <relation>: NameMember(<simple_expression>,
        <membership>, <name>)
;

transform membership
    on "IN"       <membership>: In,
    on "NOT_IN"   <membership>: NotIn
;

transform simple_expression
    on "TERM"     <simple_expression>: Term(<unary>, <term>),
    on "+"       <simple_expression>: Addition(<simple_expression>,
        <term>),
    on "-"       <simple_expression>: Subtraction(<simple_expression>,
        <term>),
    on "&"       <simple_expression>: Concat(<simple_expression>,
        <term>)
;

transform unary
    on "+"       <unary>: Plus,
    on "-"       <unary>: Minus
;

transform term
    on "FACTOR"   <term>: Factor(<factor>),
    on "*"       <term>: Mult(<term>, <factor>),
    on "/"       <term>: Divide(<term>, <factor>),
    on "MOD"     <term>: Mod(<term>, <factor>),
    on "REM"     <term>: Rem(<term>, <factor>)
;

transform factor
    on "PRIMARY"  <factor>: Primary(<primary>),
    on "NOT_PRIMARY"  <factor>: NotPrimary(<primary>),
    on "ABS_PRIMARY"  <factor>: AbsPrimary(<primary>),
    on "EXPON"     <factor>: Expon(<primary>, <primary>)
;

transform primary
    on "LITERAL"  <primary>: Literal(<literal>),
    on "NAME"     <primary>: PrimaryName(<name>),
on "ALLOCATOR" <primary>: Allocator(<allocator>),
on "QUALIFIED" <primary>: Qualified(<qualified>),
on "EXPR" <primary>: Parens(<expression>),
on "AGGREGATE" <primary>: PrimaryAgg(<aggregate>);

transform qualified
on "AGGREGATE" <qualified>: NameTicAgg(<name>, <aggregate>),
on "EXPR" <qualified>: NameTicExpr(<name>, <expression>);

transform subprog_decl
on "SUBPROG" <subprog_decl>: SubprogSpec(<generic_hdr>,
<subprog_spec>, <psdl_met_opt>),
on "GENERIC_SUBPROG" <subprog_decl>: GenericSubprogInst(<subprog_spec>,
<generic_inst>, <psdl_met_opt>),
on "ABSTRACT_SUBPROG" <subprog_decl>: AbstractSubprogSpec(<subprog_spec>,
<psdl_met_opt>);

transform psdl_met_opt
on "usec" <psdl_met_opt>: MetUsec(<integer>),
on "msec" <psdl_met_opt>: MetMs(<integer>),
on "sec" <psdl_met_opt>: MetSec(<integer>),
on "min" <psdl_met_opt>: MetMin(<integer>),
on "hrs" <psdl_met_opt>: MetHrs(<integer>);

transform subprog_spec
on "PROCEDURE" <subprog_spec>: SubProgProc(<compound_name>,
<formal_part_opt>),
on "FUNCTION" <subprog_spec>: SubProgFuncReturn(<designator>,
<formal_part_opt>, <name>),
on "FUNCTION_DESIGNATOR" <subprog_spec>: SubProgFunc(<designator>);

transform designator
on "COMPOUND_NAME" <designator>: DesigCompound(<compound_name>),
on "STRING" <designator>: DesigString(<QUOTED_STRING>);

transform pkg_decl
on "PKG_SPEC" <pkg_decl>: PkgSpec(<generic_hdr>, <pkg_spec>),
on "GENERIC_PKG_INST" <pkg_decl>: GenPkgInst(<compound_name>,
<generic_inst>);

transform use_clause
on "USE" <use_clause>: Use(<name_s>),
on "USE_TYPE"

transform rename_decl
on "SUBTYPE"

on "EXCEPTION"
on "UNIT"

transform rename_unit
on "PKG"
on "SUBPROG"

transform task_spec
on "TASK"
on "TASK_TYPE"

transform prot_spec
on "PROTECTED"
on "PROTECTED_TYPE"

transform prot_op_decl
on "ENTRY"
on "SUBPROG"
on "REP_SPEC"
on "PRAGMA"

transform prot_elem_decl
on "OP_DECL"
on "ELEM_DECL"

transform entry_decl
on "ENTRY"
on "RANGE_ENTRY"

transform context_spec_opt
on "CONTEXT_SPEC"
transform context_spec
  on "CONTEXT_WITH_USE" <context_spec>:
    ContextWithUse(<context_spec_opt>,
                   <with_clause>, <use_clause_opt>),
  on "PRAGMA" <context_spec>:
    ContextPragma(<context_spec>, <pragma>)

transform body_stub
  on "TASK" <body_stub>:
    TaskStub(<identifier>),
  on "PKG" <body_stub>:
    PkgStub(<compound_name>),
  on "SUBPROG" <body_stub>:
    SubprogStub(<subprog_spec>),
  on "PROTECTED" <body_stub>:
    ProtStub(<identifier>)

transform generic_hdr
  on "GENERIC_FORMALS" <generic_hdr>:
    GenericHdr(<generic_formal_part>)

transform generic_formal
  on "PARM" <generic_formal>:
    GenParm(<param>),
  on "TYPE" <generic_formal>:
    GenTypeParm(<identifier>,
                 <generic_discrim_part_opt>, <generic_type_def>),
  on "PROCEDURE" <generic_formal>:
    GenProcParm(<identifier>,
                 <formal_part_opt>, <subp_default>),
  on "FUNCTION" <generic_formal>:
    GenFuncParm(<designator>,
                 <formal_part_opt>, <name>, <subp_default>),
  on "PKG" <generic_formal>:
    GenPkgParmBox(<identifier>,
                  <name>),
  on "PKG" <generic_formal>:
    GenPkgParm(<identifier>, <name>),
  on "USE" <generic_formal>:
    GenUseParm(<use_clause>)

transform generic_type_def
  on "=" <generic_type_def>:
    GenTypeBox,
  on "MOD=" <generic_type_def>:
    GenTypeModBox,
  on "DELTA=" <generic_type_def>:
    GenTypeDeltaBox,
  on "DELTA_DIGITS=" <generic_type_def>:
    GenTypeDeltaDigBox,
  on "DIGITS=" <generic_type_def>:
    GenTypeDigitsBox,
  on "ARRAY" <generic_type_def>:
    GenTypeArray(<array_type>),
  on "ACCESS" <generic_type_def>:
    GenTypeAccess(<access_type>),
  on "PRIVATE" <generic_type_def>:
    GenTypePriv(<private_type>),
  on "DERIVED" <generic_type_def>:
    GenTypeDerived(<generic Derived_type>)

transform generic derived_type
  on "SUBTYPE" <generic Derived_type>:
    GenDerivedSub(<subtype_ind>)
on "PRIVATE"  <generic-derived-type>:
              GenDerivedSubPriv(<subtype_ind>),
on "ABSTRACT"<generic-derived-type>: GenDerivedAbst(<subtype_ind>)

transform rep_spec
on "ATTRIBUTE" <rep_spec>: AttribDef(<mark>, <expression>),
on "RECORD"   <rep_spec>: RecordTypeSpec(<mark>, <align_opt>,
                          <comp_loc_s>),
on "ADDRESS"   <rep_spec>: AddressSpec(<mark>, <expression>)

transform object_qualifier_opt
on "ALIASED"   <object_qualifier_opt>: Aliased,
on "CONSTANT"  <object_qualifier_opt>: Constant,
on "ALIASED_CONSTANT" <object_qualifier_opt>: AliasedConst

transform init_opt
on "ASSIGN"    <init_opt>: ExprInitOpt(<expression>)

transform discrim_part_opt
on "DISCRIM"   <discrim_part_opt>: DiscrimPart(<discrim_spec_s>),
on "BOX"       <discrim_part_opt>: Box

transform range_spec_opt
on "RANGE"    <range_spec_opt>: RangeSpecOpt(<range_spec>)

transform aliased_opt
on "ALIASED"  <aliased_opt>: AliasedOpt

transform range_constr_opt
on "RANGE_CONSTRAINT" <range_constr_opt>:
                        RangeConstr(<range_constraint>)

transform tagged_opt
on "TAGGED"    <tagged_opt>: Tagged,
on "ABSTRACT_TAGGED" <tagged_opt>: AbstractTagged

transform access_opt
on "ACCESS"    <access_opt>: AccessOpt

transform prot_opt
on "PROTECTED" <prot_opt>: Protected
transform formal_part_opt
  on "FORMALS"  <formal_part_opt>: FormalPart(<param_s>)

transform mode
  on "IN"  <mode>: InMode,
  on "OUT"  <mode>: OutMode,
  on "IN_OUT"  <mode>: InOutMode,
  on "ACCESS"  <mode>: AccessMode

transform private_part
  on "PRIVATE"  <private_part>: Private(<decl_item_s>)

transform limited_opt
  on "LIMITED"  <limited_opt>: Limited

transform task_def
  on "TASK"  <task_def>: TaskDef(<entry_decl_s>, <rep_spec_s>,
               <task_private_opt>)

transform task_private_opt
  on "PRIVATE"  <task_private_opt>: TaskPvtOpt(<entry_decl_s>,
               <rep_spec_s>)

transform prot_private_opt
  on "PRIVATE"  <prot_private_opt>: ProtPvtOpt(<prot_elem_decl_s>)

transform private_opt
  on "PRIVATE"  <private_opt>: PrivateOpt

transform generic_discrim_part_opt
  on "DISCRIMINANT"  <generic_discrim_part_opt>: GenDisc(<discrim_spec_s>),
  on "BOX"  <generic_discrim_part_opt>: GenBox

transform subp_default
  on "NAME"  <subp_default>: SubpDefName(<name>),
  on "BOX"  <subp_default>: SubpDefBox

transform align_opt
  on "ALIGN"  <align_opt>: AlignOpt(<expression>)
transform rep_spec_s
  on "REPRESENTATION_SPECS"<rep_spec_s>: RepSpecList(<rep_spec_s>,
  <rep_spec>, <pragma_s>)
;
transform comp_loc_s
  on "COMP_LOCS"<comp_loc_s>: CompLocList(<comp_loc_s>, <mark>,
  <expression>, <range>)
;
transform type_completion
  on "TYPES"<type_completion>: TypeDefCompl(<type_def>)
;
APPENDIX F. INSTALLATION AND USE

In order to use the translator, all of the SSL source files contained in Appendices A through E must be installed. An executable is created utilizing the makefile shown in Figure 26. The Synthesizer Generator version 4.1 is required in order to create the executable. This executable is created to run in either an interactive mode or a batch mode by including the -ssl_interpreter switch. In either case, the translator may only be executed from within the X Windows System environment. Execution in interactive mode is initiated by the command:

```
pkgtrans
```

In order to execute using the batch mode, the command is:

```
pkgtrans -b -l scriptfile
```
where *scriptfile* is a file containing SSL commands which are to be executed by the translator. A script file is shown in Figure 27. This script file reads in an Ada package

```ada
Open("temp.strip", "compilation", "No")!
Save_as("Text", "temp.annotated.ada", "BASEVIEW")!
Change_view("PSDL_VIEW", false)!
Save_as("Text", "temp.psdl", "PSDL_VIEW")!
Exit();
```

**Figure 27. Batch Mode Script File**

specification required to be in a file named temp.strip, this file is a preprocessed package specification which has had all comments removed from it by a comment stripping processor (source code follows text). The output of the batch mode is two files, the first is a file named temp.annotated.ada which is temp.strip with error comments from the translation inserted. The second file produced is called temp.psdl and contains the PSDL translation of temp.strip. The file names temp.* are hard coded due to restrictions on command line parameters for the translator in the batch mode. In order to provide more flexibility, a shell file is used, which allows for the use of command line parameters and provides automatic comment stripping. This shell file is shown in Figure 28. This shell

```bash
cstrip $1 temp.strip
pkgtrans -b -l transcript
mv temp.psdl $2
```

**Figure 28. Translator Shell Execution File**

allows user specified input Ada files and output PSDL files and assuming the file is named AdaToPsdl, may be executed as follows:

```
AdaToPsdl PkgSpec.a PkgSpec.psdl
```

This will translate the file PkgSpec.a to a PSDL file named PkgSpec.psdl, and will also produce the file temp.annotated.ada. The source code for a comment stripping program follows:
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

FILE *infile, *outfile;
int inQuote = 0;
char ch;
char chline[1256];
int currCh = 0;
int backup = 1;

void flush EOF() {
    char ch1;
    currCh -= backup;
    chline[currCh] = 0;
    if (currCh)
        fprintf(outFile, "%s
", chline);
    do {
        ch1 = fgetc(infile);
    } while ((ch1 != EOF) && (ch1 != '\n'));
    currCh = 0;
    backup = 1;
}

cchar GetCh() {
    char ch;
    do {
        ch = fgetc(infile);
    } while ((ch == '\n') && (ch != EOF) && !inQuote);
    chline[currCh++] = ch;
    return ch;
}

int main(int argc, char **argv) {
    if (argc < 3) {
        printf("USAGE: cstrip infile outfile\n");
        exit(0);
    }
    infile = fopen(argv[1],"r");
    if (!infile) {
        printf("Could not open %s for reading\n", argv[1]);
        exit(0);
    }
    return 0;
}
outfile = fopen(argv[2], "w");
if (!outfile) {
    printf("Could not open %s for writing\n", argv[2]);
    exit(0);
}
while ((ch = GetCh()) != EOF) {
    if (ch == 'V') inQuote = !inQuote;
    if (!inQuote) {
        if (ch == '-') {
            backup++; ch = GetCh();
            if ((ch == '-')) {
                backup++; ch = GetCh();
                if ((ch == 'P')) {
                    backup++; ch = GetCh();
                    if ((ch == 'S')) {
                        backup++; ch = GetCh();
                        if ((ch == 'D')) continue;
                    }
                }
            }
        } else if (ch == 'Y') inQuote = !inQuote;
    }
}
backup = 1;
if (ch == 'n') {
    if (currCh != 1) {
        currCh[chline] = 0;
        fprintf(outfile, "%s", chline);
    }
    currCh = 0;
}
if (currCh) {
    currCh[chline] = 0;
    fprintf(outfile,"%s\n",chline);
}
fclose(infile);
fclose(outfile);
return 0;
}

This program strips out all Ada comments with the exception of those which begin as:

--PSDL

comments of this sort are used by the translator to recognize PSDL constructs annotated
within Ada programs.
APPENDIX G. ADDING PROCEDURE WRAPPERS FOR ADA FUNCTIONS

The current implementation of CAPS expects all PSDL operators to be implemented as Ada procedures. Unfortunately, most software components are written using a mix of functions and procedures. In order to perform a complete translation of an Ada software component to PSDL, it is necessary to add procedure interfaces for any functions which are specified in the Ada package. In order to accomplish this, preprocessing must be performed on both the Ada package specification, and the Ada package body to insert the required procedure wrappers. Figure 29 shows a sample Ada package containing both a generic function and a non-generic function.

```
package TestPkg is
  generic
    type x is private;
  function func1(y : x) return float;

  function func2(z : character) return integer;
end TestPkg;

package body TestPkg is
  function func1(y : x) return float is
    begin
      return 1.0;
    end func1;

  function func2(z : character) return integer is
    begin
      return character'pos(z);
    end func2;
end TestPkg;
```

Figure 29. Ada package with functions only

package containing both a generic function and a non-generic function.
The same package following preprocessing is shown in Figure 30. Procedure

package TestPkg is

   generic
       type x is private;
   function func1(y : x) return float;

   generic
       type x is private;
   procedure procedure_func1(y : x; ProcReturn : out float);

   function func2(z : character) return integer;

   procedure procedure_func2(z : character; ProcReturn : out integer);
end TestPkg;

package body TestPkg is

   function func1(y : x) return float is
      begin
         return 1.0;
      end func1;

   procedure procedure_func1(y : x; ProcReturn : out float) is
      function func_inst is new func1(x);
      begin
         ProcReturn := func_inst(y);
      end procedure_func1;

   function func2(z : character) return integer is
      begin
         return character'pos(z);
      end func2;

   procedure procedure_func2(z : character; ProcReturn : out integer) is
      begin
         ProcReturn := func2(z);
      end procedure_func2;
end TestPkg;

Figure 30. Ada package with procedure wrappers for functions
interfaces have been added to provide access to all declared functions. Note that for generic functions, a generic procedure must be created with identical generic formal parameters which will be used to instantiate a version of the generic function within the procedure body.
<table>
<thead>
<tr>
<th>No.</th>
<th>Name and Position</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Defense Technical Information Center</td>
<td>Cameron Station, Alexandria, Virginia 22304-6145</td>
</tr>
<tr>
<td>2.</td>
<td>Library, Code 52</td>
<td>Naval Postgraduate School, Monterey, California 93943-5101</td>
</tr>
<tr>
<td>3.</td>
<td>Chairman, Code CS</td>
<td>Computer Science Department, Naval Postgraduate School, Monterey, California 93943-5000</td>
</tr>
<tr>
<td>4.</td>
<td>Dr. Man-tak Shing, Code CS/Sh</td>
<td>Computer Science Department, Naval Postgraduate School, Monterey, California 93943-5000</td>
</tr>
<tr>
<td>5.</td>
<td>Dr. Luqi, Code CS/Lq</td>
<td>Computer Science Department, Naval Postgraduate School, Monterey, California 93943-5000</td>
</tr>
<tr>
<td>6.</td>
<td>LT Christopher Eagle</td>
<td>936 Darby Road, Virginia Beach, VA 23464</td>
</tr>
<tr>
<td>7.</td>
<td>Chief of Naval Research</td>
<td>800 N. Quincy Street, Arlington, VA 22217</td>
</tr>
</tbody>
</table>
   Software Engineering Institute
   Department of Computer Science
   Attn: Mario R. Barbacci
   Pittsburgh, PA 15213-3890

10. Office of Naval Technology
    800 N. Quincy Street
    Code 227 ONT
    Attn: Elizabeth Wald
    Arlington, VA 22132-5000

11. Advanced Research Projects Agency (ARPA)
    Integrated Strategic Technology Office (ISTO)
    Attn: Kirsti Bellman
    1400 Wilson Boulevard
    Arlington, VA 22209-2308

12. ARPA/ISTO
    3701 N. Fairfax Drive
    Attn: Edward Thompson
    Arlington, VA 22203-1714

13. Attn: Mr. William McCoy
    Code K54, NSWC
    Dahlgren, VA 22448

14. Advanced Research Projects Agency (ARPA)
    Director, Naval Technology Office
    1400 Wilson Boulevard
    Arlington, VA 22209-2308

15. Advanced Research Projects Agency (ARPA)
    Director, Prototype Projects Office
    1400 Wilson Boulevard
    Arlington, VA 22209-2308

16. Advanced Research Projects Agency (ARPA)
    Director, Tactical Technology Office
    1400 Wilson Boulevard
    Arlington, VA 22209-2308
17. Dr. Aimram Yehudai ................................. 1
   Tel Aviv University
   School of Mathematical Sciences
   Department of Computer Science
   Tel Aviv, Israel 69978

18. Dr. Robert M. Balzer ................................. 1
   USC-Information Sciences Institute
   4676 Admiralty Way
   Suite 1001
   Marina del Ray, CA 90292-6695

19. Dr. Raymond Yeh ................................. 1
   International Software Systems, Inc.
   9430 Research Blvd, Bldg 4, Suite 250
   Austin, TX 78759

20. Kestrel Institute ................................. 1
    Attn: Dr. C. Green
    1801 Page Mill Road
    Palo Alto, CA 94304

21. Prof. D. Berry ................................. 1
    Computer Science Department
    Technion
    Haifa 32000
    Israel

22. Massachusetts Institute of Technology ................................. 1
    Department of Electrical Engineering and Computer Science
    Attn: Dr. B. Liskov
    545 Tech Square
    Cambridge, MA 02139

23. Massachusetts Institute of Technology ................................. 1
    Department of Electrical Engineering and Computer Science
    Attn: Dr. J. Guttag
    545 Tech Square
    Cambridge, MA 02139
24. Dr. Michael Gray ..............................................................1
    CSIS Department
    American University
    4400 Massachusetts Ave. NW
    Washington, D.C. 20016-8116

25. National Science Foundation ..............................................1
    Division of Computer and Computation Research
    1800 G Street NW
    Attn: Dr. Bruce Barnes
    Washington, D.C. 20550

26. Mr. Hans Mumm ..............................................................1
    NCCOSC RDTE DIV 4121
    53560 Hull Street
    San Diego, CA 92152-5001

27. NAVSEA, PMS-4123H ......................................................1
    Attn: William Wilder
    Arlington, VA 22202-5101

28. New Jersey Institute of Technology ......................................1
    Computer Science Department
    Attn: Dr. Peter Ng
    Newark, NJ 07102

29. Office of Naval Research ..................................................1
    Computer Science Division, Code 1133
    Attn: Dr. Van Tilborg
    800 N. Quincy Street
    Arlington, VA 22217-5000

30. Office of Naval Research ..................................................1
    Computer Science Division, Code 1133
    Attn: Dr. R. Wachter
    800 N. Quincy Street
    Arlington, VA 22217-5000

31. Southern Methodist University ..........................................1
    Computer Science Department
    Attn: Dr. Murat Tanik
    Dallas, TX 75275
32. Ohio State University
Department of Computer and Information Science
Attn: Dr. Ming Liu
2036 Neil Ave Mall
Columbus, OH 43210-1277

33. University of California at Berkeley
Department of Electrical Engineering & Computer Science
Computer Science Division
Attn: Dr. C.V. Ramamoorthy
Berkeley, CA 90024

34. University of Maryland
College of Business Management
Tydings Hall, Room 0137
Attn: Dr. Alan Hevner
College Park, MD 20742

35. University of Pittsburgh
Department of Computer Science
Attn: Dr. Alfs Berztiss
Pittsburgh, PA 15620

36. Research Administration
Code 012
Naval Postgraduate School
Monterey, CA 93943

37. Attn: Dr. David Hislop
Laboratory Command
Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

38. Monmouth College
Computer Science Department
Software Engineering Program
Attn: Jerry Powell
West Long Branch, NJ 07764