Design and construction of language processor based on attribute grammar for EIS

Vijayant Palaiya

The University of Montana

Follow this and additional works at: https://scholarworks.umt.edu/etd

Let us know how access to this document benefits you.

Recommended Citation
https://scholarworks.umt.edu/etd/6583

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.
The University of **MONTANA**

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

**Please check "Yes" or "No" and provide signature**

Yes, I grant permission
No, I do not grant permission

Author's Signature ____________

Date 5/23/96

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.
Design and construction of language processor based on attribute grammar for EIS

by

Vijayant Palaiya

B.Tech. Indian Institute of Technology, Bombay, India, 1992

presented in partial fulfillment of the requirements

for the degree of

Master of Science

The University of Montana

May 1996

Approved by:

Chairperson

Dean, Graduate School

Date
Design and construction of language processor based on attribute grammar for EIS (72 pp.)

Director: Ray Ford

The Ecosystem Information System (EIS) is a distributed system used by ecosystem modelers to create, modify and access network-based repository. The information in the EIS repository is organized hierarchically using object-oriented principles. A definition language is used to describe the EIS data, where meta-data descriptions are classes, datasets are instances, and data transformations are methods. Attribute Grammar methodology is used to formally specify the syntactic and static semantic aspects of the EIS language. The attribute grammar specifications are used to construct a language processor for EIS which supports "batch" processing of complete EIS hierarchy descriptions, and "incremental" processing of individual class, instance or method definitions in the hierarchy.
List of Figures

2.1 Derivation Tree ................................................................. 7
2.2 Attributed Tree for $5 + 4 \times 7$ ............................................. 10
2.3 Class Hierarchy ................................................................. 12

3.1 Example production rule from attribute grammar .................. 17
3.2 State Variable definition .................................................... 19
3.3 enumerated type definition ................................................. 20
3.4 array type definition ......................................................... 21
3.5 record type definition ....................................................... 22
3.6 set type definition ........................................................... 22
3.7 Constant definition .......................................................... 23
3.8 Function definition .......................................................... 24
3.9 Parameter declaration ....................................................... 25
3.10 Parameter assignment ..................................................... 25
3.11 State Variable assignment ............................................... 26
3.12 Interface Use definition ................................................... 27
3.13 Keyword definition ......................................................... 27
3.14 Document definition ....................................................... 28
3.15 Example Class Definition ................................................ 29
3.16 List of semantic checks .................................................... 31
List of Tables

2.1 Example Attribute Grammar ........................................... 8
Chapter 1

Introduction

1.1 Overview

The Ecosystem Information System (EIS) is an object-oriented network-based system that supports the creation of a distributed repository of ecosystem and natural resource information. The EIS data repository is organized hierarchically using an object-oriented framework, where meta-data descriptions are classes, datasets are instances, and data transformations are methods. EIS is a user-friendly system that allows users to browse and extend the information in a convenient fashion. The transparent nature of the physical location of the EIS database allows users to access information that can be located on any of the myriad of EIS servers across the internet.

EIS includes a definition language that allows the EIS user to construct a meta-description (i.e. class description) of a particular type of dataset. The class definition includes the description of various class attributes, such as state variables, constants, type definitions, and operational components. Also included in the language is the formulation of a class hierarchy, the definition of class instances that represent datasets of a particular class, and the definition of class methods that represent program com-

1
ponents that implement operations defined for a particular class.

A BNF (Backus-Naur Form) language specification is a standard formalism to define the syntactic aspects of a subject language [11]. BNF specifications are essentially context-free grammars, written in a stylized form. The BNF formalism can be used with standard tools to semi-automatically generate a parser for the subject language. Regular expressions are another standard formalism which use regular grammars to define the lexical specification of the language. Regular expressions can be used as the basis for semi-automatically developing a lexical analyzer, which is typically used as a "front-end" for a language parser.

An attribute grammar is a well-known language specification technique that extends the capabilities of a context-free language (i.e. BNF based) description. An attribute grammar adds a system of attributes and attribute computations to the BNF grammar specification; the additional power of the attribute computation allows an attribute grammar to address a target language's static and dynamic semantic properties. Formally, the attribute computation rules represent a form of state machine computation that extends the context-free BNF to permit a wide range of grammar based computations. Most often, attribute grammars are used to provide a standard way to specify the static semantic aspects of a language. As with BNF and regular expression specification tools, there are semi-automatic systems based on attribute grammars that generate language processing implementations [10], but none of these tools has achieved widespread use. Despite this, the attribute grammar model provides a standard basis for combining formal specification tools to specify all aspects of a programming language, and for ultimately combining semi-automatically generated lexical analyzer and parser parts with hand-crafted semantic analysis, symbol table construction and reference, and code generation parts.
1.2 Problem

The purpose of this project is to define an attribute grammar that specifies complete syntax and static semantics for the EIS language, and to verify that the attribute grammar is "correct" in some sense. The attribute grammar for EIS must define a language that allows creation of a class hierarchy whose elements (i.e., classes, instances, and methods) conform to the syntactic and semantic rules defined for the EIS language. Simple syntactic errors will violate the BNF part of the specification, and will be detected by parsing. On the other hand, static semantic errors occur whenever a part of the hierarchy description violates the static semantic rules imposed on the language, and will be detected through computations on the attributes that decorate a derivation tree in an attribute grammar based derivation. Example static semantic rules are that all the class nodes in the hierarchy should have unique names, and that a particular property cannot be defined more than once within class definitions along the path from any class $C$ upwards to the root. There are several other such rules that embody principles such as "inheritance" that are central to an object-oriented specification framework such as EIS.

The first prototype attribute grammar for EIS was built two years ago, based on a prototype syntactic description and relatively simple approach to static semantic checking [1]. A new version of the EIS language now under development has a much more complete syntactic structure and a much more extensive specification of static semantic analysis. For example, the new version of the EIS language defines syntactic and semantic specifications in an incremental fashion, consistent with the way a class hierarchy might be incrementally built in practice. This means that whenever a class node is added (or modified) in the hierarchy, the syntactic and semantic rules of the attribute grammar are used to verify that the complete hierarchy is still consistent.
Generally, the attribute grammar description should reflect our intuition: it should be possible to add or modify this node to the hierarchy by checking only its description based upon the context in which it exists/will exist in the hierarchy.

This thesis is organized as follows. Chapter 2 provides background on attribute grammars and their application in EIS. Chapter 3 describes the syntactical and semantic features of the target EIS language, and gives the (new) attribute grammar for EIS. Chapter 4 includes a discussion of the “correctness” of the attribute grammar in specific terms related to its structure and to its treatment of the collection of static semantic restrictions. Finally, Chapter 5 describes the construction of language processor for EIS based on the attribute grammar, and summarizes the current state of the language specification and language processing implementation.
Chapter 2

Background

2.1 Attribute Grammar Definition

A language can be defined in terms of what it looks like (the syntax of the language) and what it means (the semantics of the language). The syntax of the language is usually specified using a widely used notation called a context-free grammar or BNF (Backus-Naur Form) specification. The accompanying lexical structure of the language is usually defined using a regular expression. An attribute grammar uses the idea of a BNF plus a finite state machine to extend the language specification to address language semantics.

Informally, each grammar symbol in the context-free grammar is associated with a set of attributes, such that each instance of a symbol occurring within a derivation will have a value for each of its attributes. These attribute values can be either synthesized, i.e., depend on the values of descendant nodes in the derivation tree, or inherited, i.e., defined in terms of the values of attributes of the ancestor nodes of the non-terminal symbol. An attribute has a type, and can be a string, character, integer, real number, boolean, or any complex data structure. Rules that define the computation of attribute values are associated with each production rule in the
BNF, with the assumption that when a production rule is applied to create the next derivation step, the attribute computation rules are also applied to compute attribute values for the derived elements. We now formally define attribute grammar and associated terminology.

An Attribute Grammar can be defined as $AG = (G, A, R)$, where

- $G$ is the context-free grammar,
- $A$ is a finite set of attributes, such that each attribute is associated with some subset of the non-terminals in $G$, and
- $R$ is a finite set of semantic rules, such that each rule is associated with a single production in $G$.

$G$ specifies the syntax, and $A$ and $R$ specify all or part of the semantics of the target language.

A context-free grammar $G$ can be defined as $G = (V, N, S, P)$, where

- $V$ is a finite set of terminal and non-terminal symbols,
- $N$ is the finite set of non-terminal symbols, and $N \subseteq V$,
- $S$ is the start symbol, and $S \in N$, and
- $P$ is the set of production rules.

The start symbol does not appear on the right side of any production rule.

For each non-terminal symbol $X \in V$, we associate a finite set of attributes $A(X)$ [8]. This finite set of attributes can be partitioned into two disjoint subsets of attributes called inherited attributes $A_I(X)$ and synthesized attributes $A_S(X)$. The
set $A_f(X)$ is empty for $X = S$, i.e., the start symbol $S$ does not have any inherited attributes. Similarly, the set $A_S(X)$ is empty for all terminal symbols.

A production $p \in P$, $p : X_0 \rightarrow X_1...X_n$ ($n \geq 0$), has an attribute occurrence $X_i.a$, if $a \in A(X_i)$, $0 \leq i \leq n$. We associate a finite set of semantic rules $R_p$ with each production $p \in P$. In the set $R_p$, there is exactly one rule for each synthesized attribute occurrence $X_0.a$, i.e., for the right side non-terminal, and exactly one rule for each inherited attribute occurrence $X_i.a$, $1 \leq i \leq n$. The set of semantic rule $R_p$ is of the form $b = f(c_1, c_2, ..., c_k)$ where $f$ is a function, and either

1. $b$ is a synthesized attribute of $X_0$ and $c_1, c_2, ..., c_k$ are attributes belonging to the grammar symbols of the production, or

2. $b$ is an inherited attribute of one of the grammar symbols on the right side of the production, and $c_1, c_2, ..., c_k$ are attributes belonging to the grammar symbols of the production.

A derivation tree is a graphical representation that starts with the start symbol of the grammar and shows how a particular string in the language is derived. For
Table 2.1: Example Attribute Grammar

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $E' \rightarrow E$</td>
<td>$E'.val = E.val$</td>
</tr>
<tr>
<td>2. $E_1 \rightarrow E_2 + T$</td>
<td>$E_1.val = E_2.val + T.val$</td>
</tr>
<tr>
<td>3. $E \rightarrow T$</td>
<td>$E.val = T.val$</td>
</tr>
<tr>
<td>4. $T \rightarrow T*F$</td>
<td>$T.val = T.val * F.val$</td>
</tr>
<tr>
<td>5. $T \rightarrow F$</td>
<td>$T.val = F.val$</td>
</tr>
<tr>
<td>6. $F \rightarrow (E)$</td>
<td>$F.val = E.val$</td>
</tr>
<tr>
<td>7. $F \rightarrow \text{num}$</td>
<td>$F.val = \text{num}.val$</td>
</tr>
</tbody>
</table>

the production $p : X_0 \rightarrow X_1...X_n$, the derivation tree will look like in Figure 2.1. The basic form of a derivation tree for an attribute grammar is identical to that used in the underlying context-free grammar. However, a complete attribute grammar derivation tree will be an attributed derivation tree showing both the structure of the context-free derivation and the values of attributes computed at each node. Attribute evaluation is the process of computing the values for the attributes of all the nodes in a derivation tree, using the semantic rules $R$.

An example attribute grammar is given in Table 2.1. This grammar defines a language of simple arithmetic expressions. The context-free grammar describes the syntax for expressions, and the attribute computations describe the value to which the expression will evaluate. The left hand column of Table 2.1 is the set of productions $P$; the right hand column shows the set of semantic rules $R$ and their association with specific productions.

In this grammar, we have

- the set of grammar symbols $V = \{E', E, T, F, \text{num}, *, +, (, )\}$,
• the set of non-terminal symbols \( N = \{E', E, T, F\} \), and

• the start symbol \( S = E' \).

Note that subscripts are used to distinguish between multiple instances of grammar symbols that occur in recursive rules, e.g., \( E_1 \) vs. \( E_2 \) in Production 2.

An example derivation, in the form of the final attributed tree for the string "\( 5 + 4 \times 7 \)" is shown in Figure 2.2. Note that the computation of the attribute values is non-trivial, in that the values for key attributes are unknown when first referenced in many cases, e.g., for non-terminals \( E \), \( T \), and \( F \). In general this will always be the case with synthesized attributes. Thus, it is one thing to specify the attribute computation: actually computing the values can be considerably more difficult.

An attribute grammar is said to be structurally well-defined if for a program generated by the grammar \( G \), the values of the attribute instances can be unambiguously computed by the semantic rules \( R \). In other words, there are no circular dependencies between attribute computations in a well-defined attribute grammar. Ordered attribute grammars are defined as a subclass of well-defined attribute grammars which satisfy the following additional condition: for each symbol of the grammar a partial order over the associated attributes can be defined, such that in any context of the symbol the attributes are evaluable in an order consistent with the partial order [7]. This property of ordered attribute grammars is important because it guarantees that a general evaluation mechanism, based on the partial order, can be used to evaluate all derivations in the language.

Generally, it is difficult to prove that an attribute grammar is "correct" in a general sense. However, verifying that an attribute grammar is well-defined, or even better, that it is ordered, is an important demonstration that the attribute grammar's computations are well behaved, and that is structurally sound.
Figure 2.2: Attributed Tree for $5 + 4 \times 7$
2.2 Application of Attribute Grammar in specifying EIS language

The attribute grammar model is nowadays standard in the specification and implementation of programming languages [11]. This model can also be applied in other language processing techniques where relations among structured information play a key role. Some significant areas where the methodology of attribute grammar is used as a specification tool include general software engineering, distributed programming, logic programming, static analysis of programs, databases, and pattern recognition [10].

We use an attribute grammar as a formalism for defining the syntactic and static semantic aspects of the EIS language. The Ecosystem Information System organizes its database of information in a hierarchical fashion, and provides a user-friendly way of browsing, extending and performing other operations on the database. The EIS database consists of class hierarchies, which are abstractions of real life hierarchical relationships. As shown in Figure 2.3 each class hierarchy is a tree structure whose nodes represent meta-data descriptions (classes), datasets (instances) or the computational processes (methods). Each node in the hierarchy has its own description in a syntax specified by the attribute grammar. The syntax of the description is different for class, method or instance node. An example class definition is shown in Figure 2.4.

Operationally, the EIS database stores information regarding the structure of the individual hierarchies, and the detailed description of individual objects within a hierarchy. The syntactic aspect of the EIS language deals with the syntax of the description of the individual objects in the hierarchy. It also describes the syntax of the description of the whole hierarchy, which is the concatenation of the description.
of the nodes in the hierarchy in an ordered form. A valid string generated by the EIS language, therefore, can be the description of individual object, or the description of the whole hierarchy itself.

The EIS class specification syntax allows the definition of one or more properties\(^1\) within a class definition. These properties denote the set of logical characteristics of the class, and can be categorized as state variables, constants, types or functions. The EIS language supports the use of basic types, such as \textit{integer}, \textit{real}, \textit{string}, \textit{char}, and \textit{boolean}, and the definition of new types using the type constructors \textit{array}, \textit{set}, \textit{record}, and \textit{enumeration}. For example, a new type \textit{new\_type} can be defined as \textit{array [1..10] of real}. The EIS language supports the principle of \textit{property inheritance} between classes, meaning that a subclass inherits the properties of all ancestor classes in the same hierarchy. In the example hierarchy shown in Figure 2.3, the class \textit{X} inherits all the properties of the ancestors \textit{Y} and \textit{Z}.

Another feature of the language is the \textit{interface uses}, which allows a class to use the properties defined in another non-ancestor class, if that class and its attributes are explicitly identified in an \textit{interface uses} specification. For example, if the class \textit{Y}

\(^1\)We use "properties" instead of the more common term "attributes" to avoid confusion within the attributes in attribute grammar.
in Figure 2.3 designates class $W$ in its *interface uses* section, then all the properties (type, state variables, constants and functions) of $W$ can be referenced within class $Y$, though they must be explicitly identified as properties of $W$ through qualification syntax, i.e., "$W. <\text{property}>"$.

The EIS language also supports the concept of a parameterized class, used to denote a family of classes whose structure and behavior are similar, but which may differ based on the actual value of a set of formal class parameters. A parameterized class definition is in effect a template which lists the formal parameters. All formal parameters must subsequently be bound to actual parameter values in either the creation of a subclass or an instance of the parameterized class.

The EIS language must conform to these and other restrictions on identifier definition and use stemming from standard object-oriented principles and specific design decisions for EIS. For example, EIS requires all entity names in a hierarchy to be unique and that the formal parameters defined in a parameterized class should be unique among its ancestors. Also there are other simple rules, such that a property name should not be duplicated within a class definition. A list of all the static semantic restrictions is given in the Chapter 3.

The correctness of the attribute grammar for EIS must be guaranteed in some manner before we use it to construct the language processor. For structural verification, we demonstrate that there is no circular dependency between attributes, that is, if an attribute $X$ in the grammar depends on the value of attribute $Y$, then the value of attribute $Y$ cannot depend on the value of $X$. We also verify the orderliness property of the attribute grammar to ensure that our attribute grammar is both well-defined and easily evaluable. The ordering property is checked by a polynomial-time algorithm [7], which computes the *visit-sequences* that can be used to automate attribute computation. For another "correctness" guarantee of the attribute gram-
mar, in Chapter 4 we review the list of static semantic restrictions, and one-by-one demonstrate that these rules are "implemented" correctly within the EIS attribute grammar.
3.1 Original EIS language

The original EIS language was based on a simple syntactic structure and simple static semantics. It used a well-formed context-free grammar to specify a prototype syntax for class descriptions, instances, and methods. The original context-free grammar provided a formal description of the language’s syntax, but provided no corresponding formal definition of the language’s static semantics. Instead, the static semantics were implicitly defined and implemented on an \textit{ad hoc} basis in the implementation of a parser/analyzer.

The prototype language was overly restrictive in a number of ways. The only class property that could be defined was the functional component, and there were no predefined types or type constructors available for a user to define other class properties. Although the attribute grammar allowed for defining parameterized classes, the semantics of binding these formal parameters in subclasses and instances was not clearly defined. The semantic checking for a new EIS object was based on the
parent class, which would contain all the information inherited from the ancestors. Although the language allowed for property inheritance, it required each object in the hierarchy to store the information of its ancestors, along with its own information. This was done as the language processor performed the semantic checks based on the information stored in the parent class. This implementation scheme created redundant information within the hierarchy, which increased dramatically as the size of the hierarchy increased. The language allowed definition of interface-uses and forward declaration of classes, but had no semantics associated with these constructs. For example, a user could not use the properties of class X in defining his own class, even if class X was specified in the interface-uses section. The original language also required identifier names to be unique across all class definitions in a hierarchy. This required users to be aware of the identifier uses in all hierarchy objects. The language specification had other minor syntactic and static semantic deficiencies. Implementing a "production-quality" EIS clearly required a new language definition and a rigorous specification of the new language.

3.2 New EIS language

The purpose of the new attribute grammar for EIS is to clearly specify the new EIS language, to provide a fuller and more robust syntactic and static-semantic language structure. This attribute grammar specifies the syntax and semantics of not only individual EIS object definitions, but also for the entire class hierarchy description. Like any other attribute grammar, the attribute grammar for EIS is specified by a context-free grammar with a set of production rules, a set of attributes for grammar symbols, and a set of semantic rules for each production in the grammar. In the notation that we use for the EIS attribute grammar, we have two parts for representing
rule var_defn : var identifier_list of type_denoter

semantic
    SymRecList(var_defn) := add_var_defn(IdList(identifier_list), SymRec(type_denoter))

condition
    not_exists_in_symtab(IdList(identifier_list))
    not_qualified(IdList(identifier_list))

end

Figure 3.1: Example production rule from attribute grammar

semantic rules for each production. The semantic part represents a set of semantic functions associated with the production, which are evaluation rules for determining the value of attributes associated with the grammar symbols in the production rule. The condition part expresses a special class of semantic rules representing constraints that must be satisfied by the attribute values in order for a derivation to be valid. Boolean attributes are used to indicate if these constraints are satisfied or not.

Figure 3.1 shows an example, with these components:

- var_defn, identifier_list and type_denoter are non-terminal symbols,
- var and of are terminal symbols,
- SymRecList, IdList and SymRec are the attributes associated with the non-terminals var_defn, identifier_list and type_denoter respectively.
- not_exists_in_symtab, not_qualified, and add_var_defn are auxiliary functions specified in the attribute grammar; the first two functions are used to check semantic constraints which must be specified whenever we apply the above production rule, whereas the third function is used to compute a specific attribute.

The new attribute grammar provides a complete description of an entire class hierarchy and a basis for incrementally processing a description of a newly added or modified node in the hierarchy. The complete attribute grammar for EIS is included
in Appendix I. This grammar could be used to derive a complete class hierarchy specification in a manner analogous to the “batch” processing of hierarchy components. In incremental processing, the attribute grammar is applied only at a new or modified node, and inherits the context information for that node from the existing class hierarchy and its attributed derivation tree.

The new attribute grammar is composed of two parts — the upper part and the lower part. The upper part of the attribute grammar defines the attributes and attribute computations for the hierarchy as a whole. It deals with the attributes that are associated with the individual objects (classes, methods or instances) and the global attributes for the whole hierarchy. The syntactic aspects of the upper part deal with specifying the hierarchy structure and the order in which object definitions occur in the input language string for the grammar. The upper part semantics deal with the correctness of the complete hierarchy definition, which involves checking for proper implementation of property inheritance, interface use, and class parameterization features within the object definitions in the hierarchy.

The lower part of the grammar deals with the attributes and attribute computations associated with various forms of object definition in the hierarchy. It checks for the correctness of the syntax and semantics of individual node description in a local context. Logically, it constructs structures which record the information defined in the class, method or instance definition. These locally computed data structures are combined to form the attributes of the hierarchy used by the upper part of the grammar to perform conditional checking for the whole hierarchy.
BNF:
\[ \text{var_defn} : \text{var identifier_list of type_denoter} \]

Examples:
\[ \text{var} \ x \ \text{of} \ \text{integer}; \]
\[ \text{var} \ y, \ z \ \text{of} \ \text{array} [1..10] \ \text{of} \ \text{real}; \]

Restrictions:
1. An identifier in the identifier_list is being defined; an identifier can have only one definition in each block (this is true for all types of identifier definition).
2. Identifiers in the identifier_list should not be qualified.
3. The type_denoter must be a predefined simple or structured type.

Figure 3.2: State Variable definition

3.3 Syntactic Features of the Attribute Grammar

The new grammar extends the original grammar by adding several new syntactic features, adding class properties for state variables, types, and constants to the function definitions included in the original grammar.

3.3.1 State Variable Definition

State variables are used to define the logical characteristics of a class. They represent the unique data associated with any instance of the class. Every state variable has a particular type, and it can store values of only that type. The BNF, examples, and restrictions for state variable definition are given in Figure 3.2. The non-terminal type_denoter specifies the type of value — simple or structured — the state variable will represent.
BNF:
   enumerated_type : '(' identifier_list ')' 

Examples:
   type color = ( red, blue, green)
   type errno = (23, 45, 66, 78, 19)

Restrictions:
   1. Each identifier in the identifier_list must be unique.
   2. Identifiers in the identifier_list should not be qualified.

Figure 3.3: enumerated type definition

3.3.2 Type Definition

EIS type definitions allow a user to specify structured state variables. A variety of data types and type constructions are available in EIS language. The predefined simple types supported by EIS are "integer", "real", "char", "string" and "boolean". The type constructors are array, record, set and enumeration.

The EIS language supports definition of enumerated types, which are a group of values depicted by identifiers that are named and ordered by the user. Enumerated types are used to name a collection of abstract values: potential error conditions, job classifications, etc. Figure 3.3 gives the BNF, examples, and restrictions for an enumerated type construction.

EIS allows construction of structured types which can be built from the simple types or structured types themselves. These user-defined types can be defined using the type constructors array, record, and set. With property inheritance, the constructed types can be used in subsequent subclasses. They can also be used by other classes which interface-use the defining class.

An array type defines a structure that contains elements of any simple or structured type. The BNF, examples, and restrictions for array type definition are shown
BNF:
array.type : array '[index.type_list ]' of type.denoter
index.type_list : index.type_list2 ',' index.type | index.type
index.type : lower_bound '..' upper_bound
lower_bound : id | value
upper_bound : id | value

Examples:
array [1..10] of integer;
array [i..j, 100..200] of array [-1..10] of real;

Restrictions:
1. The lower_bound and upper_bound must be of type integer.
2. The type.denoter should be a simple or structured type.

Figure 3.4: array type definition

in Figure 3.4.

A record structure consists of any number of typed, named fields (Figure 3.5). A single record may include fields of different types, but each field identifier in the record type must be unique within the current record definition.

The set type in EIS allows declaration of variables that can represent a set, or group, of values of any type. Figure 3.6 specifies the BNF, examples, and restrictions for set type.

3.3.3 Constant Definition

Constants give a more programming language like appearance to the EIS language and provide alternative names — identifiers — for values. Constants can be defined by the users for the explicit purpose of acting as synonyms for other values. Zero or more constants can be defined in the constant definition part. Figure 3.7 shows the BNF along with examples and restrictions for constant definition.
BNF:

record_type : record_start field_list record_end
field_list : field_list2 ';'; record_section | record_section
record_section : identifier_list ':' type_denoter

Examples:

record_start
  x, y : integer;
  p, q : set of string;
record_end

record_start
  c : char;
record_start
  b, bow : boolean;
  f, r : real;
record_end

Restrictions:

1. Each identifier in the identifier_list must be unique.
2. Identifiers in the identifier_list should not be qualified.
3. The type_denoter should be a simple or structured type.
4. Each field identifier in the record type must be unique within the current record definition.

Figure 3.5: record type definition

BNF:

set_type : set of base_type
base_type : id | enumerated_type

Examples:

set of char;
set of (plant, tree, shrub)

Restrictions

1. The base_type must be a type identifier or an enumeration.

Figure 3.6: set type definition
BNF:
  constant_defn : const id1 ':' id2 ':=' value

Examples:
  const i : integer := 10;
  const s : string := "Lary";

Restrictions:
  1. id1 must not be a qualified identifier.
  2. id2 must be of primitive type, i.e., integer, real, char, string and boolean.
  3. The type of "value" should be same as the type of id2.

Figure 3.7: Constant definition

3.3.4 Function Definition

One of the important aspects of EIS design is the inclusion of computational processes, or data transformation operations, in the class hierarchy. A data transforming function has two components: a function specification which gives the function name, argument types, and return type, and a function method which provides an implementation (i.e. executable program) to carry out the operation. Only the function specification is part of the class definition. The function specification defines what type of inputs the function requires and what type of output it produces. The BNF, examples, and restrictions for the function definition are given in Figure 3.8.

3.3.5 Parameter Declaration

In order to allow EIS users to formulate meaningful class hierarchies, we provide an additional specification mechanism known as class parameterization. Class parameterization is analogous to formal argument declarations in function specification. The formal parameters for a class can be of type class, type, function or constant. Figure 3.9 gives the BNF, examples, and restrictions for parameter declarations in a class
BNF:

function_defn : function id1 '( arg_list ')' ':' id2
arg_list : arg_list2 ',', arg_dcl | arg_dcl
arg_dcl : type_denoter

Examples:
function get_name (integer, real) : string
function is_valid (string, char) : boolean

Restrictions:

1. id1 should not be a qualified identifier.
2. Each argument in arg_list (i.e. a type_denoter) and id2 should be a simple or structured type.

Figure 3.8: Function definition

definition.

3.3.6 Parameter Assignment

Given a parameterized class definition, each parameter must eventually be bound to an actual class, type, function or constant. When creating a subclass of a parameterized class, we either specify an actual parameter value for a formal parameter, with the result that any instance or subclass of that subclass will inherit the bound value, or we simply leave the parameter(s) unbound, leaving their binding to subsequent instance creation or further subclass refinements. Binding a formal parameter to an actual parameter in instance or subclass specification is referred to as parameter assignment. The BNF for the parameter assignment is shown in Figure 3.10, along with example and restrictions.

3.3.7 State Variable Assignment

The EIS language allows the user to bind the state variables defined in a parent class. The binding is specified in a subclass or instance definition and is interpreted
BNF:
\[\text{decl\_param\_section} : \text{param\_decl} \ \text{param\_decl\_list} \ \text{end\_param\_decl};\]
\[\text{param\_decl\_list} : \text{id} \ \text{param\_type} | \text{param\_decl\_list2} \ \text{id} \ \text{param\_type}\]
\[\text{param\_type} : \text{class} | \text{type} | \text{function} | \text{const}\]

Example:
\[
\begin{align*}
\text{param\_decl} \\
\text{p1} : \text{class} \\
\text{p2} : \text{type} \\
\text{end\_param\_decl}
\end{align*}
\]

Restrictions:

1. id must not be a qualified identifier.
2. Parameter declarations must be of type class, type, function, or const.

Figure 3.9: Parameter declaration

---

BNF:
\[\text{bind\_param\_section} : \text{param\_bind} \ \text{bind\_param\_list} \ \text{end\_param\_bind};\]
\[\text{bind\_param\_list} : \text{bind\_param\_list2} \ \text{id1} \ := \ \text{id2} | \text{id1} \ := \ \text{id2}\]

Example:
\[
\begin{align*}
\text{param\_bind} \\
\text{p1} := \text{Erdas\_Lan\_Class} \\
\text{p2} := \text{char} \\
\text{end\_param\_bind}
\end{align*}
\]

Restrictions:

1. id1 must not be a qualified identifier.
2. The type of id1 must be same as the type of id2.

Figure 3.10: Parameter assignment

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
BNF:
bind_stvar_section : bind_stvar bind_stvar_list end_bind_stvar;
bind_stvar_list : bind_stvar_list2 ‘;’ id ‘:=’ value | id ‘:=’ value

Example:
bind_stvar
  x := 43
  str := “Long”
end_bind_stvar

Restrictions:

1. id must be of primitive type or a class.
2. The type of id must be same as the type of value.

Figure 3.11: State Variable assignment

as providing an initial value for the state variable in question. Unlike parameter assignments, state variable assignments are not mandatory, i.e., an instance can be defined with state variables that have no predefined initial value. The current version of the language supports binding of state variables only for variables that are primitive types and of type “class”, as shown in Figure 3.11.

3.3.8 Interface Use Definition

The interface uses section lists all the classes upon which the definition of the current class relies. The syntax, examples, and restrictions are given in Figure 3.12. Note that ancestor class properties are automatically inherited, so interface-uses is generally used to list only non-ancestor class dependencies. Specifying classes in the interface-uses section permits users to use the visible properties of non-ancestor classes in defining local class properties. The EIS language allows interface-used classes to come from the same or different hierarchies.
BNF:

interface_uses_section : interface_uses identifier_list end_interface_uses

Example:

interface_uses
  C1, H2,C3, C4
end_interface_uses

Restrictions:

1. Each identifier in identifier_list must be unique.
2. Identifiers in the identifier_list must be of type class.

Figure 3.12: Interface Use definition

BNF:

keywords_section : keywords keywords_list end_keywords
keywords_list : keywords_list2 ';'; string | string;

Example:

keywords
  "plant"; "shrub"; "tree"
end_keywords

Restrictions:

none

Figure 3.13: Keyword definition

3.3.9 Keyword Definition

EIS is designed as a tool which allows users to create, modify, extend and browse data in EIS repository. To support more ambitious network search functionality, as is common in WWW tools, it is important to have application specific keyword indices for EIS entities. The BNF, example, and restrictions for keyword definition are given in Figure 3.13.
BNF:

\[
\text{document\_section : \text{documents document\_defn\_list end\_documents}}
\]
\[
\text{document\_defn\_list : document\_defn\_list2 ; document\_defn | document\_defn;}
\]
\[
\text{document\_defn : document\_nameloc id string | documentation string;}
\]

Examples:

\[
\text{documents}
\]
\[
\text{document\_nameloc cedar "/usr/john/cedar.01 at eisgate.cs.umt.edu";}
\]
\[
\text{documentation "This is a documentation for trees";}
\]
\[
\text{end\_documents}
\]

Restrictions:

none

Figure 3.14: Document definition

3.3.10 Document Definition

The document section in the EIS language gives EIS users the option of specifying the location of documents related to the current EIS object, or putting short documentation information within the object specification itself. Figure 3.14 gives the BNF, example, and restrictions for document definition.

3.3.11 Other Features

The EIS language permits multiple definition of property names between classes within a hierarchy. That is, the identifier "g" can specify one property in class X in hierarchy H, and a different property in the class Y also in H. This can lead to a conflict in interpretation of the identifier "g". In order to identify the use of a particular class property (here "g"), the attribute grammar allows the property to be qualified by the appropriate class name, i.e., "X.g" or "Y.g". Qualification is required when specifying properties of a class X made "visible" because X is named in the interface-uses section of the current class definition.

An example class definition is given in Figure 3.15 which shows all the constructs
class X of Y
  param_bind
    p1 := A.func3
  end_param_bind
param_decl
  new_par : class;
  inv_par : type;
end_param_decl
interface_uses
  A, I, J, K
end_interface_uses
"This is description of an example class"
  type t2 := array [ 1..10, A.i..A.j ] of t1;
  var v1 of integer;
  var v2 of ( A.l, A.m, A.n );
  const c1 : integer := 98;
  const c4 : string := "Hello";
  function f1 ( A.stk, t1, t2 ) : integer;
keywords
  "plant"; "shrub"; "tree"
end_keywords
documents
  documentnameloc cedar "/usr/john/cedar.01";
end_documents
end_class

Figure 3.15: Example Class Definition

specified above.

3.4 Semantic Features of the Attribute Grammar

Semantic specification is used to associate some meaning with the programs and constructs of the subject language. The appropriate use of EIS language features such as property inheritance, interface use, class parameterization, and parameter and state variable assignments is based on satisfying constraints on how identifiers are used and bound within the context of the whole hierarchy. The constraints listed in Figure 3.2 through 3.14 are summarized in Figure 3.16. Each semantic constraint
of the language must be formally specified by one or more semantic conditions that is defined in the attribute grammar and evaluated during a derivation. In the EIS attribute grammar, each semantic condition is represented by the value of a boolean attribute. The value *true* indicates that the corresponding constraint is met: the value *false* indicates that the constraint is not met. The specification of the semantic conditions in the EIS attribute grammar is simplified by use of boolean auxiliary functions and conditional expressions, such as $=, \neq, <, >, \leq, \geq$.

### 3.5 Attributes and Attribute Computations in EIS

Logically, we associate information with a string in the EIS language by attaching attributes to the non-terminal symbols involved in the derivation of the string. The attribute grammar for the EIS language describes the logical construction of structures for key attributes, such as symbol table and abstract syntax tree, that are required to implement the appropriate semantic checks for the input language. Use of an abstract syntax tree structure helps to depict both the derivation and the flow of control of attribute computations. In our case, the abstract syntax tree also reflects the hierarchical structure of blocks in the EIS language, i.e. class, instance and method definitions. Figure 3.17 shows an attributed abstract syntax tree for the language string `const i : integer := 10`. Each node of the syntax tree is labeled not only by the grammar symbol, but also by a set of attribute-value pairs, one for each attribute associated with the symbol. Thus, attributes *Tag*, *Val* and *Type* are associated with an occurrence of the grammar symbol *value*. The value associated with each attribute occurrence in the tree is determined by applying the evaluation rules associated with the grammar production rules. A logical condition expressing a constraint that must be satisfied by the attribute values involved might also be a-
1. All class, instance and method names should be unique within a class hierarchy.

2. Each property defined locally within a class \( C_x \) must be locally unique, i.e., defined only once in \( C_x \).

3. A formal class parameter \( P_i \) declared in class \( C_x \) must be \textit{class, type, function} or \textit{const}.

4. In function definition \( f_i \) within a class \( C_x \), the arguments and the return value must be a class, a basic or constructed type.

5. A class parameter \( P_i \) must be bound to an identifier of the same type (i.e., \textit{class, type, function} or \textit{const}).

6. Each class name \( C_i \) used in the definition of class \( C_x \) should be listed in the “forward declarations” listed in the “interface uses”, locally defined within \( C_r \), or be defined on the path from \( C_x \) to the hierarchy root (i.e., an ancestor class name).

7. Each class \( C_i \) named in the “interface uses” of class \( C_x \) should exist as a class in the same hierarchy as \( C_x \), be named in the “forward declarations” of \( C_x \), or if \( C_i \) exists in another hierarchy \( H_j \), then it should be defined as \( H_j.C_i \) in the “interface uses”.

8. Including the class name \( C_i \) in the “interface uses” or “forward declarations” of class \( C_x \) makes \( C_i \) visible in \( C_x \), but does not make any properties of \( C_i \) visible in \( C_x \). Thus, a reference to property “g” of \( C_i \) in \( C_x \) must be written in a qualified form as “\( C_i.g \)”. In contrast, properties of ancestor classes of \( C_x \) are visible in \( C_x \), and can be written without qualification.

9. A formal class parameter \( P_i \) declared in class \( C_x \) must be unique along the path from \( C_x \) to the class hierarchy root.

10. A formal class parameter name \( P_i \) assigned in class \( C_x \) must be declared in an ancestor class \( C_y \) of \( C_x \), where \( C_y \neq C_x \), and cannot be assigned in any class on the path from \( C_x \) to \( C_y \).

11. A formal class parameter name \( P_i \) assigned in instance \( I_x \) must be declared in an ancestor class \( C_y \) of \( I_x \), and cannot be assigned in any class on the path from \( I_x \) to \( C_y \).

12. For an instance definition \( I_x \), all formal class parameters defined on the path from the hierarchy root to \( I_x \) must be assigned on that path or in \( I_x \).

Figure 3.16: List of semantic checks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
associated with a given language symbol. For example, the constant_defn node has two conditions associated with it; the values of these conditions are expressed in terms of the value of the auxiliary function is_primitive_type and another boolean expression.
Figure 3.17: Abstract Syntax Tree for const i: integer := 10
Chapter 4

Correctness of attribute grammar for EIS language

4.1 General aspects of attribute grammar correctness

The attribute grammar methodology has become a standard technique in the specification and implementation of programming languages. The EIS language has a syntactic structure similar to programming languages like Pascal, Ada and Modula; so we can assume that this structure is reasonable. However, once we have defined the attribute grammar to specify the static semantic aspects of EIS, we have to make sure that the attribute grammar is "correct" before we use it as a basis to construct the language processor.

A primary requirement in any attribute grammar is that the computations defined for its attribute values should be well-defined and ordered. That is, there should be no circular dependencies between attributes. Another criteria is that the computations of the attribute grammar should be "complete" and "consistent" in the manner in which they match the intent of the language designer. In our case, this means that
the attribute grammar should specify all the semantic constraints imposed on the EIS language, and should correctly identify all strings that violate the EIS semantic restrictions. Formally, if we ignore the attribute computations and look only at the BNF grammar, we find that the set of EIS language strings derivable with the BNF includes all legal EIS specifications, plus specifications that are not legal because their use of identifiers violates one or more of the restrictions listed in Figure 3.16. The purpose of the attribute grammar computations is to ensure that the semantic constraints are implemented correctly by detecting strings that are derivable according to the BNF, but not legal strings in EIS.

Checking for semantic constraints is implemented in the attribute grammar by semantic conditions associated with various productions of the attribute grammar. Thus, correct implementation of the semantic restrictions requires correct evaluation of attribute values associated with non-terminals in a derivation tree. As described in Chapter 3, the new attribute grammar is composed of a lower part and an upper part. The lower part of the grammar addresses issues in individual object descriptions, whereas the upper part addresses issues in collecting objects in a hierarchy.

4.2 Upper Part of the Attribute Grammar

The upper part of the grammar specifies the attributes, attribute computations and condition checks associated with the structure of the hierarchy. The upper part deals with the attributes of individual nodes (classes, instances and methods) in the hierarchy. It associates attribute structures like "symbol table" with each node in the hierarchy, and constructs an appropriate symbol table value using semantic rules associated with productions in the lower part of the grammar. The attribute computations and condition checks specified in the upper part are best understood as abstract op-
erations (add entry, lookup entry) on symbol tables associated with individual nodes in the hierarchy. Also the upper part defines attributes that collect information from individual object specifications in a manner appropriate to the structure of the whole hierarchy. These global attributes are required to support semantic checks based on parent-child, ancestor-descendant, and interface-uses relationships.

Figure 4.1 shows an EIS class hierarchy which we use as continuing example for our discussion on the correctness of the EIS attribute grammar. Figure 4.2 shows a subset of the EIS attribute grammar, the upper part that describes inter-entity relationships. Figure 4.3 shows the attributed tree for this example class hierarchy, giving the final results of the attribute evaluations, including the results of semantic constraint checking.

The EIS attribute grammar is based on the following key attributes. SynST is a synthesized attribute associated with every node in the hierarchy, containing the symbol table of the node itself and symbol tables of all descendant nodes. SynST is represented as a list of tuples, where each tuple "(Name, Type, SymTab)" has Name as the name of the node, Type as the type of the node ("class", "instance" or "method"), and SymTab as the symbol table of that node in the hierarchy. The attribute values for individual SymTab entities are computed by semantic rules in the lower part of the grammar. Each SymTab includes appropriate records for the identifier definitions in that block. The values of SynST attributes for various nodes are shown in Figure 4.3.

GbST is an attribute associated only with the root node, containing the symbol tables of all the objects in the hierarchy. The global symbol table is used to validate the semantic correctness of the whole hierarchy definition. Figure 4.3 illustrates conditions at Steps 5, 9, 10 and 11 that check for the uniqueness of the node names. In Step 12, the condition check validate(root_node.GbST) at the root node encapsulates
class Z of null
  - This is definition of root class Z
  ...
end_class

class W of Z
  - Definition of class W
  ...
end_class

instance I of W
  - This is definition of an instance
  ...
end_instance

class Y of Z
  - This is another class Y
  ...
end_class

class X of Y
  - Class X definition
  interface_uses Y, U, H1, A end_interface_uses
  forward_decl U, V end_forward_decl
  const i: integer := 1;
  const j: integer := 10;
  var f, g of array [i..j] of t1;
  ...
end_class

Figure 4.1: Example Hierarchy
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| **U1. rule root_node** : class_list  
  semantic  
  root_node.GbST := class_list.SynST  
  condition  
  validate(root_node.GbST)  
  end |
| **U2. rule class_list** : class_list2 class_node  
  semantic  
  class_list.SynST := append(class_list2.SynST, class_node.SynST)  
  condition  
  disjoint(class_list2.SynST, class_node.SynST)  
  end |
| **U3. rule class_list** : ε  
  semantic  
  class_list.SynST := <>  
  end |
| **U4. rule instance_list** : instance_list2 instance_node  
  semantic  
  condition  
  disjoint(instance_list2.SynST, instance_node.SynST)  
  end |
| **U5. rule instance_list** : ε  
  semantic  
  instance_list.SynST := <>  
  end |
| **U6. rule method_list** : ε  
  semantic  
  method_list.SynST := <>  
  end |
| **U7. rule class_node** : class_defn instance_list method_list class_list  
  semantic  
  class_node.SynST := append ((class_defn.Name, C, class_defn.SymTab),  
  condition  
  disjoint( (class_defn.Name, C, class_defn.SymTab),  
  end |
| **U8. rule instance_node** : instance_defn  
  semantic  
  instance_node.SynST := <(instance_defn.Name,I,instance_defn.SymTab)>  
  end |

Figure 4.2: Subset of Upper part of the attribute grammar specification
most of the semantics associated with the hierarchy, such as correct resolution of interface-use classes and parameter declaration and assignment in both class and instance nodes. The formal definition of the function \textit{validate} is given in Figure 4.4.

4.3 Lower Part of the Attribute Grammar

The lower part of the attribute grammar specifies the attributes that are associated with the specification of individual hierarchy nodes. This includes the construction of a symbol table for each node, storing the name of all identifiers defined within the node, their type, and other relevant information. These values make up the elements of the $\textit{SynST}$ list discussed earlier, and shown in Figure 4.3. The lower part of the attribute grammar also contains all checking for correct local uses of identifiers in a block. Our discussion of the correctness of the lower part of attribute grammar in imposing the local semantic restrictions on the EIS language uses class $X$ in the example hierarchy as a continuing example. One by one we discuss the different types of identifier definitions and uses, and show that the constraints imposed on the language are specified correctly by the EIS attribute grammar.

4.3.1 Correctness of identifier definitions in a class

Identifiers are declared and used in various ways in different property definitions. As discussed in Chapter 3, identifiers can be defined as state variables, constants, types, functions and parameters in a class definition. Definition of new identifiers can use identifiers that have been already defined and are visible in the current definition block. The most common usage of identifiers is in type definitions, where new types are constructed from already defined types. For example, assuming that the type definition \texttt{type t1 of string} is visible, we can define a new type \texttt{t2 as} \texttt{type t2 of}
Figure 4.3: Attributed Tree for class hierarchy in Figure 4.1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
**Function specification:**

```plaintext
function validate (GbST) return boolean
```

**Function implementation:**

```plaintext
function validate

    check_unique_names(GbST)

    foreach symbol table i in GbST begin
        identify_unresolved_identifiers(i)
        check_interface_use_declarations(i)
        check_parameter_declarations(i)
        check_parameter_assignments(i)
    end

end function validate
```

Figure 4.4: Definition of `validate` function
set of t1". Also identifiers declared as parameters can be bound to another identifiers of the same type. For instance, a parameter declared as “p3 : function” can be bound to a function identifier f1, “p3 := f1”.

Identifiers declared in a class definition are stored in the symbol table. The information stored about the identifier will depend on the type of identifier. For example, for a state variable definition, we need to store the name and type of the identifier, whereas for a constant definition, we require to store the name, type and value of the identifier. Independent of the type of identifier, the basic operation of adding an entry and lookup on the symbol table is similar for all the identifier definitions.

We will illustrate the correctness of attribute grammar computation for symbol table construction and lookup through a detailed example. We take the state variable definition “var f, g of array [i..j] of t1” as the example string. Attributed trees in Figures 4.6 and 4.8 show different steps in the computation of attribute values at nodes of the tree, and how those values are used to check the semantic constraints specified by attribute grammar. Figures 4.5 and 4.7 list the subset of the attribute grammar specification relevant to these attributed trees. We demonstrate how local semantic checks are implemented based on entries in the symbol table.

Attribute IdList is computed for the node identifier_list in Step 1, which is a list of attributes Tag of the identifiers f and g in the identifier_list. Also the condition specified by Restriction 1 in Figure 3.2 checks that each identifier specified in the identifier_list is unique.

Steps 2 and 3 compute the Tag values of nodes lower_bound and upper_bound and impose Restriction 1 specified in Figure 3.4, which states that the identifiers i and j must be of type integer. This constraint is implemented by the semantic function is_discrete_type, which requires the type of the identifiers to be integers.
Figure 4.5: Lower part attribute grammar specification
Figure 4.6: Attributed Tree for var f, g of array [i..j] of t1
L10. rule class_defn : class id1 of id2
    interface Uses_section
    forward_decl_section
    bind_param_section
    decl_param_section
    description
    mixed_decl_list
    bind_stvar_section
    keywords_section
    document_section
end_class
semantic
class_defn.SymTab := mixed_decl_list.SymRecList
condition
unique_symtab_entries(class_defn.SymTab)
end

L11. rule mixed_decl_list : ε
semantic
mixed_decl_list.SymRecList := <>
end

L12. rule mixed_decl_list : mixed_decl ';', mixed_decl_list2
semantic
mixed_decl_list.SymRecList := append(mixed_decl_list2.SymRecList,
            mixed_decl.SymRecList)
end

L13. rule mixed_decl : var_defn
semantic
mixed_decl.SymRecList := var_defn.SymRecList
end

L14. rule var_defn : var identifier_list of type_denoter
semantic
for each i ε identifier_list.IdList
    add (i, VAR, type_denoter.SymRec, NULL) to var_defn.SymRecList
condition
for each i ε identifier_list.IdList
    not_qualified(i)
end

L15. rule type_denoter : new_type
semantic
type_denoter.SymRec := new_type.SymRec
end

L16. rule new_type : array_type
semantic
new_type.SymRec := (NULL, ARRAY, array_type.SymRec, array_type.InList)
end

Figure 4.7: Lower part attribute grammar specification

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 4.8: Attributed Tree 2 for var f, g of array [i..j] of t1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Attribute $InPair$ is associated with non-terminal $index\_type$, which is a tuple of $Tag$ values of $lower\_bound$ and $upper\_bound$. For example, in Figure 4.6, the value of $InPair$ is computed as ("i", "j"). Attribute $InList$ is a list of $InPair$ values of $index\_type$, and is associated with non-terminal $index\_type\_list$.

In Step 4, we perform a lookup for identifier $t1$ in the symbol table of class $X$ to check if $t1$ has been locally defined within class $X$. If it has, then attribute $SymRec$ is assigned the value of that entry in the symbol table. $SymRec$ is an attribute representing the symbol table record, and is represented as a 4-tuple "($Name$, $Type$, $TypeDenoter$, $InList$)". The first element $Name$ is the name of the identifier in the symbol table. $Type$ is the type of property the identifier represents. For example, for an identifier representing a state variable, $Type$ will have value VAR; for unresolved identifiers, $Type$ assumes the value UNRSLVD. $TypeDen$ represents the type of the identifier, which can be either a primitive type or a constructed type. In case of a constructed type, this attribute refers to a symbol table record, which contains information of the constructed type. We assume in our example that $t1$ is not defined in class $X$ definition. In that case, $t1$ is stored as an unresolved identifier which has to be resolved later with class definitions of ancestor and interface-use classes of $X$. The value computed for attribute $SymRec$ at node $type\_denoter$ is ("t1", UNRSLVD, -, -), where "-" denotes that there is no value associated with the elements.

In Figure 4.8, Step 5 evaluates the value of $SymRec$ associated with $new\_type$. The value of $SymRec$ represents the symbol table record containing the information of the new constructed array type, i.e. the range and type of the array. Note that there is no $Name$ associated with this symbol table record. The value of second element of 4-tuple $SymRec$ is ARRAY suggesting that it is an array type construction. The third element is the symbol table entry (or $SymRec$ tuple) for identifier $t1$, which denotes the type of the array. The fourth element of the tuple is the list of lower and upper
index pairs for the array.

In Step 6, the value of symbol table records for the variable identifiers $f$ and $g$ are computed based on the $SymRec$ value of the non-terminal $type\_denoter$. Each record contains the name and type of the corresponding identifier.

Step 7 combines the symbol table records from different property definitions of class $X$. The other property definitions of class $X$ include constant definitions for identifiers $i$ and $j$ as shown in Figure 4.1(a). The condition $unique\_syntab\_entries(SymTab)$ checks for uniqueness of names of the property identifiers in class $X$. This check corresponds to Restriction 2 in Figure 3.16.

Up to this point, we have performed all the local semantic checks related to class definition of $X$. In Step 8, we pass the symbol table for $X$ to the upper part of the grammar (Step 6 in Figure 4.3) where global checks are performed for the whole hierarchy. The global check $validate$ at $root\_node$ checks for any external unresolved referencing used by identifier definitions in class $X$.

In our example, we have $t1$ as an unresolved identifier. The type $t1$ can be defined locally in class $X$, in any of its ancestor classes (i.e. $Y$ or $Z$), or in any of its interface-use class (i.e. $W$ or $H1.A$). We consider each case separately.

Case 1: $t1$ is defined in class $X$.

In this case, the reference to $t1$ is resolved in Step 4 of Figure 4.6, where function $get\_entry$ for $t1$ returns the symbol table record for $t1$.

Case 2: $t1$ is defined in ancestor class $Z$.

Property inheritance makes the properties defined in ancestor classes, $Z$ and $Y$, visible in class $X$. In Step 4 of Figure 4.6, $t1$ is stored as an unresolved identifier in the symbol table which is resolved later by $validate$ function (Step 12 of Figure 4.3). First, the symbol table of $Y$ is searched for a definition for $t1$. If the definition for $t1$ is not found, then the symbol table of the class $Z$ is searched, where we find the type
definition for $t_1$, and thus we resolve $t_1$.

Case 3: $t_1$ is defined in interface-use class $W$

If type definition for $t_1$ does not exist in any of the ancestor classes, the $\text{validate}$ function does a lookup into the symbol tables of all interface-use classes. In our example, if $t_1$ is not defined in any of the ancestor class of $X$, we then look for the type definition of $t_1$ in the interface-use classes of $X$. The type definition for $t_1$ is found in interface-use class $W$, and thus we resolve for identifier $t_1$.

Each identifier specified in the interface-use section of class definitions must be a class existing in the current hierarchy and/or in some other hierarchy, or must be specified in the forward declarations section. (Semantic check 7 of Figure 3.16). For example, in Figure 4.1, the interface-use section of class $X$ includes class $W$, which exists in the hierarchy, class $U$, which is also specified as forward declarations, and class $H1.A$, where $A$ must be a class existing in hierarchy $H1$. The implementation of this semantic check is specified by the $\text{validate}$ function associated with the root.$\text{node}$ in Figure 4.3.

Another important feature of EIS where checking the correctness of EIS language becomes important is class parameterization and parameter assignment. Semantic checks 9-12 specified in Figure 3.16 are also implemented in the $\text{validate}$ function which verifies the correct parameter declarations and bindings.

4.4 Orderness property of the attribute grammar

Besides checking the correctness of the attribute grammar based on the attribute computations and semantic constraints, we also have to verify that our attribute grammar is non-circular, and obeys the orderness property. The ordering property of the attribute grammar is checked by an algorithm, which depends polynomially
in time on the size of the input grammar [7]. The implementation of this algorithm is discussed in [14], and has been applied in EIS to verify the orderness property of the attribute grammar [1]. This attribute analysis algorithm not only determines if the attribute grammar is ordered, but also produces the visit sequences, which describes the order of the visits to the nodes in the syntax tree and of evaluations of the semantic functions between those visits. Visit sequences are computed from the attribute dependencies given by the ordered attribute grammar. They describe the control flow of an algorithm for attribute evaluation which can be part of an automatically generated language processor.
Chapter 5

Language Processor for EIS

5.1 Implementation of Language Processor based on Attribute Grammar

Formal regular expressions and context free grammars have been used extensively to generate language processors. Several specific subclasses of context free grammar have been defined specifically to facilitate the automatic generation of efficient syntax analyzers from syntax definitions [10]. Also, automatic generation of the lexical analyzers has been made possible by mapping regular expressions into finite automata. Extending this work to the even more powerful formalism of attribute grammars has been a subject of a great deal of research, both for theoretical and practical applications. Practical applications of attribute grammar have lead to the creation of a large number of automated systems based on attribute grammars. These systems have been used to generate different kinds of language processors from a high-level specification [12], such as compilers, interpreters, debuggers, editors, etc. [10]. However, despite the interest in automatic generation of language systems based in attribute grammars, no such tool has become as widely used as those for regular expressions and context-free grammars. Most uses of attribute grammars continue to be in the form...
of a specification used to guide an implementation effort, rather than automatically generating the implementation.

5.2 Implementation of Language Processor for EIS

We use the EIS attribute grammar specification as a guide in the generation of the language processor for EIS. The EIS language processor processes class hierarchy definitions, checks the syntactic and semantic validity of the input language string, and records information of definitions in the class hierarchy in the database. The information provided by the EIS language processor for correctly specified elements is stored in the EIS database, to be used later for incremental processing, when a new object is added to an existing hierarchy.

The actual EIS processor was built with the attribute grammar as a guide, but using traditional regular expression and BNF based generation tools. We use Lex and Yacc to build the lexical analyzer [9] and parser for the EIS language [6]. The link between Yacc and the EIS attribute grammar lies in mapping the formal semantic rules of the attribute grammar to the implementation code associated with each production of the grammar in the Yacc parser generator. These “semantic actions rules” associated with the code implement the logical computation of attribute values as defined in the attribute grammar. The formal specification of attribute structures like list, tuple and tag in the attribute grammar are reflected directly in corresponding attribute data structures in the language processor. The formal operations for symbol table construction and lookup in the grammar are also reflected directly in implementation operations that construct the symbol table and abstract syntax tree structures. The syntax tree structure stores the values of attributes associated with various instances of the grammar symbols, mimicking the attribute values that decorate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
the attribute grammar derivation tree. The symbol table implementation corresponds to the formal attribute definition, and the attribute grammar condition representing semantic checks are implemented in the code in a manner corresponding directly to the attribute grammar specification.

The language processor for EIS constructs global structures for the complete hierarchy, which are used to perform the semantic checks associated with the whole hierarchy. These global structures include structure for the hierarchy and the symbol table for each object in the hierarchy. The hierarchically-structured information is used to determine parent-child, ancestor-descendant and interface-use relationships. The symbol table structure of individual objects is used to resolve identifier use in a class. These global structures constructed by the language processor enables implementation of all the semantic constraints imposed on the EIS language.

The EIS language processor can take a whole hierarchy description for "batch" processing, or a preprocessed hierarchy and a new EIS object (class, method or instance) specification for "incremental" processing. Inputs can be generated by the user through the EIS user-interface, or imported from an external file. The language processor performs the syntactic and semantic analysis of the input string as discussed in Chapters 3 and 4, and based on the analysis, either accepts the input as a valid EIS definition to be added to the database, or rejects the input as illegal due to some syntactic or semantic error. In the case of incremental processing, the language processor makes sure that the modified data is consistent with the constraints imposed on the EIS language, and the context in which the new entry is being added.
5.3 Status and Conclusion

The new EIS attribute grammar, with new attributes, attribute computations and semantic constraints, has been completely defined and is included in Appendix I. The semantic checks imposed on the EIS language as defined in Table 3.14 have been successfully implemented. The EIS language processor is implemented using lex, yacc, and C++ programs, and is available for distribution ftp://www.cs.umt.edu/pub/eis/.

Further information on EIS can be found in [4], [13], [5] and [2].
Appendix I

rule root_node : class_list
  semantic
    root_node.GbST := class_list.SynST
    root_node.Info := class_list.Info
    root_node.Parent := null
    class_list.Parent := "root"
  condition
    validate(root_node.GbST)
end

rule class_list : class_list2 class_node
  semantic
    class_list.SynST := append(class_list2.SynST, class_node.SynST)
    class_list.Info := append(class_list2.Info, class_node.Info)
    class_list2.Parent := class_list.Parent
    class_node.Parent := class_list.Parent
  condition
    disjoint(class_list2.SynST, class_node.SynST)
end

rule class_list : ε
  semantic
    class_list.SynST := <>
    class_list.Info := <>
end

rule instance_list : instance_list2 instance_node
  semantic
    instance_list.Info := append(instance_list2.Info,instance_node.Info)
    instance_list2.Parent := instance_list.Parent
    instance_node.Parent := instance_list.Parent
  condition
    disjoint(instance_list2.SynST,instance_node.SynST)
end

rule instance_list : ε
  semantic
    instance_list.SynST := <>
    instance_list.Info := <>
end

rule method_list : method_list2 method_node
semantic
  method_list.Info := append(method_list2.Info, method_node.Info)
  method_list2.Parent := method_list.Parent
  method_node.Parent := method_list.Parent
condition
  disjoint(method_list2.SynST, method_node.SynST)
end

rule method_list : ε
semantic
  method_list.SynST := <>
  method_list.Info := <>
end

rule class_node : class_node class_defn instance_list method_list class_list
end_class_node
semantic
  classjaode.SynST := append((class_defn.Name, C, class_defn.SymTab),
                              instance_list.SynST, method_list.SynST,
                              class_list.SynST)
  classmode.Info := append(class_defn.Info, instance_list.Info,
                           method_list.Info, class_list.Info)
  class_defn.Parent := class_node.Parent
  class_list.Parent := class_defn.Name
  instance_list.Parent := class_defn.Name
  method_list.Parent := class_defn.Name
condition
  disjoint((class_defn.Name, C, class_defn.SymTab),
end

rule instance_node : instance_defn
semantic
  instance_node.SynST := (instance_defn.Name,I,instance_defn.SymTab)
  instance_node.Info := instance_defn.Info
condition
  instance_node.Parent = instance_defn.Parent
end

rule method_node : method_defn
semantic
method_node.Info := method.defn.Info
method.defn.Parent := method_node.Parent

end

rule class.defn : class id1 of id2
  interface.uses_section
  forward_decl_section
  bind_param_section
  decl_param_section
  description
  mixed_decl_list
  bind_stvar_section
  keywords_section
  document_section

end_class

semantic
  class.defn.Name := id1.Tag
  class.defn.Desc := description.Tag
  class.defn.SymTab := append(bind_param_section.SymRecList,
    decl_param_section.SymRecList),
  forward_decl_section.SymRecList,
  interface.uses_section.SymRecList,
  mixed_decl_list.SymRecList,
  bind_stvar_section.SymRecList)
  class.defn.KeyList := keywords_section.KeyList
  class.defn.Info := (class.defn.Name, C,
    (class.defn.Parent, class.defn.Desc,
    class.defn.KeyList, class.defn.DocList))

condition
  unique_symtab_entries(class.defn.SymTab)
  class.defn.Parent = id2.Tag
end

rule instance.defn : instance id1 of id2
  bind_param_section
  description
  bind_stvar_section
  keywords_section
  document_section

semantic
  instance.defn.Name := id1.Tag
  instance.defn.Parent := id2.Tag
  instance.defn.Desc := description.Tag
instance.defn.SymTab := append(bind_param_section.SymRecList,
    bind_stvar_section.SymRecList)
instance.defn.KeyList := keywords_section.KeyList
instance.defn.Info := (instance.defn.Name, I,
    (instance.defn.Parent, instance.defn.Desc,
    instance.defn.KeyList,instance.defn.DocList))
end

rule method_defn : method id1 of id2
    description
    keywords_section
    document_section

    semantic
    method_defn.Name := id1.Tag
    method_defn.Desc := description.Tag
    method_defn.KeyList := keywords_section.KeyList
    method_defn.Info := (method_defn.Name, M,
        (method_defn.Parent, method_defn.Desc,
        method_defn.KeyList, method_defn.DocList))
end

rule forward_decl_section : ε
    semantic
    forward_decl_section.SymRecList := <>
end

rule forward_decl_section : forward_decl identifier_list end_forward_decl
    semantic
    forward_decl_section.SymRecList := add_fwd_dcl_list(identifier_list.IdList)
    condition
    not_qualified(identifier_list.IdList)
end

rule interface_uses_section : ε
    semantic
    interface_uses_section.SymRecList := <>
end

rule interface_uses_section : interface_uses identifier_list end_interface_uses
    semantic
    interface_uses_section.SymRecList := add_int_use_list(identifier_list.IdList)
    condition
    not_qualified(identifier_list.IdList)
end

rule decl_param_section : ε
  semantic
decl_param_section.SymRecList := <>
end

rule decl_param_section : param_decl param_decl_list end_param_decl
  semantic
decl_param_section.SymRecList := param_decl_list.SymRecList
end

rule param_decl_list : id '::' param_type
  semantic
param_decl_list.SymRecList := add_param_decl(id.Tag,param_type.PType)
end

rule param_decl_list : param_decl_list2 '::' id '::' param_type
  semantic
param_decl_list.SymRecList := append(param_decl_list.SymRecList,
  add_param_decl(id.Tag,param_type.PType))
  condition
  disjoint(param_decl_list.SymRecList, add_param_decl(id.Tag, param_type.PType))
end

rule param_type : CLASS
  semantic
param_type.PType := CLASS
end

rule param_type : TYPE
  semantic
param_type.PType := TYPE
end

rule param_type : CONST
  semantic
param_type.PType := CONST
end

rule param_type : FUNCTION
  semantic
param_type.PType := FUNCTION
end

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
rule bind_param_section : ε
  semantic
    bind_param_section.SymRecList := <>
end

rule bind_param_section : param_bind bind_param_list end_param_bind
  semantic
    bind_param_section.SymRecList := bind_param_list.SymRecList
end

rule bind_param_list : id1 ':=′ id2
  semantic
    bind_param_list.SymRecList := add_bind_params(id1.Tag, get_entry(id2.Tag))
end

rule bind_param_list : bind_param_list2 ' ; ′ id1 ':=′ id2
  semantic
    bind_param_list.SymRecList := append(bind_param_list2.SymRecList,
      add_bind_params(id1.Tag, get_entry(id2.Tag)))
end

rule mixed_decl_list : ε
  semantic
    mixed_decl_list.SymRecList := <>
end

rule mixed_decl_list : mixed_decl ' ; ′ mixed_decl_list2
  semantic
    mixed_decl_list.SymRecList := append(mixed_decl_list2.SymRecList,
      mixed_decl.SymRecList)
  condition
    disjoint(mixed_decl_list2.SymRecList, mixed_decl.SymRecList)
end

rule mixed_decl : type_defn
  semantic
    mixed_decl.SymRecList := type_defn.SymRecList
end

rule mixed_decl : var_defn
  semantic
    mixed_decl.SymRecList := var_defn.SymRecList
end

rule mixed_decl : constant_defn
semantic
    mixed_decl.SymRecList := constant_defn.SymRecList
end

rule mixed_decl : function_defn
semantic
    mixed_decl.SymRecList := function_defn.SymRecList
end

rule bind_stvar_section : ε
semantic
    bind_stvar_section.SymRecList := <>
end

rule bind_stvar_section : bind_stvar bind_stvar_list end_bind_stvar
semantic
    bind_stvar_section.SymRecList := bind_stvar_list.SymRecList
end

rule bind_stvar_list : bind_stvar_list2 ';' id ':=' value
semantic
    bind_stvar_list.SymRecList := append(bind_stvar_list2.SymRecList,
        add_bind_stvars(id.Tag, value.Tag,
        get_entry(value.Type)))
end

rule bind_stvar_list : id ':=' value
semantic
    bind_stvar_list.SymRecList := add_bind_stvars(id.Tag, value.Tag,
        get_entry(value.Type)))
end

rule type_defn : type id ':=' type_denoter
semantic
    type_defn.SymRecList := add_type_defn(id.Tag, type_denoter.SymRec)
condition
    not_qualified(id.Tag)
end

rule var_defn : var identifier_list of type_denoter
semantic
    var_defn.SymRecList := add_var_defn(identifier_list.IdList,
        type_denoter.SymRec)
condition
    not_qualified(identifier_list.IdList)
rule constant_defn : const id1 ':=' id2 ':=' value
  semantic
  constant_defn.SymRecList := add_constant_defn(id1.Tag, find_type(id2.Tag), value.Tag)
  condition
  is_primitive_type(id2.Tag)
  id2.Type = value.Type
  not_qualified(id1.Tag)
end

rule function_defn : function id1 '(': argList ')' id2
  semantic
  function_defn.SymRecList := add_function_defn(id1.Tag, argList.SymRecList, get_entry(id2.Tag))
  condition
  is_type(id2.Tag)
  not_qualified(id1.Tag)
end

rule argList : ε
  semantic
  argList.SymRecList := <>
end

rule argList : arg_dcl
  semantic
  argList.SymRecList := arg_dcl.SymRec
end

rule argList : argList2 ',' arg_dcl
  semantic
end

rule arg_dcl : type_denoter
  semantic
  arg_dcl.SymRec := add_arg_dcl(type_denoter.SymRec)
end

rule type_denoter : id
  semantic
  if(lookup(id.Tag) = FALSE)
    type_denoter.SymRec := (id.Tag, UNRSLVD, NULL, NULL)
else
    type_denoter.SymRec := get_entry(id.Tag)
end

rule type_denoter : new_type
semantic
    type_denoter.SymRec := new_type.SymRec
end

rule new_type : enumerated_type
semantic
    new_type.SymRec := addEnumeratedType(enumerated_type.SymRecList)
end

rule new_type : array_type
semantic
    new_type.SymRec := addArrayType(array_type.SymRec, array_type.InList)
end

rule new_type : record_type
semantic
    new_type.SymRec := addRecordType(record_type.SymRecList)
end

rule new_type : set_type
semantic
    new_type.SymRec := addSetType(set_type.SymRec)
end

rule enumerated_type : '( identifier_list ')' 
semantic
    enumerated_type.SymRecList := addEnumvalId(identifier_list.IdList)
condition
    notQualified(identifier_list.IdList)
end

rule record_type : record_start field_list record_end
semantic
    record_type.SymRecList := field_list.SymRecList
end

rule field_list : record_section
semantic
    field_list.SymRecList := record_section.SymRecList
end

rule field_list : field_list2 ;' ;' record_section
semantic
  field_list.SymRecList := append(field_list2.SymRecList,
                                     record_section.SymRecList)
condition
  disjoint(field_list2.IdList,record_section.IdList)
end

rule record_section : identifier_list ;':'; type_denoter
semantic
  record_section.SymRecList := add_id_type_from_idlist(identifier_list.IdList,
                                                       type_denoter.SymRec)
condition
  not_qualified(identifier_list.IdList)
end

rule array_type : array ['index_type_list '] of type_denoter
semantic
  array_type.SymRec := type_denoter.SymRec
  array_type.InList := index_type_list.InList
end

rule index_type_list : index_type
semantic
  index_type_list.InList := index_type.InPair
end

rule index_type_list : index_type_list2 ,',', index_type
semantic
  index_type_list.InList := append(index_type_list2.InList, index_type.InPair)
end

rule index_type : lower_bound ',' upper_bound
semantic
  index_type.InPair := (lower_bound.Tag, upper_bound.Tag)
end

rule lower_bound : value
semantic
  lower_bound.Tag := value.Tag
condition
  is_discrete_type(value.Tag)
end
rule lower_bound : id
  semantic
    lower_bound.Tag := id.Tag
  condition
    is_discrete_type(id.Tag)
end

rule upper_bound : value
  semantic
    upper_bound.Tag := value.Tag
  condition
    is_discrete_type(value.Tag)
end

rule upper_bound : id
  semantic
    upper_bound.Tag := id.Tag
  condition
    is_discrete_type(id.Tag)
end

rule set_type : set of base_type
  semantic
    set_type.SymRec := base_type.SymRec
end

rule base_type : id
  semantic
    base_type.SymRec := get_entry(id.Tag)
end

rule base_type : enumerated_type
  semantic
    base_type.SymRec := add.enumerated_type(enumerated_type.SymRecList)
end

rule keywords.section : keywords keywords_list end_keywords
  semantic
    keywords.section.KeyList := keywords_list.KeyList
end

rule keywords.section : ε
  semantic
    keywords.section.KeyList := <>
end
rule keywords_list : string
  semantic
  keywords_list.KeyList := string.Tag
end

rule keywords_list : keywords_list2 ';' string
  semantic
  keywords_list.KeyList := append(keywords_list2.KeyList, string.Tag)
  condition
  disjoint(keywords_list2.KeyList, string.Tag)
end

rule document_section : ε
  semantic
  document_section.DocList := <>
end

rule document_section : documents document_defn_list end_documents
  semantic
end

rule document_defn_list : document_defn
  semantic
end

rule document_defn_list : document_defn_list2 ';' document_defn
  semantic
  condition
end

rule document_defn : documentnameloc id string
  semantic
  document_defn.Doc := (id.Tag, string.Tag)
end

rule document_defn : documentation string
  semantic
  document_defn.Doc := (Null, string.Tag)
end
rule value : sign unsigned_number
semantic
  value.Tag := concat(sign.Tag, unsigned_number.Tag)
  value.Type := unsigned_number.Type
end

rule value : unsigned_number
semantic
  value.Tag := unsigned_number.Tag
  value.Type := unsigned_number.Type
  value.Val := unsigned_number.Val
end

rule value : string
semantic
  value.Tag := string.Tag
  value.Type := STRING
end

rule value : character
semantic
  value.Tag := character.Tag
  value.Type := CHAR
end

rule value : boolean
semantic
  value.Tag := boolean.Tag
  value.Type := BOOL
  value.Val := boolean.Val
end

rule unsigned_number : unsigned_integer
semantic
  unsigned_number.Tag := unsigned.integer.Tag
  unsigned_number.Val := unsigned.integer.Val
  unsigned_number.Type := INT
end

rule unsigned_number : unsigned_real
semantic
  unsigned_number.Tag := unsigned.real.Tag
  unsigned_number.Val := unsigned.real.Val
  unsigned_number.Type := REAL

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
end

rule unsigned_real : unsigned_integer '.' fractional_part
semantic
  unsigned_real.Tag := concat(unsigned_integer.Tag,
      concat(".", fractional_part.Tag))
  unsigned_real.Val := unsigned_integer.Val +
      fractional_part.Val/10**fractional_part.Len
end

rule unsigned_integer : DIGITSEQUENCE
semantic
  unsigned_integer.Tag := DIGITSEQUENCE.Tag
  unsigned_integer.Val := DIGITSEQUENCE.Val
end

rule fractional_part : DIGITSEQUENCE
semantic
  fractional_part.Tag := DIGITSEQUENCE.Tag
  fractional_part.Len := DIGITSEQUENCE.Len
  fractional_part.Val := DIGITSEQUENCE.Val
end

rule sign : PLUS
semantic
  sign.Tag := "+"
  sign.SVal := 1
end

rule sign : MINUS
semantic
  sign.Tag := "-"
  sign.SVal := -1
end

rule identifier_list : id
semantic
  identifier_list.IdList := id.Tag
end

rule identifier_list : identifier_list2 ',' id
semantic
  identifier_list.IdList := append(identifier_list2.IdList, id.Tag)
condition
  disjoint(identifier_list2.IdList, id.Tag)
end

rule description : string
semantic
description.Tag := string.Tag
condition
  notnull(string)
end

rule id : id2 .'.' IDENTIFIER
semantic
  id.Tag := concat(id2.Tag, concat("." , IDENTIFIER.Tag))
end

rule id : IDENTIFIER
semantic
  id.Tag := IDENTIFIER.Tag
end

rule string : STRING_TOKEN
semantic
  string.Tag := STRING_TOKEN.Tag
end

rule character : CHARACTER_TOKEN
semantic
  character.Tag := CHARACTER_TOKEN.Tag
end

rule boolean : TRUE_TOKEN
semantic
  boolean.Tag := TRUE_TOKEN.Tag
  boolean.Val := TRUE
end

rule boolean : FALSE_TOKEN
semantic
  boolean.Tag := FALSE_TOKEN.Tag
  boolean.Val := FALSE
end
Definitions of Auxiliary Functions:

function \texttt{add\_fwd\_decl\_list}(\texttt{IdList}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{for each } i \in \texttt{IdList} & \quad \text{add } (i, \texttt{FWDCL}, \texttt{NULL}, \texttt{NULL}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_fwd\_decl\_list}

function \texttt{add\_int\_use\_list}(\texttt{IdList}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{for each } i \in \texttt{IdList} & \quad \text{add } (i, \texttt{INTUSE}, \texttt{NULL}, \texttt{NULL}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_int\_use\_list}

function \texttt{add\_param\_decl}(\texttt{Tag}, \texttt{PType}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{add } (\texttt{Tag}, \texttt{PDECL}, \texttt{NULL}, \texttt{Ptype}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_param\_decl}

function \texttt{add\_param\_bind}(\texttt{Tag}, \texttt{SymRec}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{add } (\texttt{Tag}, \texttt{PBIND}, \texttt{SymRec}, \texttt{NULL}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_param\_bind}

function \texttt{add\_stvar\_bind}(\texttt{Tag}, \texttt{Value}, \texttt{SymRec}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{add } (\texttt{Tag}, \texttt{SBIND}, \texttt{SymRec}, \texttt{Value}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_stvar\_bind}

function \texttt{add\_type\_defn}(\texttt{Tag}, \texttt{SymRec}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{add } (\texttt{Tag}, \texttt{TYPE}, \texttt{SymRec}, \texttt{NULL}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_type\_defn}

function \texttt{add\_var\_defn}(\texttt{IdList}, \texttt{SymRec}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{for each } i \in \texttt{IdList} & \quad \text{add } (i, \texttt{VAR}, \texttt{SymRec}, \texttt{NULL}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_var\_defn}

function \texttt{add\_constant\_defn}(\texttt{Tag}, \texttt{SymRec}, \texttt{Value}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{add } (\texttt{Tag}, \texttt{CONST}, \texttt{SymRec}, \texttt{Value}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_constant\_defn}

function \texttt{add\_function\_defn}(\texttt{Tag}, \texttt{SymRecList}, \texttt{SymRec}) \rightarrow \texttt{RetSymRecList}
\begin{align*}
\text{add } (\texttt{Tag}, \texttt{FUNC}, \texttt{SymRec}, \texttt{SymRecList}) \text{ to } \texttt{RetSymRecList} \\
\end{align*}
end function \texttt{add\_function\_defn}

function \texttt{add\_enumerated\_type}(\texttt{SymRecList}) \rightarrow \texttt{RetSymRec}
\begin{align*}
\text{return } (\texttt{NULL}, \texttt{ENUM}, \texttt{NULL}, \texttt{SymRecList}) \\
\end{align*}
end function \texttt{add\_enumerated\_type}

function \texttt{add\_array\_type}(\texttt{SymRec}, \texttt{InList}) \rightarrow \texttt{RetSymRec}
\begin{align*}
\text{return } (\texttt{NULL}, \texttt{ARRAY}, \texttt{SymRec}, \texttt{InList}) \\
\end{align*}
end function \texttt{add\_array\_type}
function add_record_type(SymRecList) return RetSymRec
    return (NULL, RECORD, NULL, SymRecList)
end function add_record_type

function add_set_type(SymRec) return RetSymRec
    return (NULL, SET, SymRec, NULL)
end function add_set_type
Bibliography


