1998

Ordered attribute grammar for the Ecosystem Information System

Trish Duce
The University of Montana

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

Recommended Citation
http://scholarworks.umt.edu/etd/5545

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 Theses, Dissertations, Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mail.lib.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

11/5/98

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.
An Ordered Attribute Grammar for the

Ecosystem Information System

by

Trish Duce

B.S. The University of Montana, 1993

presented in partial fulfillment of the requirements

for the degree of

Master of Science

The University of Montana

October 1998

Approved by:

[Signatures]

Chairperson

Dean, Graduate School

11-6-98

Date
Attribute grammar methodology is used to formally specify the syntactic and static semantic aspects of a language. In his original description of attribute grammars, D.E. Knuth states that semantic rules are well-defined if they are formulated in such a way that all attributes can always be defined at all nodes in any conceivable derivation tree [D.E. Knuth, Semantics of context-free languages, Math. Syst. Theory 2, 1968, 127-145]. Uwe Kastens introduces "ordered attribute grammars" as a subclass of well-defined attribute grammars, such that grammars of this class satisfy the following 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 in any derivation the attributes are evaluable in that order [U.Kastens, Ordered Attribute Grammars, Acta Informatica, Berlin; New York : Springer-Verlag, Vol 13, 1980, 229-256]. Kastens developed an algorithm to determine if an attribute grammar is "ordered". An implementation of this algorithm exists, but it contains errors and significant performance constraints. The work described here begins with debugging and reimplementing the algorithm in the programming language Java. As a major example, an attribute grammar for the Ecosystem Information System (EIS) is developed and analyzed for the "orderness" property. EIS is a network-accessible repository containing various types of information of interest to natural resource modelers and managers. Included in this repository are meta-data descriptions for various data sources, datasets, and modeling components. As such, the EIS description language involves a number of complex constraints on the use of identifiers, which represents a significant test of the use of attribute grammars in the specification of such constraints, and on the use of the new implementation of Kastens' algorithm in the analysis of such grammars.
# Table of Contents

1 Introduction ......................................................................................................................... 1  
   1.1 Overview ......................................................................................................................... 1  
   1.2 Purpose .......................................................................................................................... 2  
   1.3 Attribute Grammar Background ................................................................................. 3  
   1.4 Visit Sequences ............................................................................................................. 9  
   1.5 Spencer’s Implementation of Kastens’ Algorithm ...................................................... 10  
   1.6 Converting to Java ....................................................................................................... 11  
   1.7 Thesis Overview ......................................................................................................... 12  

2 Kastens’ Algorithm ........................................................................................................... 14  
   2.1 Attribute Grammar Notation ...................................................................................... 14  
   2.2 The “orderness” property ........................................................................................... 16  
   2.3 Constructing partial orders for symbols ................................................................. 18  
   2.4 Visit Sequences ......................................................................................................... 22  

3 Kastens’ Implementation .................................................................................................. 23  
   3.1 Attribute Grammar ...................................................................................................... 23  
   3.2 Data Structures .......................................................................................................... 23  
   3.3 Implementation .......................................................................................................... 29  

4 The EIS Attribute Grammar ............................................................................................. 44  
   4.1 Overview of EIS .......................................................................................................... 44  
   4.2 The EIS Language ....................................................................................................... 48  
      4.2.1 EIS Classes .......................................................................................................... 48  
         4.2.1.1 Class Attributes ............................................................................................ 48  
         4.2.1.2 Class Interface ............................................................................................. 51  
         4.2.1.3 Class Parameter Declarations ................................................................. 52  
         4.2.1.4 Inherited Parameter Bindings ..................................................................... 52  
         4.2.1.5 State Variable Bindings ............................................................................. 52  
         4.2.1.6 Documents and Keywords ........................................................................ 53  
   4.3 Semantic Checking in EIS ......................................................................................... 53  
   4.4 The EIS Attribute Grammar ....................................................................................... 57  

5 Execution Results .............................................................................................................. 67  
   5.1 A Simple Example ...................................................................................................... 67  
   5.2 A More Complicated Example .................................................................................. 68  
   5.3 The EIS Attribute Grammar ....................................................................................... 69  
   5.4 Dividing the EIS Attribute Grammar ......................................................................... 71
6 Analysis of Results ................................................................. 72
  6.1 The EIS Attribute Grammar .............................................. 72
  6.2 Attribute Evaluator ......................................................... 72
  6.3 Conclusion ...................................................................... 77

Appendix A - ........................................................................ 78
  An attribute grammar of a simple expression language and the visit sequences
  produced from the analysis of this attribute grammar

Appendix B ........................................................................... 82
  A very simple attribute grammar and the visit sequences produced from the
  analysis of this attribute grammar

Appendix C ........................................................................... 85
  The EIS attribute grammar and the visit sequences produced from the analysis of
  the EIS attribute grammar

References ............................................................................. 116
# List of Figures

1.1 Example Attribute Grammar ................................................................. 6  
1.2 Example Derivation Tree ................................................................. 7  
1.3 Example Attribute Grammar ............................................................. 8  

2.1 Elements of an Attribute Grammar ...................................................... 15  
2.2 Derivation Tree ................................................................................. 17  
2.3 Dependency graph $IDS$ ..................................................................... 20  

3.1 Example Symbol Table, Attribute Table, ............................................. 27  
3.2 Example Production Table ................................................................. 29  
3.3 Dependency graphs TDP and TDS ..................................................... 33  
3.4 Disjoint Partitions and F values ......................................................... 35  
3.5 Example Occurrence Map 1 after visit values and condition are added .... 38  

4.1 Example of EIS Interface ..................................................................... 46  
4.2 An EIS Hierarchy ............................................................................. 47  
4.3 Production rule for a class definition ................................................ 49  
4.4 Interface for creating a Class ............................................................. 50  
4.5 Interface for creating an Instance ..................................................... 54  
4.6 Interface for creating a Method ........................................................ 55  
4.7 EIS Hierarchy .................................................................................. 59  
4.8 Attributed Tree for class “A” ............................................................. 60  
4.9 Attribute Grammar Specification for a “classdef” ................................. 62  
4.10 Attributed Tree for class “B” ........................................................... 63  
4.11 Attributed Tree for class “C” .......................................................... 64  
4.12 Attributed Tree for upper Part of EIS Hierarchy ................................. 66  

5.1 Summary of Executions Results .......................................................... 70  

6.1 Evaluation of class “A” ..................................................................... 73  
6.2 Evaluation of class “B” .................................................................... 74  
6.3 Evaluation of class “C” .................................................................... 75  
6.4 Evaluation of Upper Part of EIS Hierarchy ......................................... 76
Chapter 1

Introduction

1.1 Overview

The Ecosystem Information System (EIS) is a network-accessible repository containing various types of information of interest to natural resource modelers and managers. Included in this repository are meta-data descriptions for various data sources, datasets, and modeling components. The EIS data repository is organized hierarchically using an object-oriented framework to order the myriad collection of components used in ecosystem modeling. In collaboration with other ecosystem modeling laboratories, the repository is being populated with information from important ecosystem modeling and management applications.

EIS needs a specification language to allow users to define EIS meta-data descriptions, datasets, and modeling components. The EIS language could be specified with a context free grammar. The context-free grammar would provide a parser/analyzer a formal description of the language's syntax, but give no corresponding formal definition of the language's "static semantics". A more complete specification of the EIS language can be formalized using an attribute grammar. An attribute grammar gives both a syntactic and static semantic language description, which can also be used as the basis for the implementation of both the parsing and the static semantic checking. This thesis uses the concepts of attribute grammars, attribute analysis algorithms, and attribute evaluation algorithms to provide a more
rigorous approach to the EIS language specification and implementation.

1.2 Purpose

The thesis has several purposes. First, it describes a well-formed attribute grammar that defines the syntactic and semantic checking that must be done to process the EIS object description language. This attribute grammar formalizes the ad hoc checking currently embedded in the parser/analyzer. The attribute grammar is also well-formed, corresponding to an ordered attribute grammar as defined by Uwe Kastens [2].

Second, the thesis describes the effort required to mechanically prove the orderness property for a non-trivial attribute grammar, using an attribute analysis algorithm developed by Uwe Kastens [2], the implementation of that algorithm by Patricia Spencer [5], and the EIS attribute grammar. The analyzer must first guarantee that the attribute grammar has the critical "orderness" property and then produce what are known as "visit sequences" for the given attribute grammar. Testing with Spencer's implementation shows that her program does not work correctly on large grammars. Thus, a major portion of the project described here is to debug and revise the original analysis code. Ultimately the decision was made to rewrite the code in the portable programming language Java. The new implementation of the attribute analysis algorithm can be used successfully with any attribute grammar; however, for our illustration purposes we focus on only the EIS attribute grammar.

The third purpose of this project is to demonstrate, with simple examples of the EIS language, that the attribute grammar and the analysis program are "correct" in the sense that the grammar specifies semantic constraints intended for the EIS language, and that attribute
evaluation identifies strings that violate the EIS semantic restrictions.

The final purpose of this thesis is to provide enough information for a future student to implement an efficient attribute evaluation algorithm. That is, the analysis currently ends with the ability to produce “visit sequences” from the analysis of any attribute grammar and an informal discussion of how evaluation would proceed. An attribute evaluation algorithm would use the visit sequences and a derivation in the same attribute grammar, and construct an attributed derivation tree which contains values for all appropriate attributes in the derivation.

1.3 Attribute Grammar Background

A language can be defined in terms of what legal strings it includes (the syntax of the language) and what meaning is attached to any string (the semantics of the language). When it comes to writing a standard definition of a language, a formal method must be used if there is any hope of the language's specification having one or more of the following qualities: completeness, consistency, precision, absence of ambiguity, conciseness, understandability, and usefulness [4].

Backus-Naur form (BNF) is a formal metalanguage that can be used to write a description or specification of a language. Basically, it is a notation that one can use to specify a generative grammar which defines the set of all possible strings of symbols that constitute programs in the subject language, together with a syntactic structure that reflects the generation process. Grammars expressible in BNF constitute the class of context-free grammars [4].
A BNF grammar has a set of production rules. Each production rule has a left side and a right side separated by some metasymbol. The left side consists of a nonterminal symbol. The right side of a rule consists of a sequence of terminal symbols and/or nonterminal symbols, where a terminal symbol is a token of the subject language.

For example, consider the following production rules, where "::" is the metasymbol used to separate left and right sides and "|" is used to separate multiple right sides with the same left side:

```
numeral : numeral digit | digit
digit : '0' | '1' | '2'
```

The nonterminal "numeral" consists of either the nonterminal "numeral" followed by the nonterminal "digit" or just the nonterminal "digit". The nonterminal "digit" consists of the terminal '0', '1', or '2'. The use of recursion in the first production rule allows an infinite number of terminal strings to be generated by a finite number of production rules. The rule for "numeral", together with the rules for the nonterminals it references and the rules for the nonterminals referenced in those rules, etc., determines the set of all strings of terminal symbols that constitute programs in the subject language.

An attribute grammar is a well-known language specification technique that extends a context free specification to allow one to formally specify aspects of the language's semantics. An attribute grammar is a context-free grammar augmented with finite state machine-like formal devices. These formal devices include "attributes" or variables associated with instances of non-terminal symbols, and "evaluation rules" associated with production rules. There is a finite set of attributes associated with each distinct symbol of the context-
free grammar. The variables are typed, i.e., a domain of values is associated with each distinct attribute.

Each node of the syntax tree of a valid program has a set of attributes associated with the symbol represented by that node. Boolean attributes can be used to indicate whether or not “extra-grammatical” aspects of the derivation are correct, i.e., to impose conditions on the derivation that lie outside normal context-free specification. The evaluation rules associated with the grammar's production rules determine the values of all attribute occurrences. That is, when a production rule is applied to generate a step in a language string derivation, its corresponding evaluation rules are also (logically) applied to define the values of attributes at that point in the derivation.

There are two kinds of attributes, inherited and synthesized. Inherited attributes have values defined totally in terms of attribute values of the ancestor of the nonterminal symbol. Synthesized attributes have values defined in terms of attribute values of the descendants of the corresponding nonterminal symbol. Examples of an attribute grammar and a derivation tree with a synthesized attribute are given in Figures 1.1 and 1.2 respectively.

For each production rule, there must be an evaluation rule for each synthesized attribute of the symbol on the left (the symbol being defined) and for each inherited attribute of each symbol on the right. In general, a given grammatical symbol may have both synthesized and inherited attributes, and a given attribute may be synthesized with respect to one symbol and inherited with respect to another. An example of an attribute grammar with an inherited attribute is given in Figure 1.3.
numeral : digit;

semantic
  numeral.Val := digit.Val;
end;

numeral : numeral digit;

semantic
condition
  numeral.Val <= 2,147,483,647;
end;

digit : '0';

semantic
  digit.Val := 0;
end;

... digit : '9';

semantic
  digit.Val := 9;
end;

Figure 1.1 Example Attribute Grammar
Nonterminal instance —<numeral>— Integer attribute instance

Synthesized attribute — Val: — Boolean attribute instance

Finite set

<numeral> Val: <digit> Cond:

<digit> Val: 9

<digit> Val: 0

<digit> Val: 9

Note that nonterminal and attribute instances have been defined by the derivation, but attribute values have yet to be computed.

Figure 1.2 Example Derivation Tree
string : char;
semantic
condition
  string.Size = 1;
end;

string : string2 char;
semantic
  string2.Size := string.Size -1;
end;

char : 'A';
char : 'B';
...

Figure 1.3 Example Attribute Grammar
1.4 Visit Sequences

In his original description of attribute grammars, D.E. Knuth states that semantic rules are well-defined if they are formulated in such a way that all attributes can always be defined at all nodes in any conceivable derivation tree [3]. Kastens introduces “ordered attribute grammars” as a subclass of well-defined attribute grammars, such that grammars of this class satisfy the following 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 in any derivation the attributes are evaluable in that order [2]. Furthermore, Kastens shows that for attribute grammars of this type, “visit sequences” can be derived that can drive a general purpose attribute evaluation algorithm to correctly evaluate all attributes for any valid derivation tree.

A visit sequence for an ordered attribute grammar simply formalizes the intuitive notation that if the value of attribute $x$ depends on the value of attribute $y$, then attribute $y$ must be evaluated before attribute $x$. The evaluation order defined by a visit sequence reflects all such dependencies. Kastens formulates his algorithm in somewhat vague, set-theoretic terms. Spencer describes an implementation of Kastens' attribute analysis algorithm that produces the visit sequences as one of its several outputs [5].

To understand what information a visit sequence must encode, consider the following. Evaluation of attributes proceeds as “control” is applied to a particular node. As part of the evaluation, control may be passed from the current node to its parent or one of its children. In this manner a node may receive control several times. When it receives control, it must resume execution where it left off, so it needs to remember its prior state. The purpose of
passing around control like this is to allow the evaluation of complex sets of dependencies. If node $x$ is a parent of node $y$, when control is initially passed down to node $x$, it should calculate as many of $x$'s synthesized attributes as possible. However, some attributes of $x$ may depend on attributes not yet determined. So $x$ passes control to node $y$ and other descendants which eventually calculate the values of upon which $x$'s synthesized attributes depend. Thus $y$ may return control to $x$, or pass control to one of $y$'s children, or halt if all attributes have been computed. The critical points are that when control is passed from one node to another, enough attributes have been evaluated so the node with newly granted control can proceed, and that the exchange of control eventually terminates in a state where all attributes have been assigned a value. This must be true for all possible derivations.

1.5 Spencer's Implementation of Kastens' Algorithm

In [5] Spencer describes the details of the implementation of Kastens' attribute grammar analysis algorithm, along with her design details for input/output for grammar specification and visit sequences. It is very difficult to translate Kastens' abstract algorithm design into an implementation. Kastens' algorithm describes a construction based on large abstract sets of data, different types of set operations, and multiple passes over the data. The size of the sets is determined by the number of grammar symbols, productions and symbol/attribute occurrences. As grammars increase in size, constructing and manipulating these data objects efficiently is extremely important, and is highly dependent on the data structures used to represent the sets. Spencer's implementation attempts to reduce time and space requirements by using a carefully selected sequence of set representations during
different phases of the algorithm.

Spencer's original work demonstrates the correct processing of several small attribute grammars. However, excessive compute time and space requirements of her original implementation prevents the analysis of larger attribute grammars. Furthermore, recent testing of her program on larger attribute grammars reveals that it contains bugs -- for some attribute grammars it produces visit sequences that obviously are incorrect. Thus, Spencer's program has to be fixed so it executes properly.

1.6 Converting to Java

"Java is: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language" [10]. Spencer's original implementation of analysis algorithm is in the programming language Ada. Java is the programming language chosen for the new version of Spencer's implementation of Kastens' algorithm. How to represent data is critical in the design of the attribute analysis algorithm. Java supports the object-oriented concept of class, consisting of a collection of data and methods that operate on that data [10]. Java also provides several pre-defined classes, including a class "Vector" which is basically a dynamic array. This type of data structure is ideal to represent and manipulate the large abstract sets of data in Kastens' algorithm. At runtime, the standard implementation of Vector almost completely eliminates wasted space due to the dynamic growth of the Vector. More importantly, use of a standard, predefined class and its operations helps avoid subtle programming bugs.
1.7 Thesis Overview

As noted above, EIS requires a well-formed language that defines EIS meta-data descriptions, datasets, and modeling components. The goal is to formally specify the EIS language using an attribute grammar, then use the formal specification as the basis for implementation of EIS language processing tools. Currently, a parser and semantic analyzer perform all syntactic and semantic checking. The semantic analysis done for the EIS object description language is embedded in the parser/analyser. The purpose of constructing an ordered attribute grammar for EIS, is (a) to formalize the specification of syntactic checking, (b) to formalize analysis of the static semantics specification, and (c) to formalize implementation of static semantic specification [5].

Chapter 2 discusses Kastens' algorithm, including basic notation, how the algorithm logically works, and its inputs/outputs. This method of semantic analysis is time efficient, in the sense that the evaluation order of the attributes only needs to be determined once for a given grammar. It is space efficient because the visit sequences, which can be subsequently used to evaluate the attributes for any derivation in that attribute grammar, are also constructed only once. Chapter 3 discusses the Java implementation of Kastens algorithm, including details borrowed from Spencer's program and specific features new in the Java version. Chapter 4 discusses the EIS language specification and defines the EIS language using an attribute grammar. This attribute grammar formally defines the syntactic and semantic checking that must be done for the EIS object description language. Chapter 5 analyzes the EIS attribute grammar via Kastens' algorithm implementation in Java. It also discusses a second and third version of the analysis code that was written to meet memory
requirements of large attribute grammars such as the EIS attribute grammar.

Finally, Chapter 6 discusses the correctness of the overall language design, in the sense of matching the syntactic/semantic intent for EIS. Examples are used to argue informally that the EIS attribute grammar produces computations that match the intent for object-oriented class, instance, and method specification. Chapter 6 also describes how an attribute evaluation algorithm could be constructed so that when the visit sequences produced from the analyzer with the EIS attribute grammar as input, and example derivations of meta-data descriptions, datasets, or modeling components are run through the evaluator, the results are evaluated EIS attributes.
Chapter 2

Kastens' Algorithm

2.1 Attribute Grammar Notation

The notation for attribute grammars used by Kastens is described in the following. An attribute grammar is a context-free grammar which is augmented by attributes. Semantic functions define the value of an attribute occurrence. Boolean attributes can be used to indicate whether or not “extra-grammatical” aspects of the derivation are correct, i.e., to impose conditions on the derivation that lie outside normal context-free specification.

An attribute grammar AG is defined as $AG = (G, A, VAL, SF, SC)$. $G = (N, T, S, P)$ is a context-free grammar, where $N$ is the set of nonterminal symbols, $T$ is the set of terminal symbols, $V = N \cup T$ is the vocabulary of the grammar, $S \in N$ is the start symbol, and $P$ is the set of syntactic rules. Each syntactic rule $p \in P$ has the form:

$$p = X_0 : X_1, \ldots, X_n \text{ for } n \geq 0.$$ 

$X_i$ denotes an occurrence of a symbol of $N$, for $i = 0$ and $V$ for $i > 0$. $A$ represents a set of attributes. $A_x$ is the set of attributes associated to symbol $X$. $X_{a}, X_{b}, \ldots$ denote the elements of $A_x$. $AI_x$ and $AS_x$ are subsets of $A_x$ that represent inherited and synthesized attributes respectively. $SF_p$ is the set of semantic functions associated with rule $p \in P$. Each semantic function defines the value of an attribute occurrence in $p$. These occurrences defined by semantic functions make up the set of defining occurrences, $AF_p$. Figure 2.1 demonstrates the elements of an attribute grammar.
second syntactic rule \((p \in P)\)

\[
\begin{align*}
\begin{array}{c}
\text{vocabulary } (V) \\
\text{nonterminals } (N) \\
\text{terminals } (T)
\end{array}
\end{align*}
\]

\[
p_2 : \textnormal{rule } \text{primary : ' declaration ', assignment '}'
\]

\[
\begin{align*}
\text{semantic} \\
\begin{array}{l}
\text{declaration.access := primary.access;} \\
\text{assignment.access := include }\text{(primary.access, declaration.description);} \\
\text{primary.primode := assignment.primode;} \\
\text{assignment.postmode := primary.postmode;} \\
\text{primary.evaluatable := false;} \\
\text{primary.value := undefined;}
\end{array}
\end{align*}
\]

\[
\text{the set of defining occurrences for this production are: } (AF_p)
\]

\[
\begin{array}{l}
\text{declaration.access} \\
\text{assignment.access} \\
\text{primary.primode} \\
\text{assignment.postmode} \\
\text{primary.evaluatable} \\
\text{primary.value}
\end{array}
\]

\[
\text{inherited attributes: } (AI)
\]

\[
\begin{array}{l}
\text{declaration.access} \\
\text{assignment.access} \\
\text{assignment.postmode}
\end{array}
\]

\[
\text{synthesized attributes: } (AS)
\]

\[
\begin{array}{l}
\text{primary.primode} \\
\text{primary.evaluatable} \\
\text{primary.value}
\end{array}
\]

Figure 2.1 Elements of an Attribute Grammar
2.2 The “orderness” property

D. E. Knuth proposes the concept of well-defined attribute grammars, and states that an attribute grammar is well-defined if and only if there is no sentence of the language with circularly dependent attributes [3]. Kastens goes on to introduce “ordered attribute grammars” as a subclass of well-defined attribute grammars. Grammars of this class meet the following 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 that order” [2]. Further, Kastens demonstrates that one can automatically construct algorithms to evaluate the attributes of any sentence of an ordered attribute grammar.

The problem of deciding whether a given attribute grammar is ordered is solved by projection of the attribute dependencies into dependency relations associated with production rules and symbols. The basic idea for ordered attribute grammars is: for each symbol of a given attribute grammar, construct a partial order over the attributes. This order determines the evaluation order for the attributes of a symbol, in any derivation context in which that symbol occurs. The evaluation order must reflect all direct and indirect dependencies, which may be derived from any possible context of that symbol. The evaluation order is used to construct “visit-sequences” that describe the control flow of an efficient attribute evaluation algorithm. Elements of the visit-sequence give instructions to move up to the ancestor, move down to a certain descendant, or evaluate a certain attribute.

The syntactic structure of a given terminal string generated by a grammar is depicted in Figure 2.2. During a visit to node $K_r$ some attributes of $AF_p$ are evaluated according to semantic functions of $SF_p$. Several visits to each node are generally needed until all
symbol instance

Figure 2.2 Derivation Tree
attributes are evaluated. The partial order constructed for each symbol is used to assure that the visit-sequences for a tree node and for its descendants fit together. A move down from $K_y$ to $K_x$ is made in order to evaluate a certain subset of synthesized attributes of symbol $X$. Any move back up to $K_y$ is used to evaluate a certain subset of inherited attributes of symbol $X$. Therefore the partial order for symbol $X$ must define a linear order over subsets of $A_x$, which contain alternating inherited and synthesized attributes. The order is partial because the evaluation order within each subset is not relevant.

2.3 Constructing partial orders for symbols

An attribute grammar is ordered if a partial order $DS$ (dependencies between symbols) with the properties discussed above can be constructed according to the following definitions [2]. Examples from the simple expression language given by Kastens’ [2], listed in Appendix A, are given in bold in the cases below.

**Definition 1.** Let $DP_p$ be the relation of direct dependencies between attribute occurrences associated to production rules, where

$$DP_p = \{(X_i \cdot a, X_j \cdot b) | \text{there is a semantic function in } SF_p \text{ defining } X_j \cdot b \text{ in terms of } X_i \cdot a\}$$

$$DP_2 = \{(primary\cdot access, declaration\cdot access), (primary\cdot access, assignment\cdot access), (declaration\cdot access, assignment\cdot access), (assignment\cdot primode, primary\cdot primode), (primary\cdot postmode, assignment\cdot postmode)\}$$

$DP_2$ is the set of dependencies given directly by the semantic functions.
Definition 2. Let $IDP_p$ be the relation of induced dependencies between attribute occurrences, where

$$IDP_p = DP_p \lor \{(X_i,a, X_j,b) \mid X_i \text{ occurs in rule } p, Y_j \text{ occurs in rule } q, X_i = Y_j \text{ and } (Y_j,a, Y_j,b) \in IDP_q^*\}.$$ 

$$IDP_1 = \{(\text{primary.access, primary.postmode}), (\text{primary.access, primary.primode}), (\text{primary.access, primary.value}), (\text{primary.primode, primary.postmode}), (\text{primary.postmode, primary.value}), (\text{primary.primode, primary.value}),\}$$

$IDP_1$ contains all the direct dependencies of rule 1 and those induced by attributes of similar symbol occurrences in other productions.

Definition 3. Let $IDS_X$ be the relation of induced dependencies between attribute of symbols, where

$$IDS_X = \{(X,a, X.b) \mid \text{there is an } X_x = X \text{ in a rule } p \text{ and } (X,t,a, X,t.b) \in IDP_p\}.$$ 

$$IDS_{\text{primary}} = \{(\text{primary.access, primary.postmode}), (\text{primary.access, primary.primode}), (\text{primary.access, primary.value}), (\text{primary.primode, primary.postmode}), (\text{primary.postmode, primary.value}), (\text{primary.primode, primary.value}),\}$$

$IDS_{\text{primary}}$ contains direct and induced dependencies of attributes of symbol occurrence $\text{primary}$ found in some production $p$. Figure 2.3 gives a graphical representation of $IDS_{\text{primary}}$.

If $IDS$ is cyclic, the grammar is not "ordered". In the next steps $IDS$ is completed to $DS$.

$DS_X$ defines a linear order over disjoint alternating subsets of synthesized and inherited attributes of symbol $X$. Each subset, denoted by $A_{X,k}$ consists of those attributes (synthesized or inherited) whose values are additionally available after a move up or down in the syntax tree. The evaluation order corresponds to the decreasing value of $k$. Therefore, $A_{X,k}$ contains attributes that need to be evaluated before attributes in $A_{X,k-1}$. 
Figure 2.3 Dependency graph $IDS_{\text{primary}}$
Definition 4. Let $IDS$ be acyclic. For each $X \in V$:

\[
A_{X,1} = \{ X.a \in AS \mid \text{there is no } X.b \text{ such that } (X.a, X.b) \in IDS^* \},
\]

\[
A_{X,2n} = \{ X.a \in AI \mid \text{for all } X.b \in A_X : (X.a, X.b) \in IDS^* \text{ implies } X.b \in A_X \}, \quad m < 2n \}
\]

\[
A_{X,2n+1} = \{ X.a \in AS \mid \text{for all } X.b \in A_X : (X.a, X.b) \in IDS^* \text{ implies } X.b \in A_{X,m}, m < 2n+1 \}
\]

This is done until each attribute $X.a \in A_X$ is in a disjoint partition $A_{X,k}$. The subsets are defined such that the values of $A_{X,k}$ are needed to compute the values of $A_{X,k-1}$, the values of $A_{X,k}$ are needed to compute the values of $A_{X,k-2}$, etc. Let $m_x$ equal the largest $k$ value for symbol $X$.

\[
A_{primary,1} = \{ \text{value, evaluable} \}
\]

\[
A_{primary,2} = \{ \text{postmode} \}
\]

\[
A_{primary,3} = \{ \text{primode} \}
\]

\[
A_{primary,4} = \{ \text{access} \}
\]

\[
m_{primary} = 4
\]

Definition 5. Let $IDS$ be acyclic.

\[
DS_X = IDS_X \cup \{ (X.a, X.b) \mid X.a \in A_{X,k}, X.b \in A_{X,k-1}, 2 \leq k \leq m_X \}.
\]

$DS_X$ defines a linear order over the subset $A_{X,k}$ of $A_X$. For each two attributes $X.a \in AI, X.b \in AS$, either $(X.a, X.b) \in DS_X$ or $(X.b, X.a) \in DS_X$.

Definition 6.

\[
EDP_p = DP_p \cup \{ (X.a, X.b) \mid (X.a, X.b) \in DS_X, X_i = X \text{ for each } X \text{ that is contained in } p \}.
\]

$EDP_p$ extends the dependencies of a production to reflect all dependencies (direct, induced and linearly ordered) between attributes of symbols, for the symbols contained in $p$. 

Definition 7. A given attribute grammar is "ordered" if the dependency relationship $DS$ exists and the extended dependency relationship $EDP$ is acyclic.

2.4 Visit Sequences

Just because an attribute grammar is ordered does not imply that a predefined strategy for attribute evaluation exists. An algorithm that produces such a strategy can be constructed based on the attribute dependencies as discussed above, in the form of what are known as visit sequences. Visit sequences are independent of the compilation of any particular sentence of the language; therefore they can be constructed once for a given attribute grammar as part of its analysis.
Chapter 3

Kastens’ Implementation

3.1 Attribute Grammar

The first decision Spencer makes when implementing Kastens’ algorithm is, how to represent the attribute grammar. An example of her syntax for specifying the attribute grammar is given in Appendix A. Attributes and their corresponding types are listed first, terminated with a “%”. Function names follow, terminated by a “%”. The grammar is then listed, in BNF form, with some minor syntactic rules. Each production begins with the word “rule”. A “:” follows the lefthand symbol of a production, and a “:=” follows the left hand side of a semantic function. Each production, semantic function and semantic condition must terminate with a “;”. The word “semantic” must proceed the list of semantic functions, the word “condition” must proceed the list of semantic conditions, and the word “end” must terminate each production. All nonterminals must be enclosed by single quotes. Symbol/attribute occurrences are represented by “symbol.attribute”.

3.2 Data Structures

Data is constantly being manipulated throughout Kastens’ algorithm. How to represent this data is critical. Spencer uses the programming language Ada and data structures including: arrays, records, and pointers. These data structures are easy to manipulate, but the size of datasets places maximum values on the number of symbols, attributes, productions and symbol/attribute occurrences allowed in the grammar. This
creates a problem in attempting to analyze very large grammars. In addition, since Spencer’s program uses statically allocated arrays, allocated space is wasted on smaller grammars. Due to the uncertain size of the grammar ahead of time, Spencer’s implementation [5] is not as efficient as we would like it to be. In addition, Ada programming environments are becoming somewhat rare, so the decision was made to re-implement Kastens’ algorithm in the more portable language Java.

Java is an object-oriented programming language. A class is a collection of data and methods that operate on that data. Java comes with a large number of predefined classes. One of those predefined classes is Vector, which implements a variable sized list of objects. In this case, an object is some instance of another class. The methods associated with the class Vector allow you to store and retrieve objects of any type, as well as to easily manipulate and keep track of the size of the Vector. Thus, our reimplementation is based on Spencer’s implementation, but with data structures converted into more appropriate Java forms.

Java also has many other nice features. It is relatively easy to learn. It is an interpreted language. The Java compiler generates byte-codes for the Java Virtual Machine (JVM) (instead of the native machine code) which executes the compiled byte-codes. Java byte-codes are platform independent. Therefore Java programs can run on any platform that the JVM has been ported to. Java is designed for writing robust software. There are no pointers, which eliminates one of the most bug-prone aspects of other programming languages. There is extensive compile-time type checking. There are many more advantages to using Java; however, those listed above are the most important in why the language was chosen for this project.
A symbol table, attribute table, production table, and symbol/attribute occurrence maps are the initial data structures created to implement Kastens' algorithm. The Java version is based on the following class definitions. The class `symbol` represents a grammar symbol:

```java
public class symbol {
    String sym_name;
    int sym_base;
}
```

The variable `symbol.sym_name` holds the string representation of the lexical token. A Vector `sym_table` represents a symbol table. A unique integer is associated with each symbol that is given by the index of `sym_table`. Production rules can be recursive, i.e., `numeral : numeral2 '+' digit`. For those symbols that have an integer attached at the end, `symbol.sym_base` holds the unique integer representation of the symbol without the integer attached. For those symbols without an integer attached to the end, `symbol.sym_base` holds the unique integer representation of that symbol.

The class `attribute` represents an attribute:

```java
public class attribute {
    String att_name;
    String att_type;
    public boolean check_type() { ... }
}
```

The variable `attribute.att_name` holds the string representation of the lexical token that represents the attribute name. The variable `attribute.att_type` holds the string representation of the lexical token that represents the attribute type. The Vector `att_table` represents an attribute table. A unique integer is associated with each attribute that is given by the index of `att_table`. The method `attribute.check_type()` determines if a particular attribute type is
Symbol/attribute occurrences are represented in two maps. As a semantic function is being parsed, if the symbol/attribute occurrence did not previously exist it is assigned a unique integer value (starting at 1). \textit{map1} is a one dimensional array of \textit{maprec}. \textit{maprec} is the following class:

\begin{verbatim}
public class maprec {
    int sym;
    int att;
}
\end{verbatim}

The index of \textit{map1} represents a unique integer for a particular symbol/attribute occurrence. \textit{maprec} contains the unique integer representation of the symbol for that occurrence in the variable \textit{maprec.sym}, and the unique integer representation of the attribute for that occurrence in the variable \textit{maprec.att}. \textit{map2} is a two dimensional array whose indices (the integer representation of a symbol, and the integer representation of an attribute) yield the unique integer representation for that occurrence. These are the only two arrays used in the Java implementation of Kastens' algorithm. There is a maximum limit of 500 symbol/attribute occurrences. The ease of manipulating these arrays became more of a priority than the small amount of wasted space allocated. Figure 3.1 shows the symbol table, attribute table, occurrence map 1, and occurrence map 2 for the simple attribute grammar listed in Appendix B.

The class \textit{prod} represents a production:

\begin{verbatim}
public class prod {
    int lhs;
    Vector sym_list = new Vector();
    Vector occur_list = new Vector();
    Vector cond_list = new Vector();
    Vector vis_seq = new Vector();
}
\end{verbatim}
Figure 3.1 Example Symbol Table, Attribute Table, Occurrence Map1 and Occurrence Map2
The variable \textit{prod.lhs} holds the integer for the nonterminal symbol representing the left hand side of the production. \textit{prod.sym_list} is a Vector containing the integer representation of all the symbols in the production. \textit{prod.occur_list} is a Vector containing "occur"(s) (holds the occurrence arguments for the function definition of an occurrence). \textit{prod.cond_list} is a Vector containing "cond"(s) (holds the occurrence arguments for a condition). And finally, \textit{prod.vis_seq} is a Vector containing "seq"(s) (holds an action for the visit sequence). The Vector \textit{prod_table} is created to represent a production table. Figure 3.2 shows the production table for the attribute grammar listed in Appendix B immediately after the grammar has been parsed.

The main data structure in Kastens' algorithm represents dependency relations. Dependencies are easily represented in adjacency matrices. Logically, matrix(i,j) = 1 indicates that j depends on i, whereas matrix(i,j) = 0, indicates that there is no dependency. In the Java version, Vectors are used to simulate and replace Spencer's adjacency matrices. A Vector of Vectors takes the place of a two-dimensional matrix.

\subsection*{3.3 Implementation}

Before we actually begin implementing Kastens' algorithm we must take an attribute grammar with the correct syntax as input and create a symbol table, attribute table, production table and occurrence maps as discussed above. Additionally, a function table, lists of all attributes \((A)\), inherited attributes \((AI)\), and synthesized attributes \((AS)\) for each symbol must be defined as well as defining occurrences \((AF)\) for each production. These data structures are referred to throughout the entire program.
### Figure 3.2 Example Production Table

<table>
<thead>
<tr>
<th>production</th>
<th>left hand nonterminal symbol</th>
<th>Vector of symbols</th>
<th>Vector of arguments for semantic function</th>
<th>Vector of semantic functions</th>
<th>Vector of conditions</th>
<th>Vector of actions for visit sequences</th>
<th>Vector of arguments for condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 2 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 3 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The class `grammar` was created to hold all of the data structures associated with an attribute grammar.

```java
class grammar {
    Vector att_table = new Vector(); // attribute table
    Vector sym_table = new Vector(); // symbol table
    Vector prod_table = new Vector(); // production table
    Vector fun_table = new Vector(); // function table
    occmaps omaps = new occmaps(); // contains maps for occurrences
    attsets aset = new attsets(); // contains A, AI and AS as well as AF
    Vector tdp = new Vector(); // contains dependencies for each production
    Vector tds = new Vector(); // contains dependencies for each symbol
    Vector mark = new Vector(); // temporary variable
    Vector partition = new Vector(); // contains disjoint partitions of occurrences
    Vector f = new Vector(); // contains the smallest even number >=k (partition for each symbol)
    Vector vseq = new Vector(); // contains the visit sequences for each production
}
```

Values in the variables `tdp`, `tds`, `mark`, `partition`, `f`, and `vseq` are constructed in the rest of the algorithm to hold dependency relations, partitions and visit sequences.

Dependency relations between attribute occurrences in productions as well as between attributes of symbols are the basis for computing the visit sequences for a given attribute grammar. If at any point a dependency relationship is found to be cyclic, that particular attribute grammar is not ordered. Each rule in the attribute grammar is represented by a dependency relation \(TDP_p\) over attribute occurrences in that production. Each symbol is represented by a dependency relation \(TDS_x\) over attributes \(A_x\). Several functions are used in the next steps for updating dependency relations.

```java
add_arc_trans(Vector am, int size, int v1, int v2)
```

adds the dependency "\(v2 \text{ depends on } v1\)" to the adjacency matrix `am`, and then adds any additional dependencies needed to implement the closure on `am`. 


add_arc_induce(Vector mark, Vector tdp, Vector am, Vector tds, int v1, int v2, occmaps occ, Vector sym_table)

adds the dependency "v2 depends on v1," and then adds any additional dependencies needed to implement the closure on am. This function is applied to the relation TDP_p.

Additionally, each new dependency added is also added to TDS_X, along with additional dependencies needed to reach the transitive closure on TDS_X, if the symbols of symbol/attribute occurrence v1 and v2 are the same.

The following steps convert the recursive definitions of DP_p, IDP_p and EDP_p listed in Chapter 3 into iterative algorithms that compute their transitive closures. The first step computes DP+. Below is an outline of the method create_tdp_and_tds.

create_tdp_and_tds(grammar g) {
    for each production p
        loop
            for each semantic function f ∈ SF_p defining X, b
                loop
                    for each argument X, a of f
                        loop
                            if (X, a, X, b) ∈ TDP_p
                                then    add_arc_induce(TDP_p, X, a, X, b)
                            fi
                        repeat
                    repeat
                repeat
            repeat
        }
create_idp(grammar g) {
    while there is a dependency \((X.a, X.b)\) in TDS not marked
    loop
        mark(X.a, X.b)
        for each occurrence \(X_i\) of \(X\) in any rule \(p\)
        loop
            if \((X_i.a, X_i.b) \not\in \text{TDP}_p\)
                then add_arc_induce(\text{TDP}_pX_i.aX_i.b)
            fi
        repeat
    repeat
}

Each dependency in TDS which is not marked is induced at each occurrence of the symbol in TDP. If new dependencies are found that need to be induced, they are added to TDS by add_arc_induce. When the algorithm is completed \(TDP = IDP^+\) where \(IDP\) is the set of all induced dependencies (including direct dependencies) between attribute occurrences. \(IDP^+\) ensures that all attribute dependencies for a symbol \(X\) are obtained for any context of \(X\). Marking the dependency in TDS ensures that no dependency is unnecessarily induced more than once. \(TDS = IDS^+\) where \(IDS\) is the set of all induced dependencies (including direct dependencies) between attributes of symbols. The variables \(tdp\) and \(tds\) in the Java implementation hold the dependency relations for a given attribute grammar. Figure 3.3 shows the dependencies graphs \(TDP\) and \(TDS\) at this point for the example attribute grammar.

The third step computes the disjoint partitions of \(Ax\). Starting with symbol 0 and \(k=1\), the algorithm loops until all attributes \(A\) are assigned to some \(Ax_k\). Partitions with odd \(k\) contain only synthesized attributes. Partitions with even \(k\) contain only inherited attributes.

create_partition(grammar g) {
    for each symbol \(X\)
    loop
        \(k = 1;\)
        \(\text{not}\_\text{assigned} = Ax\)
        while (\(\text{not}\_\text{assigned} \neq \text{empty}\))
Figure 3.3 Dependency graphs TDP and TDS
loop
    found_one = false;
    for each attribute $X.a \in (\text{not\_assigned} \&\& \text{if } odd \ k \text{ then } AS_{X} \text{ else } AI_{X} \text{ fi})$
    loop
        condition_holds = true;
        for each $X.b \in \text{not\_assigned}$
        loop
            if $(X.a, X.b) \in TDS_{X}$
                then condition_holds = false;
                break;
            fi
        repeat
        if condition_holds
            then partition$(X.a) = k$;
                not_assigned = not_assigned \ {X.a};
                found_one = true;
                break;
            fi
        repeat
        if (!found_one \&\& not_assigned $\neq 0$)
            then $k = k + 1$;
        fi
    repeat
    $m_{X} = k$;
    $f_{X} = \text{if } (odd \ k) \text{ then } k+1 \text{ else } k \text{ fi}$
    repeat
The algorithm loops for each symbol of the attribute grammar. $k$ is initially 1. The variable not\_assigned contains all the attributes associated with symbol $X$. If $k$ is odd and an attribute $X.a$ is synthesized and an element of not\_assigned, then the algorithm determines if any other element in not\_assigned depends on $X.a$. In the actual Java implementation a Vector partition, whose index is the integer representation of that occurrence is assigned the value of $k$. A Vector $f$, whose index is the integer representation of a symbol is assigned the smallest even number $\geq k$. Figure 3.4 shows the variables partition and $f$ for the attribute grammar in Appendix B.

The next step computes the relation $EDP^{*}$. The algorithm adds dependencies to $TDP$ according to the relation given by the disjoint partitions of the attribute occurrences for each
NOTE: All occurrences with symbols whose base value differs from the integer representation are given the value of 0

Figure 3.4 Disjoint Partitions and F values
create_edp(grammar g) {
    for each production p loop
        for each symbol X in p loop
            X = X,
            for each X.a loop
                for each X.b loop
                    if partition(X.a) > partition(X.b) then
                        add_arc_trans(TDP_pX,a,X,b)
                    fi
                repeat
            repeat
    repeat
} repeat
repeat
repeat
repeat

When the algorithm is completed $TDP = EDP^+$. If each $TDP_p$ is acyclic, then the attribute grammar is ordered.

The final step of Kastens’ algorithm constructs the visit-sequences. Consider evaluating the attributes of symbol $X$ where $f_X = 4$ and the largest value of $k = 4$. $A_{X,4}$ are those inherited attributes evaluated first. A move to a descendant must be made and then the synthesized attributes $A_{X,3}$ are evaluated and so forth.

$A_{X,4}$ ! 1 ! 1 ! 2 ! 2

$A_{X,3}$ $A_{X,2}$ $A_{X,1}$

The number of ancestor and descendant visits are both $f_X \text{ div } 2$.

An occurrence is created to represent a visit. This makes it easy to keep track of dependencies between occurrences and visits. If a production contains the symbol in the example above, two occurrences would be created to represent the two visits needed. The
integer representation of the symbol (of the occurrence), represents the symbol to be visited. The integer representation of the attribute represents the value of \( k \) (in the form of \( k + \) number of attributes). Due to the fact the value of the attribute of the occurrence is greater than the total number of attributes, we know the occurrence is a visit.

Conditions are also represented as an additional occurrence. The integer representation of the symbol (of the occurrence) represents the number of the condition (in the form of \( \text{cond} + \) number of symbols). Due to the fact the value of the symbol of the occurrence is greater that the total number of symbols, we know the occurrence is a condition. The integer representation of the attribute has no relevant value. Figure 3.5 show the occurrences maps of the attribute grammar in Appendix B after the visit values and conditions have been added.

### 3.3.1 Creating Visit Sequences

The following algorithm presented by Kastens and implemented by Spencer is intended to construct the visit sequences:

```plaintext
create_visseq(grammar g) {
    for each production p
        loop
            for each \((X_r, a, X_f, b) \in TDP_p\)
                loop
                    \(mi = \text{partition}(X_r, a)\);
                    \(mj = \text{partition}(X_f, b)\);
                    \(ki = (f_u - mi + 1) \div 2\);
                    \(kj = (f_u - mi + 1) \div 2\);
                    if \((ki > 0 \& \& kj > 0)\)
                        then add_arc_trans(VSp, if \(X_r, a \in AF_p, then X_r, a \text{ else } v_{ki, fi}\), (if \(X_f, b \in AF_p, then X_f, b \text{ else } v_{kj, fi}\))
                        fi
                repeat
            add_cond_vertices_to_vs();
        for each \(g \in \Delta p\)
    }
```
Occurrence Map 1

symbol 0  

attribute 0  

occurrence 1  
0 0  

occurrence 2  
1 0  

occurrence 3  
2 0  

occurrence 4  visit symbol 0  
0 2  
k = 1 (2 - # of attributes)  

occurrence 5  visit symbol 1  
1 2  
k = 1 (2 - # of attributes)  

occurrence 6  visit symbol 2  
2 2  
k = 1 (2 - # of attributes)  

occurrence 7  
4 1  
not relevant  

condition 1 (4 - # of symbols)  

Figure 3.5 Example Occurrence Map 1 after visit values and condition are added
The algorithm takes the relation $TDP$, and for each dependency determines whether the occurrences for that dependency are inherited and can be evaluated immediately, i.e., if $(k_i > 0 \&\& k_j > 0)$. If the occurrences can be, this dependencies is not added to the new dependency relation $VS$. If they can’t, the dependency is added to the new dependency relation $VS$. If an occurrence is not in the defining occurrence set $AF_p$, then the occurrence value of visiting the given symbol with the given k value is determined. Therefore, $VS$ contains dependencies between occurrences of the defining occurrence set, and dependencies between occurrences of the defining occurrence set and visit values (which are represented by occurrence values).

A couple of changes were made to the above algorithm in the Java implementation. The statement if $(k_i > 0 \&\& k_j > 0)$ implies if two occurrence’s k values are not greater than zero they can be evaluated immediately and there is no need to add the dependency to the relation $VS$. This isn’t correct. The second occurrence depends on the first occurrence regardless of the k value. Therefore, the first occurrence must be evaluated before the second occurrence and this dependency must show up in the list of visit sequences unless the first
occurrence has a k value less than or equal to zero. The Java implementation changed the statement to \( \text{if} (k_i > 0) \), meaning if the occurrence that is depended on has a value less than or equal to zero, its occurrence value is available immediately so the occurrence that depended on it can be evaluated immediately also and the dependency does not need to be added to \( VS \).

The second change was the positioning of the procedure \textit{add_cond_vertices_to_vs}. \textit{add_cond_vertices_to_vs} adds the dependencies found in conditions to \( VS \). As mentioned before, conditions are represented as occurrences. The occurrence value of the condition depends on the occurrence values of the arguments of the condition. Therefore \textit{add_cond_vertices_to_vs} was moved to the beginning of \textit{create_visseq} and conditions are treated just like any other occurrences. If a condition depends on an occurrence with a k value equal to zero, the dependency does not need to be added to \( VS \) because the occurrence value is available immediately and the condition can be evaluated immediately.

The final part of the algorithm arbitrarily adds dependencies to \( VS \) until it is linearly ordered, ensuring that the last move to the ancestor is the last element of the visit sequence by making it depend on all other occurrences (regular or visit). The final change made to Kastens' algorithm has to do with evaluating occurrences after a move up or down a derivation tree. All dependencies are reflected in the dependency relation \( VS \). However when a move is made up or down the tree there is nothing to indicate that available attributes should be evaluated at that moment before any other move takes place. The available attributes and the move may not depend on one another but attributes in the node visited may depend on evaluated previous attributes. The Java implementation corrects this problem by
comparing occurrences in the defining, visit, and condition occurrence set of a production. If two occurrences have no dependency and one of them is a visit, a dependency is added to $VS$ where the visit depends on the other occurrence. This ensures all available attributes and conditions will be evaluated before a move up to an ancestor or down to a descendant. The rest of the occurrences that have no dependencies between them are evaluated arbitrarily.

### 3.4 Problems with Spencer’s Implementation

Spencer did an excellent job of creating data structures and manipulating them throughout her implementation of Kastens’ algorithm. However, the excessive compute time and space required by the data structures in the analysis algorithm prevent her implementation from use with larger attribute grammars. In fact, Spencer’s implementation has one major mistake that is easily overlooked with smaller attribute grammars.

The problem occurs in the creation of the dependency relation $TDS$. As mentioned before, the procedure `add_arc_induce` adds the dependency “$v_2$ depends on $v_1$,” and then updates $TDP_p$ and $TDS_x$ appropriately. Spencer’s implementation of `add_arc_induce` calls a procedure `ADD_TO_TDS`.

```plaintext
procedure ADD_TO_TDS
(TDS : in out ADJ_MATRIX_PTR_TYPE;
V1,V2 : in OCCURRENCE.OCCUR_VALUES) is
begin
SYM1,SYM2 : INTEGER;
ATT1,ATT2 : INTEGER;
TEMP_PTR : ADJ_MATRIX_PTR_TYPE;
TEMP_V1,TEMP_V2 : INTEGER;
SYM1 := OCCURRENCE.LOOKUP_SYM(OCCURRENCE.MAP1,V1);
SYM2 := OCCURRENCE.LOOKUP_SYM(OCCURRENCE.MAP1,V2);
```
In Spencer's procedure `ADD_TO_TDS` occurrence values $V1$ and $V2$ are to be added to the adjacency matrix $TDS$ if the occurrences share the same symbol. However, a problem occurs in lines 12 - 16. In lines 12 and 13 the symbol for occurrence $V1$ and occurrence $V2$ are found. Lines 13 and 14 determine the base values of the symbols found in lines 12 and 13. If the base values are the same and the attributes are not the same the dependency is added to $TDS$.

This procedure will produce circular dependencies when two occurrences have different symbols yet share the same base symbol, e.g., $expression2.postmode := expression.primode$. $TDS$ is supposed to contain dependencies between attributes of symbols. $expression2$ and $expression$ share the same base symbol, $expression$, but do not share the
same instance of the symbol \textit{expression}, so no such dependency should get added to the relation $TDS_{expression}$. This mistake is easily overlooked, because many grammars do not have constructs like one discussed above, especially small grammars. This problem is easy to correct once it is discovered and traced back, by making sure two occurrences share the same symbol, not the same base symbol, when creating the dependency relation $TDS$. 
Chapter 4

The EIS Attribute Grammar

4.1 Overview of EIS

The Ecosystem Information System has two major components. First, EIS is a software system that supports a particular set of operations that are used to create, access, and share a distributed data repository. The database is partitioned among a number of host machines. The potential database user does not need to be concerned with which machine the data is physically located. He or she only needs to be aware that there exists a database “out there” somewhere in the global information space accessible via the network, and that the EIS software system is the tool that permits access to this database. The second component of EIS is that it is a data repository organized hierarchically using an object-oriented framework. The object-oriented approach is relatively simple, inherently hierarchical, and easily extensible.

The EIS data repository is represented by a hierarchical structure known as a class hierarchy. At each primary point in the hierarchy is a class definition, which represents a meta-description of a particular type of dataset. The meta-description includes both the description of data attributes and the description of operational components that are used to access, give values to, and manipulate the data attributes. Also included in the hierarchy, attached to particular class nodes, are class instances that represent datasets of that type. Finally, also attached to class nodes are class methods that represent program components.
that implement an operation defined for that class. Figure 4.1 shows an example of a class hierarchy through the EIS interface.

This object-oriented approach to data modeling places primary emphasis upon the data objects in terms of the attributes of those objects that are most relevant in the application domain [8]. Identifying critical relationships between classes allows the development of the class hierarchy. Figure 4.2 represents an EIS hierarchy. Class A is the root of the hierarchy. Class A is extended by the subclasses B, and C. B and C have all the properties of their parent class, A, plus one or more new properties. Class B and class C are specializations of class A, while class A is a generalization of classes B and C.

Class B and class C inherit the operations “read” and “display” from their parent class A. Inherited properties need not be defined in a class specification; only newly defined properties need to be specified in the class interface. Instance X is an instance of “B” and any ancestor of “B”, including “A”. Therefore, the principle of attribute inheritance provides an effective means to organize data on the basis of shared properties. Dataset instances that are similar to one another will be found closer together in the hierarchy, while instances that are dissimilar will be located further apart.

Data transformations or operations, have two components: an operation specification (i.e., its name, argument types and return type), and an operation method (i.e., program). Only the operation specification is part of the class interface. The operation specification in the interface of class “A” indicates that “read” takes no arguments, returns no value, and that is defined for all the classes shown. Two operation methods provide implementation for this operation specification -- one implementation for each subclass. Therefore, clients need not
Figure 4.1 Example of EIS Interface
CLASS A
Services:
read()
display()

CLASS B
Services:
filter()
merge()

CLASS C
Services:
label()
get_state()

INSTANCE X
METHOD
read()
....

INSTANCE Y
METHOD
read()
...

Figure 4.2 An EIS Hierarchy
be aware of low-level details of operation execution. The implementation of "read" can be changed without affecting clients that use instances of "A" [8].

4.2 The EIS Language

Each node in an EIS hierarchy has its own description in a syntax specified by the EIS language. The syntax is different for a class, method or instance. The EIS language also describes the syntax of the whole hierarchy, which mainly consists of the concatenation of the syntax of the nodes in the hierarchy in an ordered form.

4.2.1 EIS Classes

The production rule for a class definition is shown in Figure 4.3. "class", "of" and "end_class" are terminals or tokens of the EIS language. "class_defn", "id1", "id2", "interface_uses_section", ... etc. are all nonterminals. As implied by the production rule, the class specification allows for much more information than just a class name. Figure 4.4 shows the EIS interface for constructing a class.

4.2.1.1 Class Attributes

The EIS class specification syntax allows the definition of one or more properties within a class definition. These properties denote characteristics of the class, and can be categorized as state variables, constants, types or functions. These properties are specified by the EIS user by clicking on the "Class Attributes" button. (See Figure 4.4) State variables represent the data associated with any instance of the class. Every state variable has a
rule classdefn : '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
    'endclass'

Figure 4.3 Production rule for a class definition
Figure 4.4 Interface for creating a Class
particular type, for example:

```
VAR var1 OF integer
VAR var2 OF char
```

Constants can be defined by the EIS user to provide alternative names for values. For example:

```
CONST con1 : string := "Trish"
CONST con2 : boolean := false
```

The EIS language supports several data types and type constructors. The predefined simple types are "integer", "real", "char", "string", and "boolean". The type constructors are "array", "record", "set" and "enumeration". The EIS user can construct a structured type from the simple types or structured types themselves. For example:

```
TYPE type1 := integer
TYPE type2 := (id1, id2, id3)
TYPE type3 := array [1..10] OF real
```

As mentioned before, data transformation functions, have two components: a function specification (i.e., its name, argument types and return type), and a function method (i.e., executable program). Only the function specification is part of the class interface. For example:

```
FUNCTION func1 (char, real) : integer
```

### 4.2.1.2 Class Interface

The EIS user can specify classes in the interface-uses section by clicking on the "Class Interface" button. (See Figure 4.4) This section lists all the classes upon which the definition of the current class relies. Ancestor class properties are automatically inherited, so interface-
uses is generally used to list only non-ancestor class dependencies.

4.2.1.3 Class Parameter Declarations

*Class parameterization* allows the EIS user to formulate meaningful class hierarchies, in a manner analogous to formal argument declarations in function specification. The formal parameters for a class can be of type *class, type, constant or function*. The EIS user can declare parameters by clicking on the "Class Parameter Declarations" button. (See Figure 4.4) The following is an example of some parameter declarations:

```
param1 : class
param2 : type
```

4.2.1.4 Inherited Parameter Bindings

Once a parameter has been declared, it must eventually be bound to an actual class, type, function or constant. We can specify an actual parameter value for a formal parameter in an instance or subclass of a parameterized class. The EIS user can assign parameters by clicking on the "Inherited Parameter Bindings" button. (See Figure 4.4) The following is an example of some parameter assignments:

```
param1 := Erdas_Lan_Class
param2 := char
```

4.2.1.5 State Variable Bindings

State variables defined in a parent can also be bound in an instance or subclass. This binding is interpreted as providing an initial value for the state variable in question. The EIS
user can bind state variables by clicking on the "State Variable Bindings" button. (See Figure 4.4) The following is an example of binding state variables:

\[
\text{flag := true} \\
\text{a := 15.02}
\]

4.2.1.6 Documents and Keywords

The EIS user can specify the location of documents related to the current EIS object, or put short documentation information within the object specification itself by clicking on the "Documents" button. (See Figure 4.4) The "Keywords" button is used to specify keywords for EIS entities to support more ambitious network search functionality.

Figures 4.5 and 4.6 show the EIS interface for creating an instance and method respectively. The components of the instance and method differ from the components of the class, so the syntax of the description of EIS instances and EIS methods differs from that of EIS classes. However, most of the components listed in the instance or method description can be found as components in the class description. Therefore, the syntax of these individual components is the same as those found in the class description but the syntax for the instance, method, and class objects as a whole are not the same.

4.3 Semantic Checking in EIS

We have just seen what the EIS language looks like, the syntax of the language. Now let's take a look at what it means, the semantics of the language. As mentioned above, the EIS language supports the definition of properties, interface-use, class parameterization,
Figure 4.5 Interface for creating an Instance
Figure 4.6 Interface for creating a Method
parameter and state variable binding, property inheritance, etc. In order to use the EIS language appropriately, constraints must be satisfied. Below is a list of the semantic checking that must be done to construct a well-formed class hierarchy in EIS [6].

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 of type \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 type 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 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 $C_x$, 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 $H_p$, then it should be defined as $H_p C_i$ 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 $I_x$.

In the current version of EIS, a parser and semantic analyzer performs all the syntactic checking in an ad hoc manner. Initially, there was no formal definition of the conditions listed above. An attribute grammar was created to formalize the condition checking, and replace the ad hoc implementation embedded in the parser/analizer.

4.4 The EIS Attribute Grammar

The first attribute grammar for EIS was built several years ago as part of this thesis. Vijayant Palaiya did an implementation based on that language specification [6]. He also implemented a few grammatical changes, due to request by EIS users for modified syntactic and semantic aspects. The EIS language has thus evolved into a language with a more complete syntactic structure and a more extensive specification of static semantics. Description of the newest EIS language specification, based on the newest EIS attribute grammar completes the thesis project presented here.

As background each semantic constraint in the EIS language can be formally specified by a boolean attribute and evaluation rules defined by the attribute grammar. Whether or not a semantic condition is met is determined by the evaluation of a boolean attribute during a derivation: true indicates the constraint is met, and false indicates that the constraint is not met.

The EIS Attribute Grammar is divided into an upper part and a lower part. The upper part of the attribute grammar defines attributes appropriate to the structure of a whole class
hierarchy, and uses global attributes to perform the semantic checking based on parent-child, ancestor-descendant, and interface-uses relationships. The lower part of the attribute grammar defines attributes appropriate to the structure of individual hierarchy nodes, and uses local attributes to perform semantic checking on local uses of identifiers. Key local values are also passed to the upper part of the attribute grammar. Figure 4.7 is an EIS hierarchy that is used as an example throughout the rest of this chapter.

The lower part of the attribute grammar constructs a symbol table (the attribute SymTab) for each node, storing the name of all identifiers defined within the node, their type, and other relevant information. Identifiers include the names of classes, instances, methods, state variables, constants, types, functions and parameters. Figure 4.8 shows the attributed derivation tree for the node that represents class “A” in our example. The tree illustrates the computation of attribute values, as well as those values that are used to check the semantic constraints specified by the grammar. Every attribute in the derivation is synthesized.

The nonterminal “classdefn” has a key attribute called SymTab. SymTab represents the symbol table for the class node “A” in the EIS hierarchy. SymTab contains the identifier definitions for that node. The values of SymTab are computed by semantic rules in the descendants of “classdefn”.

The nonterminal “functiondefn” has an attribute called SymRec. SymRec represents a symbol table record, and consists of a 4-tuple (Name, Type, TypeDenoter, InList). Name is the name of the identifier in the symbol table. In our example Name has the value “compute”, which is the name of the function. Type is the type of property the identifier represents. The identifier represents a function so Type has the value of FUNC. TypeDen
class A of null
Class A
  function compute() : real
end_class

class B of A
Class B
  variable i of integer
end_class

class C of A
Class C
  const c : string := "Tree"
end_class

Figure 4.7 EIS Hierarchy
Figure 4.8 Attributed Tree for class “A”
represents the return type of the function, which can either be a primitive type or a constructed type. In the case of a constructed type, this attribute refers to a symbol table record, which contains information of the constructed type. InList refers to the list of argument types, which can also be a primitive type or a constructed type. If the return type is not a primitive or constructed type, the condition istype(id2.Tag), evaluates to false which indicates the function definition is not legal. If the name of the function is qualified, the condition notqualified(id1.Tag), evaluates to false which also indicates the function definition is not legal.

Eventually symbol table records from different property definitions of class “A” are combined to form the SymTab attribute as shown in Figure 4.9. Each property defined locally within class “A” must be locally unique. The condition uniquenessymtabentries(SymTab), checks for uniqueness of names of the identifiers. The attributed trees for classes “B” and “C” are shown in Figures 4.10 and Figure 4.11 respectively.

The upper part of the attribute grammar has an important synthesized attribute SynST. SynST is associated with every node in the hierarchy, containing the symbol table of the node itself and the symbol tables of all descendant nodes. Each symbol table in SynST is represented by (Name, Type, SymTab), where Name is the name of the node, Type is the type of the node (“class”, “instance”, or “method”), and SymTab is the symbol table of that node in the hierarchy. The lower part of the attribute grammar computes the values for individual SymTab entities. Only the root node of the hierarchy contains the attribute GbST. GbST contains the symbol tables of all the objects in the hierarchy. The condition validate() uses the global symbol table to check the semantic correctness of the whole hierarchy definition.
rule classdefn : 'class' id1 'of' id2
  interfaceusessection
  forwarddeclsection
  bindparamsection
desc param section
description
mixeddeclclist
bindstvaresection
keywordssection
documentsection
'endclass';

semantic
  classdefn.Name := id1.Tag;
  classdefn.Desc := description.Tag;
  classdefn.SymTab := append((bindparamsection.SymRecList, declparamsection,
                              SymRecList), forwarddeclsection.SymRecList, interfaceusessection.SymRec
List,
                              mixeddeclclist.SymRecList, bindstvaresection.SymRecList);
  classdefn.KeyList := keywordssection.KeyList;
  classdefn.Info :=
    (classdefn.Name, C, (classdefn.Parent, classdefn.Desc, classdefn.KeyList,
                          classdefn.DocList));

condition
  uniquesymtabentries(classdefn.SymTab);
  classdefn.Parent = id2.Tag;
end;

Figure 4.9 Attribute Grammar Specification for a “classdefn”
Figure 4.10 Attributed Tree for class "B"
Figure 4.11 Attributed Tree for class "C"
Figure 4.12 shows the attributed derivation tree for the upper part of the EIS hierarchy in our example.
Figure 4.8

Figure 4.9

Figure 4.10

Figure 4.11

Figure 4.12 Attributed Tree for Upper Part of EIS Hierarchy
Chapter 5

Execution Results

5.1 A Simple Example

The simple attribute grammar listed in Appendix B was taken from Pagan [4]. The grammar defines an integer constant. With the single attribute "val", the attribute grammar ensures that no syntactically correct numeral can exceed 32 bits. The grammar has twelve productions, two symbols, one attribute, and three symbol/attribute occurrences.

The visit sequences for the attribute grammar are listed below. For each production in the grammar, a visit sequences is given. There are three possible actions in a visit sequence, move to a nonterminal, evaluate a symbol/attribute occurrence, or evaluate a condition. Moving to a nonterminal is indicated by the word "MOVE" followed by the nonterminal to be visited, followed by the number of times that nonterminal has been visited within that particular sequence. Evaluating a symbol/attribute occurrence is indicated by the word "EVAL" followed by the symbol/attribute occurrence. Evaluating a condition is indicated by the word "COND" followed by the number of the condition to be evaluated.

| Production: 0 | MOVE DIGIT 1 | EVAL NUMERAL.VAL | MOVE NUMERAL 1 |
| Production: 1 | MOVE DIGIT 1 | MOVE NUMERAL2 1 | EVAL NUMERAL.VAL | COND 1 | MOVE NUMERAL 1 |
| Production: 2 | EVAL DIGIT.VAL | MOVE DIGIT 1 |
| Production: 3 | EVAL DIGIT.VAL |

Sat Sep 12 19:39:46 PDT 1998

***VISIT SEQUENCES***

67
It is easy to look at this grammar and the results and determine the visit sequences are correct. That is, they provide a way to correctly evaluate all attributes for any valid derivation tree. The runtime for this particular attribute grammar was approximately 1 second.

5.2 A More Complicated Example

The attribute grammar listed in Appendix A was taken directly from Kastens [2]. The grammar is a simple expression language with nine productions, eight attributes, eight symbols, and twenty-three symbol/attribute occurrences. This grammar provides us a means to verify that our implementation is correct, since the visit sequences (listed after the attribute grammar in Appendix A) match up with those derived by Kastens [2]. The runtime for this particular attribute grammar was approximately one minute and twenty seconds, a significant increase over the attribute grammar in Appendix B. This is due to the greater number of symbol/attribute occurrences. As the number of symbol/attribute occurrences increase the
time to manipulate the datasets increases exponentially.

5.3 The EIS Attribute Grammar

The EIS Attribute Grammar (listed in Appendix C) contains twenty-one attributes, seventy-nine symbols, one hundred productions, and one hundred twenty-nine symbol/attribute occurrences. A machine with one hundred twenty-eight megabytes of RAM could not meet the memory requirements of running the analyzer with the EIS attribute grammar.

A second version of the analyzer was written to accommodate very large attribute grammars. In the new version, data originally stored in three-dimensional Vectors in memory is now written as a group of files, where each file contains a two-dimensional Vector. This version of the analyzer works correctly, however the runtime increases dramatically. For example, the simple attribute grammar listed in Appendix B took one minute and twenty seconds to run with this version, i.e., approximately one minute and nineteen seconds longer than the first version.

The more complex attribute grammar listed in Appendix A took eight hours, forty-one minutes and fifty-one seconds with the new version, approximately eight hours, forty minutes and thirty-one seconds longer than the original version. By looking at the results it was obvious the EIS attribute grammar would take weeks to run through the analyzer. The time to produce the needed visit sequences was not practical. Therefore a third version of the analyzer was written.

As mentioned before, version 2 stores data as a two dimensional Vector in file. To
reduce the amount of time needed to maintain a two dimensional Vector, version 3 stores data as a one dimensional Vector. All the information is still maintained, just in a different data structure. The simple attribute grammar listed in Appendix B took one minute and two seconds to run. The more complex attribute grammar listed in Appendix A took seven hours, thirty-three minutes and fifty seconds to run. The performance of version 3 is significantly better than that of version 1, yet not enough to be used for practical purposes on large attribute grammars. Figure 5.1 summarizes the execution results.

<table>
<thead>
<tr>
<th></th>
<th>Version 1</th>
<th>Version 2</th>
<th>Version 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix B</td>
<td>1 sec.</td>
<td>1 min. 20 sec.</td>
<td>1 min. 2 sec.</td>
</tr>
<tr>
<td>Appendix A</td>
<td>1 min. 20 sec.</td>
<td>8 hours 41 min. 51 sec.</td>
<td>7 hours 33 min. 50 sec.</td>
</tr>
</tbody>
</table>

Figure 5.1 Summary of Execution Results

Due to time constraints, a fourth implementation was never written. After analyzing the dependency relations of several attribute grammars it is noted that only a small portion of the dependency graphs are marked with a dependency. A possible solution to the memory problem could be to just keep track of the marked dependencies. A large portion of the analyzer would have to be rewritten if a new data structure was used. Most of the procedures in the analyzer access or manipulate the data structures that represent the dependency relations.
5.4 Dividing the EIS Attribute Grammar

The EIS attribute grammar was divided into four sections. Each section was run through the analyzer individually. The break points were productions that contained only synthesized attributes and control only needed to be passed to descendants once. Dividing a grammar up in such a way has no affect on the final visit sequences, decreases the run time exponentially, and allows us to analyze a large attribute grammar. Each section of the EIS attribute grammar was successfully run through the analyzer. The visit sequences for each section are listed after the attribute grammar in Appendix C.
Chapter 6
Analysis of Results

6.1 The EIS Attribute Grammar

The EIS hierarchy in Figure 4.7 was derived and attributes were evaluated according to the visit sequences produced by the analysis algorithm. Figures 6.1, 6.2, 6.3, and 6.4 show the evaluation order from left to right. For example, in Figure 6.1 a move is made from the symbol `classdefn` to its descendant `id`. Another move is made from `id` to the terminal `A`. When `id` receives control again attribute `id.Tag` is evaluated and a move is made up to its ancestor `classdefn`. The attribute `classdefn.Name` is evaluated and control is passed down to the nonterminal `mixeddecllist`. Attributes listed in the figures without a symbol are assumed to be synthesized.

Every attribute and condition in the derivation tree is evaluated correctly. All dependencies are reflected in the visit sequences. All constraints (listed in section 4.3) intended for the hierarchy are met: all class names are unique, each property defined locally within a class is locally unique, the arguments and return value of a function is a class, basic type or constructed type.

6.2 Attribute Evaluator

An attribute evaluator must be implemented to efficiently evaluate the attributes for any given derivation. The work from this thesis provides a critical piece of data for an
Figure 6.1 Evaluation of class “A”
Figure 6.2 Evaluation of class "B"
Figure 6.3 Evaluation of class “C”
Figure 6.1 Figure 6.2 Figure 6.3 Figure 6.4 Evaluation of Upper Part of EIS Hierarchy
attribute evaluator, the visit sequences. Kastens [2] explains four ways an attribute evaluator could be implemented: using coroutines, recursive procedures, stack automaton, or finite automaton. Constructing an attribute evaluator can be a possible thesis project for a future computer science student.

6.3 Conclusion

In conclusion, the EIS attribute grammar gives a formal definition of both the syntactic and semantic checking that must be done to process the EIS object description language. With an efficient attribute evaluator the formal specification can be used for implementation of EIS language processing tools. The EIS attribute grammar is well-defined, ordered, and meets the intent of the EIS user.
Appendix A

```
access set
description strseq
primode string
postmode string
evaluable boolean
value string
id string
val string
%
include
identify
isdefined
widen
add
%

rule program : primary;
  semantic
    primary.access := 0;
    primary.postmode := primary.primode;
  end;

rule primary : '(' declaration assignment ');
  semantic
    declaration.access := primary.access;
    assignment.access := include(primary.access,declaration.description);
    primary.primode := assignment.primode;
    assignment.postmode := primary.postmode;
    primary.evaluable := false;
    primary.value := undefined;
  end;

rule primary : identifier;
  semantic
    primary.primode := identify(identifier.id,primary,access);
    primary.evaluable := false;
    primary.value := undefined;
  condition
    isdefined(identifier.id,primary.access);
  end;

rule primary : intconstant;
  semantic
    primary.primode := int;
    primary.evaluable := true;
    primary.value := if primary.postmode = real
        then widen(intconstant.value) else intconstant.value fi;
  end;
```

78
rule primary : realconstant;
  semantic
    primary.primode := real;
    primary.evaluable := true;
    primary.value := realconstant.value;
  end;

rule assignment : identifier '!=' expression;
  semantic
    expression.access := assignment.access;
    assignment.primode := identify(identifier.id, assignment.access);
    expression.postmode := assignment.primode;
  condition
    isdefined(identifier.id, assignment.access) and not (expression.primode = real and expression.postmode = int);
  end;

rule expression : expression2 '+' primary;
  semantic
    expression2.access := expression.access;
    primary.access := expression.access;
    expression.primode := if expression2.primode = int and primary.primode = int then int else real fi;
    expression2.postmode := expression.primode;
    primary.postmode := expression.primode;
    expression.evaluatable := expression2.evaluatable and primary.evaluatable;
    expression.value := if expression.evaluatable then add(expression2.value, primary.value) else undefined fi;
  end;

rule expression : primary;
  semantic
    primary.access := expression.access;
    primary.postmode := expression.postmode;
    expression.primode := primary.primode;
    expression.evaluatable := primary.evaluatable;
    expression.value := primary.value;
  end;

rule declaration : 'new' identifier '!=' expression;
  semantic
    expression.access := declaration.access;
    declaration.description := (identifier.id, expression.primode);
    expression.postmode := expression.primode;
  end;
***VISIT SEQUENCES***

Production: 0
EVAL PRIMARY.ACCESS
MOVE PRIMARY 1
EVAL PRIMARY.POSTMODE
MOVE PRIMARY 2
MOVE PROGRAM 1

Production: 1
EVAL DECLARATION.ACCESS
MOVE DECLARATION 1
EVAL ASSIGNMENT.ACCESS
MOVE ASSIGNMENT 1
EVAL PRIMARY.PRIMODE
MOVE PRIMARY 1
EVAL ASSIGNMENT.POSTMODE
EVAL PRIMARY.EVALUABLE
EVAL PRIMARY.VALUE
MOVE ASSIGNMENT 2
MOVE PRIMARY 2

Production: 2
MOVE IDENTIFIER 1
EVAL PRIMARY.PRIMODE
COND 1
MOVE PRIMARY 1
EVAL PRIMARY.EVALUABLE
EVAL PRIMARY.VALUE
MOVE PRIMARY 2

Production: 3
EVAL PRIMARY.PRIMODE
MOVE PRIMARY 1
EVAL PRIMARY.EVALUABLE
MOVE INTCONSTANT 1
EVAL PRIMARY.VALUE
MOVE PRIMARY 2

Production: 4
EVAL PRIMARY.PRIMODE
MOVE PRIMARY 1
EVAL PRIMARY.EVALUABLE
MOVE REALCONSTANT 1
EVAL PRIMARY.VALUE
MOVE PRIMARY 2

Production: 5
EVAL EXPRESSION.ACCESS
MOVE IDENTIFIER 1
EVAL ASSIGNMENT.PRIMODE
MOVE EXPRESSION 1
EVAL EXPRESSION.POSTMODE
COND 1
MOVE ASSIGNMENT 1
MOVE EXPRESSION 2
MOVE ASSIGNMENT 2
Production: 6
EVAL PRIMARY.ACCESS
EVAL EXPRESSION2.ACCESS
MOVE PRIMARY 1
MOVE EXPRESSION2 1
EVAL EXPRESSION.PRIMODE
EVAL PRIMARY.POSTMODE
EVAL EXPRESSION2.POSTMODE
MOVE PRIMARY 2
MOVE EXPRESSION 1
MOVE EXPRESSION2 2
EVAL EXPRESSION.EVALUABLE
EVAL EXPRESSION.VALUE
MOVE EXPRESSION 2
Production: 7
EVAL PRIMARY.ACCESS
MOVE PRIMARY 1
EVAL EXPRESSION.PRIMODE
MOVE EXPRESSION 1
EVAL PRIMARY.POSTMODE
MOVE PRIMARY 2
EVAL EXPRESSION.EVALUABLE
EVAL EXPRESSION.VALUE
MOVE EXPRESSION 2
Production: 8
EVAL EXPRESSION.ACCESS
MOVE EXPRESSION 1
EVAL EXPRESSION.POSTMODE
MOVE IDENTIFIER 1
EVAL DECLARATION.DESCRIPTION
MOVE EXPRESSION 2
MOVE DECLARATION 1
Sat Sep 12 19:42:11 PDT 1998
Appendix B

val    string
%
%
rule numeral : digit;
semantic
    numeral.val := digit.val;
end;

rule numeral : numeral2 digit;
semantic
    numeral.val := 10 * numeral2.val + digit.val;
condition
    numeral.val <= 2147483647;
end;

rule digit : '0';
semantic
    digit.val := 0;
end;

rule digit : '1';
semantic
    digit.val := 1;
end;

rule digit : '2';
semantic
    digit.val := 2;
end;

rule digit : '3';
semantic
    digit.val := 3;
end;

rule digit : '4';
semantic
    digit.val := 4;
end;

rule digit : '5';
semantic
    digit.val := 5;
end;

rule digit : '6';
semantic
    digit.val := 6;
end;
rule digit : '7';
  semantic
    digit.val := 7;
  end;

rule digit : '8';
  semantic
    digit.val := 8;
  end;

rule digit : '9';
  semantic
    digit.val := 9;
  end,
***VISIT SEQUENCES***

Production: 0
MOVE DIGIT 1
EVAL NUMERAL.VAL
MOVE NUMERAL 1

Production: 1
MOVE DIGIT 1
MOVE NUMERAL 2 1
EVAL NUMERAL.VAL
COND 1
MOVE NUMERAL 1

Production: 2
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 3
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 4
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 5
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 6
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 7
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 8
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 9
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 10
EVAL DIGIT.VAL
MOVE DIGIT 1

Production: 11
EVAL DIGIT.VAL
MOVE DIGIT 1
Appendix C

EIS Attribute Grammar

GbST setseq
Info strseq
Parent string
SynST setseq
SymTab set
Name string
Desc strseq
KeyList set
Doc set
DocList setseq
SymRec set
SymRecList setseq
Tag string
IdList set
PType string
InList setseq
InPair set
Type string
Val string
SVVal int
Len int
%
add
exp
div
validate
append
disjoint
uniquesymtabentries
addfwdcllist
notqualified
addintuselist
adddparamdecl
addbindparams
addbindstvars
addtypeofn
addvardefn
findtype
addconstantdefn
addfunctiondefn
getentry
istype
isprimitivetypen
addargdecl
addenumeratedtypeofn
addarraytypeofn
addrecordtypeofn
addsettypeofn
addenumvalid
addidtypefromidlist
isdetecttype
nonnull
concat
lookup
%
rule rootnode : classlist;
semantic
    rootnode.GbST := classlist.SynST;
    rootnode.Info := classlist.Info;
    rootnode.Parent := null;
    classlist.Parent := root;
condition
    validate(rootnode.GbST);
end;

rule classlist : classlist2 classnode;
semantic
    classlist.Info := append(classlist2.Info,classnode.Info);
    classlist2.Parent := classlist.Parent;
    classnode.Parent := classlist.Parent;
condition
    disjoint(classlist2.SynST,classnode.SynST);
end;

rule classlist : ' ';
semantic
    classlist.SynST := <>;
    classlist.Info := <>;
end;

rule instancelist : instancelist2 instancenode;
semantic
    instancelist.Info := append(instancelist2.Info,instancenode.Info);
    instancelist2.Parent := instancelist.Parent;
    instancenode.Parent := instancelist.Parent;
condition
    disjoint(instancelist2.SynST,instancenode.SynST);
end;

rule instancelist : ' ';
semantic
    instancelist.SynST := <>;
    instancelist.Info := <>;
end;

rule methodlist : methodlist2 methodnode;
semantic
methodlist.Info := append(methodlist2.Info,methodnode.Info);
methodnode.Parent := methodlist.Parent;

condition
disjoint(methodlist2.SynST,methodnode.SynST);
end;

rule methodlist : ');
semantic
methodlist.SynST := <>;
methodlist.Info := <>;

end;

rule classnode : 'classnode' classdefn instancelist methodlist classlist 'endclassnode';
semantic
classnode.SynST := append((classdefn.Name,C,classdefn.SymTab),
instancelist.SynST,methodlist.SynST,classlist.SynST);
classnode.Info := append(classdefn.Info,instancelist.Info,methodlist.Info,classlist.Info);
classdefn.Parent := classnode.Parent;
classlist.Parent := classdefn.Name;
instancelist.Parent := classdefn.Name;
methodlist.Parent := classdefn.Name;

condition
disjoint((classdefn.Name,C,classdefn.SymTab),instancelist.SynST,
methodlist.SynST,classlist.SynST);
end;

rule instancenode : instancedefn;
semantic
instancenode.SynST := (instancedefn.Name,I,instancedefn.SymTab);
instancenode.Info := instancedefn.Info;

condition
instancenode.Parent = instancedefn.Parent;
end;

rule methodnode : methoddefn;
semantic
methodnode.SynST := (methoddefn.Name,M,-);
methodnode.Info := methoddefn.Info;

condition
methodnode.Parent := methoddefn.Parent;
end;

rule classdefn : 'class' id 'of' id2 interfaceusessection forwarddeclsection bindparamsection
declparamsection description mixeddecllist bindstvarsection keywordssection documentsection 'endclass';
semantic
classdefn.Name := id.Tag;
classdefn.Desc := description.Tag;
classdefn.SymTab := append((bindparamsection.SymRecList,declparamsection.SymRecList),
forwarddeclsection.SymRecList,interfaceusessection.SymRecList,
mixeddecllist.SymRecList,bindstvarsection.SymRecList);
classdefn.KeyList := keywordssection.KeyList;
classdefn.Info := (classdefn.Name,C,(classdefn.Parent,classdefn.Desc,
classdefn.KeyList,classdefn.DocList));

condition
    uniquesymtabentries(classdefn.SymTab);
classdefn.Parent = id2.Tag;
end;

rule instancedefn : 'instance' id 'of' id2 bindparamsection description bindstvarsection keywordssection
documentsection;
semantic
    instancedefn.Name := id.Tag;
    instancedefn.Parent := id2.Tag;
    instancedefn.Desc := description.Tag;
    instancedefn.SymTab := append(bindparamsection.SymRecList,bindstvarsection.SymRecList);
    instancedefn.KeyList := keywordssection.KeyList;
    instancedefn.Info := (instancedefn.Name, I, (instancedefn.Parent,
        instancedefn.Desc,instancedefn.KeyList,instancedefn.DocList));
end;

rule methoddefn : 'method' id 'of' id2 description keywordssection documentsection;
semantic
    methoddefn.Name := id.Tag;
    methoddefn.Parent := id2.Tag;
    methoddefn.Desc := description.Tag;
    methoddefn.KeyList := keywordssection.KeyList;
    methoddefn.Info := (methoddefn.Name, M, (methoddefn.Parent,
        methoddefn.Desc,methoddefn.KeyList,methoddefn.DocList));
end;

rule forwarddeclsection : '^
semantic
    forwarddeclsection.SymRecList := <>;
end;

rule forwarddeclsection : 'forwarddecl' identifierlist 'endforwarddecl';
semantic
    forwarddeclsection.SymRecList := addfwddcllist(identifierlist.IdList);
condition
    notqualified(identifierlist.IdList);
end;

rule interfaceusessection : '^
semantic
    interfaceusessection.SymRecList := <>;
end;

rule interfaceusessection : 'interfaceuses' identifierlist 'endinterfaceuses';
semantic
    interfaceusessection.SymRecList := addintuselist(identifierlist.IdList);
condition  
    notqualified(identifierlist.IdList);
end;

rule declparamsection : ' ';
semantic  
    declparamsection.SymRecList := <>;
end;

rule declparamsection : 'paramdecl' paramdecllist 'endparamdecl';
semantic  
    declparamsection.SymRecList := paramdecllist.SymRecList;
end;

rule paramdecllist : id ':' paramtype;
semantic  
    paramdecllist.SymRecList := addparamdecl(id.Tag,paramtype.PType);
end;

rule paramdecllist : paramdecllist2 ':' id ':' paramtype;
semantic  
    paramdecllist.SymRecList := append(paramdecllist2.SymRecList, 
        addparamdecl(id.Tag,paramtype.PType));
condition  
    disjoint(paramdecllist2.SymRecList, addparamdecl(id.Tag,paramtype.PType));
end;

rule paramtype : 'CLASS';
semantic  
    paramtype.PType := CLASS;
end;

rule paramtype : 'TYPE';
semantic  
    paramtype.PType := TYPE;
end;

rule paramtype : 'CONST';
semantic  
    paramtype.PType := CONST;
end;

rule paramtype : 'FUNCTION';
semantic  
    paramtype.PType := FUNCTION;
end;

rule bindparamsection : ' ';
semantic  
    bindparamsection.SymRecList := <>;
end;
rule bindparamsection: 'parambind' bindparamlist 'endparambind';
  semantic
    bindparamsection.SymRecList := bindparamlist.SymRecList;
  end;

rule bindparamlist: id ':=' id2;
  semantic
    bindparamlist.SymRecList := addbindparams(id.Tag, getentry(id2.Tag));
  end;

rule bindparamlist: bindparamlist2 ',' id ':=' id2;
  semantic
    bindparamlist.SymRecList := append(bindparamlist2.SymRecList, 
                              addbindparams(id.Tag, getentry(id2.Tag)));
  end;

rule mixeddecllist: ';';
  semantic
    mixeddecllist.SymRecList := <>;
  end;

rule mixeddecllist: mixeddecl ';' mixeddecllist2;
  semantic
    mixeddecllist.SymRecList := append(mixeddecllist2.SymRecList, mixeddecl.SymRecList);
  condition
    disjoint(mixeddecllist2.SymRecList, mixeddecl.SymRecList);
  end;

rule mixeddecl: typedefn;
  semantic
    mixeddecl.SymRecList := typedefn.SymRecList;
  end;

rule mixeddecl: vardefn;
  semantic
    mixeddecl.SymRecList := vardefn.SymRecList;
  end;

rule mixeddecl: constantdefn;
  semantic
    mixeddecl.SymRecList := constantdefn.SymRecList;
  end;

rule mixeddecl: functiondefn;
  semantic
    mixeddecl.SymRecList := functiondefn.SymRecList;
  end;

rule bindstvarsection: ' '; 
  semantic
    bindstvarsection.SymRecList := <>;
  end;
rule bindstvarsection : 'bindstvar' bindstvarlist 'endbindstvar';
  semantic
    bindstvarsection.SymRecList := bindstvarlist.SymRecList;
end;

rule bindstvarlist : bindstvarlist2 ';'; id '=' value;
  semantic
    bindstvarlist.SymRecList := append(bindstvarlist2.SymRecList,
           addbindstvars(id.Tag,value.Tag,getentry(value.Type)));
end;

rule bindstvarlist : id '=' value;
  semantic
    bindstvarlist.SymRecList := addbindstvars(id.Tag,value.Tag,getentry(value.Type));
end;

rule typedefn : 'type' id ':' typedenoter;
  semantic
    typedefn.SymRecList := addtypedefn(id.Tag,typedenoter.SymRec);
  condition
    notqualified(id.Tag);
end;

rule vardefn : 'var' identifierlist 'of' typedenoter;
  semantic
    vardefn.SymRecList := addvardefn(identifierlist.IdList,typedenoter.SymRec);
  condition
    notqualified(identifierlist.IdList);
end;

rule constantdefn : 'const' id ':' id2 ':=' value;
  semantic
    constantdefn.SymRecList := addconstantdefn(id.Tag,fmdtype(id2.Tag),value.Tag);
  condition
    isprimitivetype(id2.Tag);
    id2.Type = value.Type;
    notqualified(id.Tag);
end;

rule functiondefn : 'function' id '(' arglist ')' ':' id2;
  semantic
    functiondefn.SymRecList := addfunctiondefn(id.Tag,arglist.SymRecList,getentry(id2.Tag));
  condition
    istype(id2.Tag);
    notqualified(id.Tag);
end;

rule arglist : ' ';
  semantic
    arglist.SymRecList := <>;
end;
rule arglist : argdcl;
  semantic
    arglist.SymRecList := argdcl.SymRec;
  end;

rule arglist : arglist1 ',' argdcl;
  semantic
    arglist.SymRecList := append(arglist.SymRecList,argdcl.SymRec);
  end;

rule argdcl : typedenoter;
  semantic
    argdcl.SymRec := addargdcl(typedenoter.SymRec);
  end;

rule typedenoter : id;
  semantic
    typedenoter.SymRec := if (lookup(id.Tag) = FALSE)
      then (id.Tag,UNRSLVD,NULL,NULL) else getentry(id.Tag) fi;
  end;

rule typedenoter : newtype;
  semantic
    typedenoter.SymRec := newtype.SymRec;
  end;

rule newtype : enumeratedtype;
  semantic
    newtype.SymRec := addenumeratedtype(enumeratedtype.SymRecList);
  end;

rule newtype : arraytype;
  semantic
    newtype.SymRec := addarraytype(arraytype.SymRec,arraytype.InList);
  end;

rule newtype : recordtype;
  semantic
    newtype.SymRec := addrecordtype(recordtype.SymRecList);
  end;

rule newtype : settype;
  semantic
    newtype.SymRec := addsettype(settype.SymRec);
  end;

rule enumeratedtype : '(' identifierlist ')';
  semantic
    enumeratedtype.SymRecList := addenumvalid(identifierlist.IdList);
  condition
    notqualified(identifierlist.IdList);
  end;
rule recordtype : 'recordstart' fieldlist 'recordend';
  semantic
    recordtype.SymRecList := fieldlist.SymRecList;
  end;

rule fieldlist : recordsection;
  semantic
    fieldlist.SymRecList := recordsection.SymRecList;
  end;

rule fieldlist : fieldlist2 ',' recordsection;
  semantic
    fieldlist.SymRecList := append(fieldlist2.SymRecList, recordsection.SymRecList);
  condition
    disjoint(fieldlist2.IdList, recordsection.IdList);
  end;

rule recordsection : identifierlist ':' typedenoter;
  semantic
    recordsection.SymRecList := addidtypefromidlist(identifierlist.IdList, typedenoter.SymRec);
  condition
    notqualified(identifierlist.IdList);
  end;

rule arraytype : 'array' '[' indextypelist ']' 'of' typedenoter;
  semantic
    arraytype.SymRec := typedenoter.SymRec;
    arraytype.InList := indextypelist.InList;
  end;

rule indextypelist : indextype;
  semantic
    indextypelist.InList := indextype.InPair;
  end;

rule indextypelist : indextypelist2 ',' indextype;
  semantic
    indextypelist.InList := append(indextypelist2.InList, indextype.InPair);
  end;

rule indextype : lowerbound '..' upperbound;
  semantic
    indextype.InPair := (lowerbound.Tag, upperbound.Tag);
  end;

rule lowerbound : value;
  semantic
    lowerbound.Tag := value.Tag;
  condition
    isdiscretetype(value.Tag);
  end;
rule lowerbound : id;
  semantic
    lowerbound.Tag := id.Tag;
  condition
    isdiscretetype(id.Tag);
end;

rule upperbound : value;
  semantic
    upperbound.Tag := value.Tag;
  condition
    isdiscretetype(value.Tag);
end;

rule upperbound : id;
  semantic
    upperbound.Tag := id.Tag;
  condition
    isdiscretetype(id.Tag);
end;

rule settype : 'set' 'of' basetype;
  semantic
    settype.SymRec := basetype.SymRec;
end;

rule basetype : id;
  semantic
    basetype.SymRec := getentry(id.Tag);
end;

rule basetype : enumeratedtype;
  semantic
    basetype.SymRec := addenumeratedtype(enumeratedtype.SymRecList);
end;

rule keywordssection : 'keywords' keywordslist 'endkeywords';
  semantic
    keywordssection.KeyList := keywordslist.KeyList;
end;

rule keywordssection : ' '*;
  semantic
    keywordssection.KeyList := < >;
end;

rule keywordslist : string;
  semantic
    keywordslist.KeyList := string.Tag;
end;
rule keywordslist : keywordslist2 ',' string;
semantic
    keywordslist.KeyList := append(keywordslist2.KeyList,string.Tag);
condition
    disjoint(keywordslist2.KeyList,string.Tag);
end;

rule documentsection : ' ';
semantic
    documentsection.DocList := <>;
end;

rule documentsection : 'documents' documentdefnlist 'enddocuments';
semantic
end;

rule documentdefnlist : documentdefn;
semantic
end;

rule documentdefnlist : documentdefnlist2 ';' documentdefn;
semantic
condition
    disjoint(documentdefnlist2.DocList,documentdefn.Doc);
end;

rule documentdefn : 'documentnameloc' id string;
semantic
    documentdefn.Doc := (id.Tag,string.Tag);
end;

rule documentdefn : 'documentation' string;
semantic
    documentdefn.Doc := (NULL,string.Tag);
end;

rule value : sign unsignednumber;
semantic
    value.Tag := concat(sign.Tag,unsignednumber.Tag);
    value.Type := unsignednumber.Type;
end;

rule value : unsignednumber;
semantic
    value.Tag := unsignednumber.Tag;
    value.Type := unsignednumber.Type;
    value.Val := unsignednumber.Val;
end;
rule value : string;
  semantic
    value.Tag := string.Tag;
    value.Type := STR;
  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 unsignednumber : unsignedinteger;
  semantic
    unsignednumber.Tag := unsignedinteger.Tag;
    unsignednumber.Val := unsignedinteger.Val;
    unsignednumber.Type := INT;
  end;

rule unsignednumber : unsignedreal;
  semantic
    unsignednumber.Tag := unsignedreal.Tag;
    unsignednumber.Val := unsignedreal.Val;
    unsignednumber.Type := REAL;
  end;

rule unsignedreal : unsignedinteger fractionalpart;
  semantic
    unsignedreal.Tag := concat(unsignedinteger.Tag,concat(".",fractionalpart.Tag));
  end;

rule unsignedinteger : DIGITSEQUENCE;
  semantic
    unsignedinteger.Tag := DIGITSEQUENCE.Tag;
    unsignedinteger.Val := DIGITSEQUENCE.Val;
  end;

rule fractionalpart : DIGITSEQUENCE;
  semantic
    fractionalpart.Tag := DIGITSEQUENCE.Tag;
    fractionalpart.Len := DIGITSEQUENCE.Len;
    fractionalpart.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 identifierlist : id;
  semantic
    identifierlist.IdList := id.Tag;
end;

rule identifierlist : identifierlist2 , id;
  semantic
    identifierlist.IdList := append(identifierlist2.IdList, id.Tag);
  condition
    disjoint(identifierlist2.IdList.id.Tag);
end;

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

rule character : CHARACTERTOKEN;
  semantic
    character.Tag := CHARACTERTOKEN.Tag;
end;
rule boolean : TRUETOKEN;
semantic
    boolean.Tag := TRUETOKEN.Tag;
    boolean.Val := TRUE;
end;

rule boolean : FALSETOKEN;
semantic
    boolean.Tag := FALSETOKEN.Tag;
    boolean.Val := FALSE;
end;
** ***VISIT SEQUENCES***

Production: 0
EVAL ROOTNODE.PARENT
EVAL CLASSLIST.PARENT
MOVE CLASSLIST 1
EVAL ROOTNODE.GBST
EVAL ROOTNODE.INFO
COND 1
MOVE ROOTNODE 1

Production: 1
EVAL CLASSLIST2.PARENT
EVAL CLASSNODE.PARENT
MOVE CLASSLIST2 1
MOVE CLASSNODE 1
EVAL CLASSLIST.SYNST
EVAL CLASSLIST.INFO
COND 1
MOVE CLASSLIST 1

Production: 2
EVAL CLASSLIST.SYNST
EVAL CLASSLIST.INFO
MOVE CLASSLIST 1

Production: 3
EVAL INSTANCENAMELIST2.PARENT
EVAL INSTANCENAMENODE.PARENT
MOVE INSTANCENAMELIST2 1
MOVE INSTANCENAMENODE 1
EVAL INSTANCENAMELIST.SYNST
EVAL INSTANCENAMELIST.INFO
COND 1
MOVE INSTANCENAMELIST 1

Production: 4
EVAL INSTANCENAMELIST.SYNST
EVAL INSTANCENAMELIST.INFO
MOVE INSTANCENAMELIST 1

Production: 5
EVAL METHODLIST2.PARENT
EVAL METHODNODE.PARENT
MOVE METHODLIST2 1
MOVE METHODNODE 1
EVAL METHODLIST.SYNST
EVAL METHODLIST.INFO
COND 1
MOVE METHODLIST 1

Production: 6
EVAL METHODLIST.SYNST
EVAL METHODLIST.INFO
MOVE METHODLIST 1

Production: 7
EVAL CLASSDEFN.PARENT
MOVE CLASSDEFN 1
EVAL INTERFACEUSESSECTION.SYMRECLIST
MOVE INTERFACEUSESSECTION 1
Production: 14
MOVE IDENTIFIERLIST 1
EVAL INTERFACEUSESSECTION.SYMRECLIST
COND 1
MOVE INTERFACEUSESSECTION 1
Production: 15
EVAL DECLPARAMSECTION.SYMRECLIST
MOVE DECLPARAMSECTION 1
Production: 16
MOVE PARAMDECLLIST 1
EVAL DECLPARAMSECTION.SYMRECLIST
MOVE DECLPARAMSECTION 1
Production: 17
MOVE ID 1
MOVE PARAMTYPE 1
EVAL PARAMDECLLIST.SYMRECLIST
MOVE PARAMDECLLIST 1
Production: 18
MOVE ID 1
MOVE PARAMTYPE 1
MOVE PARAMDECLLIST2 1
EVAL PARAMDECLLIST.SYMRECLIST
COND 1
MOVE PARAMDECLLIST 1
Production: 19
EVAL PARAMTYPE.PTYPE
MOVE PARAMTYPE 1
Production: 20
EVAL PARAMTYPE.PTYPE
MOVE PARAMTYPE 1
Production: 21
EVAL PARAMTYPE.PTYPE
MOVE PARAMTYPE 1
Production: 22
EVAL PARAMTYPE.PTYPE
MOVE PARAMTYPE 1
Production: 23
EVAL BINDPARAMSECTION.SYMRECLIST
MOVE BINDPARAMSECTION 1
Production: 24
MOVE BINDPARAMLIST 1
EVAL BINDPARAMSECTION.SYMRECLIST
MOVE BINDPARAMSECTION 1
Production: 25
MOVE ID 1
MOVE ID2 1
EVAL BINDPARAMLIST.SYMRECLIST
MOVE BINDPARAMLIST 1
Production: 26
MOVE ID 1
MOVE ID2 1
MOVE BINDPARAMLIST2 1
EVAL BINDPARAMLIST.SYMRECLIST
MOVE BINDPARAMLIST 1
Production: 27
EVAL BINDSTVARSECTION.SYMRECLIST
MOVE BINDSTVARSECTION 1
Production: 28
MOVE BINDSTVARLIST 1
EVAL BINDSTVARSECTION.SYMRECLIST
MOVE BINDSTVARSECTION 1
Production: 29
MOVE ID 1
MOVE BINDSTVARLIST2 1
MOVE VALUE 1
EVAL BINDSTVARSECTION.SYMRECLIST
MOVE BINDSTVARLIST 1
Production: 30
MOVE ID 1
MOVE VALUE 1
EVAL BINDSTVARLIST.SYMRECLIST
MOVE BINDSTVARLIST 1
Production: 31
MOVE KEYWORDSLIST 1
EVAL KEYWORDSSECTION.KEYLIST
MOVE KEYWORDSSECTION 1
Production: 32
EVAL KEYWORDSSECTION.KEYLIST
MOVE KEYWORDSSECTION 1
Production: 33
MOVE STRING 1
EVAL KEYWORDSLIST.KEYLIST
MOVE KEYWORDSLIST 1
Production: 34
MOVE STRING 1
MOVE KEYWORDSLIST2 1
EVAL KEYWORDSLIST.KEYLIST
COND 1
MOVE KEYWORDSLIST 1
Production: 35
EVAL DOCUMENTSECTION.DOCLIST
MOVE DOCUMENTSECTION 1
Production: 36
MOVE DOCUMENTDEFNLIST 1
EVAL DOCUMENTSECTION.DOCLIST
MOVE DOCUMENTSECTION 1
Production: 37
MOVE DOCUMENTDEFN 1
EVAL DOCUMENTDEFNLIST.DOCLIST
MOVE DOCUMENTDEFNLIST 1
Production: 38
MOVE DOCUMENTDEFN 1
MOVE DOCUMENTDEFNLIST2 1
EVAL DOCUMENTDEFNLIST.DOCLIST
COND 1
MOVE DOCUMENTDEFNLIST 1
Production: 39
MOVE ID 1
MOVE STRING 1
EVAL DOCUMENTDEFN.DOC
MOVE DOCUMENTDEFN 1
Production: 40
MOVE STRING 1
EVAL DOCUMENTDEFN.DOC
MOVE DOCUMENTDEFN 1
Sat Sep 12 20:59:02 PDT 1998
EVAL BINDSTVARLIST. SYMRECLIST
MOVE BINDSTVARLIST 1
Production: 9
MOVE KEYWORDSLIST 1
EVAL KEYWORDSSECTION.KEYLIST
MOVE KEYWORDSSECTION 1
Production: 10
EVAL KEYWORDSSECTION.KEYLIST
MOVE KEYWORDSSECTION 1
Production: 11
MOVE STRING 1
EVAL KEYWORDSLIST.KEYLIST
MOVE KEYWORDSLIST 1
Production: 12
MOVE STRING 1
MOVE KEYWORDSLIST2 1
EVAL KEYWORDSLIST.KEYLIST
COND 1
MOVE KEYWORDSLIST 1
Production: 13
EVAL DOCUMENTSECTION.DOCLIST
MOVE DOCUMENTSECTION 1
Production: 14
MOVE DOCUMENTDEFNLIST 1
EVAL DOCUMENTSECTION.DOCLIST
MOVE DOCUMENTSECTION 1
Production: 15
MOVE DOCUMENTDEFN 1
EVAL DOCUMENTDEFNLIST.DOCLIST
MOVE DOCUMENTDEFNLIST 1
Production: 16
MOVE DOCUMENTDEFN 1
MOVE DOCUMENTDEFNLIST2 1
EVAL DOCUMENTDEFNLIST.DOCLIST
COND 1
MOVE DOCUMENTDEFNLIST 1
Production: 17
MOVE ID 1
MOVE STRING 1
EVAL DOCUMENTDEFN.DOC
MOVE DOCUMENTDEFN 1
Production: 18
MOVE STRING 1
EVAL DOCUMENTDEFN.DOC
MOVE DOCUMENTDEFN 1
Production: 19
MOVE UNSIGNEDNUMBER 1
EVAL VALUE.TYPE
MOVE SIGN 1
EVAL VALUE.TAG
EVAL VALUE.VAL
MOVE VALUE 1
Production: 20
MOVE UNSIGNEDNUMBER 1
EVAL VALUE.TAG
EVAL VALUE.TYPE
EVAL VALUE.VAL
MOVE VALUE 1
Production: 21
EVAL VALUE.TYPE
MOVE STRING 1
EVAL VALUE.TAG
MOVE VALUE 1
Production: 22
EVAL VALUE.TYPE
MOVE CHARACTER 1
EVAL VALUE.TAG
MOVE VALUE 1
Production: 23
EVAL VALUE.TYPE
MOVE BOOLEAN 1
EVAL VALUE.TAG
EVAL VALUE.VAL
MOVE VALUE 1
Production: 24
EVAL UNSIGNEDNUMBER.TYPE
MOVE UNSIGNEDINTEGER 1
EVAL UNSIGNEDNUMBER.TAG
EVAL UNSIGNEDNUMBER.VAL
MOVE UNSIGNEDNUMBER 1
Production: 25
EVAL UNSIGNEDNUMBER.TYPE
MOVE UNSIGNEDREAL 1
EVAL UNSIGNEDNUMBER.TAG
EVAL UNSIGNEDNUMBER.VAL
MOVE UNSIGNEDNUMBER 1
Production: 26
MOVE UNSIGNEDINTEGER 1
MOVE FRACTIONALPART 1
EVAL UNSIGNEDREAL.TAG
EVAL UNSIGNEDREAL.VAL
EVAL UNSIGNEDREAL.LEN
MOVE UNSIGNEDREAL 1
Production: 27
MOVE DIGITSEQUENCE 1
EVAL UNSIGNEDINTEGER.TAG
EVAL UNSIGNEDINTEGER.VAL
MOVE UNSIGNEDINTEGER 1
Production: 28
MOVE DIGITSEQUENCE 1
EVAL FRACTIONALPART.TAG
EVAL FRACTIONALPART.VAL
EVAL FRACTIONALPART.LEN
MOVE FRACTIONALPART 1
Production: 29
EVAL SIGN.TAG
EVAL SIGN.SVAL
MOVE PLUS 1
MOVE SIGN 1
Production: 30
EVAL SIGN.TAG
EVAL SIGN.SVAL
MOVE MINUS 1
MOVE SIGN 1
Production: 31
MOVE STRING 1
EVAL DESCRIPTION.TAG
COND 1
MOVE DESCRIPTION 1
Production: 32
MOVE ID2 1
MOVE IDENTIFIER 1
EVAL ID.TAG
MOVE ID 1
Production: 33
MOVE IDENTIFIER 1
EVAL ID.TAG
MOVE ID 1
Production: 34
MOVE STRINGTOKEN 1
EVAL STRING.TAG
MOVE STRING 1
Production: 35
MOVE CHARACTERTOKEN 1
EVAL CHARACTER.TAG
MOVE CHARACTER 1
Production: 36
EVAL BOOLEAN.VAL
MOVE TRUETOKEN 1
EVAL BOOLEAN.TAG
MOVE BOOLEAN 1
Production: 37
EVAL BOOLEAN.VAL
MOVE FALSETOKEN 1
EVAL BOOLEAN.TAG
MOVE BOOLEAN 1
Sat Sep 12 19:49:25 PDT 1998
***VISIT SEQUENCES***

Production: 0
MOVE ID 1
EVAL METHODDEFN.NAME
MOVE ID2 1
EVAL METHODDEFN.PARENT
MOVE DESCRIPTION 1
EVAL METHODDEFN.DESC
MOVE KEYWORDSSECTION 1
EVAL METHODDEFN.KEYLIST
MOVE DOCUMENTSECTION 1
EVAL METHODDEFN.DOCLIST
EVAL METHODDEFN.INFO
MOVE METHODDEFN 1

Production: 1
MOVE KEYWORDSLIST 1
EVAL KEYWORDSSECTION.KEYLIST
MOVE KEYWORDSSECTION 1

Production: 2
EVAL KEYWORDSSECTION.KEYLIST
MOVE KEYWORDSSECTION 1

Production: 3
MOVE STRING 1
EVAL KEYWORDSLIST.KEYLIST
MOVE KEYWORDSLIST 1

Production: 4
MOVE STRING 1
MOVE KEYWORDSLIST2 1
EVAL KEYWORDSLIST.KEYLIST
COND 1
MOVE KEYWORDSLIST 1

Production: 5
EVAL DOCUMENTSECTION.DOCLIST
MOVE DOCUMENTSECTION 1

Production: 6
MOVE DOCUMENTDEFNLIST 1
EVAL DOCUMENTSECTION.DOCLIST
MOVE DOCUMENTSECTION 1

Production: 7
MOVE DOCUMENTDEFN 1
EVAL DOCUMENTDEFNLIST.DOCLIST
MOVE DOCUMENTDEFNLIST 1

Production: 8
MOVE DOCUMENTDEFN 1
MOVE DOCUMENTDEFNLIST2 1
EVAL DOCUMENTDEFNLIST.DOCLIST
COND 1
MOVE DOCUMENTDEFNLIST 1

Production: 9
MOVE ID 1
MOVE STRING 1
***VISIT SEQUENCES***

Production: 0
EVAL MIXEDDECLLIST.SYMRECLIST
MOVE MIXEDDECLLIST 1

Production: 1
MOVE MIXEDDECL 1
MOVE MIXEDDECLLIST2 1
EVAL MIXEDDECLLIST.SYMRECLIST
COND 1
MOVE MIXEDDECLLIST 1

Production: 2
MOVE TYPEDEFN 1
EVAL MIXEDDECL.SYMRECLIST
MOVE MIXEDDECL 1

Production: 3
MOVE VARDEFN 1
EVAL MIXEDDECL.SYMRECLIST
MOVE MIXEDDECL 1

Production: 4
MOVE CONSTANTDEFN 1
EVAL MIXEDDECL.SYMRECLIST
MOVE MIXEDDECL 1

Production: 5
MOVE FUNCTIONDEFN 1
EVAL MIXEDDECL.SYMRECLIST
MOVE MIXEDDECL 1

Production: 6
MOVE ID 1
EVAL TYPEDEFN.SYMRECLIST
COND 1
MOVE TYPEDENOTER 1
MOVE TYPEDEFN 1

Production: 7
MOVE IDENTIFIERLIST 1
COND 1
MOVE TYPEDENOTER 1
EVAL VARDEFN.SYMRECLIST
MOVE VARDEFN 1

Production: 8
MOVE ID2 1
COND 1
MOVE VALUE 1
COND 2
MOVE ID 1
EVAL CONSTANTDEFN.SYMRECLIST
COND 3
MOVE CONSTANTDEFN 1

Production: 9
MOVE ID2 1
COND 1
MOVE ID 1
COND 2
MOVE ARGLIST 1
EVAL FUNCTIONDEFN.SYMRECLIST
MOVE FUNCTIONDEFN 1
Production: 10
EVAL ARGLIST.SYMRECLIST
MOVE ARGLIST 1
Production: 11
MOVE ARGDCL 1
MOVE ARGLIST.SYMRECLIST
MOVE ARGLIST 1
Production: 12
MOVE ARGDCL 1
MOVE ARGDCL2 1
MOVE ARGLIST.SYMRECLIST
MOVE ARGLIST 1
Production: 13
MOVE TYPEPDEDENOTER 1
EVAL ARGDCL.SYMREC
MOVE ARGDCL 1
Production: 14
MOVE ID 1
EVAL TYPEPDEDENOTER.SYMREC
MOVE TYPEPDEDENOTER 1
Production: 15
MOVE NEWTYPE 1
EVAL TYPEPDEDENOTER.SYMREC
MOVE TYPEPDEDENOTER 1
Production: 16
MOVE ENUMERATEDTYPE 1
EVAL NEWTYPE.SYMREC
MOVE NEWTYPE 1
Production: 17
MOVE ARRAYTYPE 1
EVAL NEWTYPE.SYMREC
MOVE NEWTYPE 1
Production: 18
MOVE RECOMMARETYPE 1
EVAL NEWTYPE.SYMREC
MOVE NEWTYPE 1
Production: 19
MOVE SETTYPE 1
EVAL NEWTYPE.SYMREC
MOVE NEWTYPE 1
Production: 20
COND 1
MOVE IDENTIFIERLIST 1
EVAL ENUMERATEDTYPE.SYMRECLIST
MOVE ENUMERATEDTYPE 1
Production: 21
MOVE FIELDLIST 1
EVAL RECOMMARETYPE.SYMRECLIST
MOVE RECORDTYPE 1
Production: 22
MOVE RECORDSECTION 1
EVAL FIELDLIST.SYMRECLIST
MOVE FIELDLIST 1
Production: 23
MOVE RECORDSECTION 1
MOVE FIELDLIST2 1
EVAL FIELDLIST.SYMRECLIST
COND 1
MOVE FIELDLIST 1
Production: 24
MOVE IDENTIFIERLIST 1
COND 1
MOVE TYPEDENOTER 1
EVAL RECORDSECTION.SYMRECLIST
MOVE RECORDSECTION 1
Production: 25
MOVE TYPEDENOTER 1
EVAL ARRAYTYPE.SYMREC
MOVE INDEXTYPELIST 1
EVAL ARRAYTYPE.INLIST
MOVE ARRAYTYPE 1
Production: 26
MOVE INDEXTYPE 1
EVAL INDEXTYPELIST.INLIST
MOVE INDEXTYPELIST 1
Production: 27
MOVE INDEXTYPE 1
MOVE INDEXTYPELIST2 1
EVAL INDEXTYPELIST.INLIST
MOVE INDEXTYPELIST 1
Production: 28
MOVE LOWERBOUND 1
MOVE UPPERBOUND 1
EVAL INDEXTYPE.INPAIR
MOVE INDEXTYPE 1
Production: 29
MOVE VALUE 1
EVAL LOWERBOUND.TAG
COND 1
MOVE LOWERBOUND 1
Production: 30
MOVE ID 1
EVAL LOWERBOUND.TAG
COND 1
MOVE LOWERBOUND 1
Production: 31
MOVE VALUE 1
EVAL UPPERBOUND.TAG
COND 1
MOVE UPPERBOUND 1
Production: 32
MOVE ID 1
EVAL UPPERBOUND.TAG
COND 1
MOVE UPPERBOUND 1
Production: 33
MOVE BASETYPE 1
EVAL SETTYPE.SYMREC
MOVE SETTYPE 1
Production: 34
MOVE ID 1
EVAL BASETYPE.SYMREC
MOVE BASETYPE 1
Production: 35
MOVE ENUMERATEDTYPE 1
EVAL BASETYPE.SYMREC
MOVE BASETYPE 1
Production: 36
MOVE UNSIGNEDNUMBER 1
EVAL VALUE.TYPE
MOVE SIGN 1
EVAL VALUE.TAG
EVAL VALUE.VAL
MOVE VALUE 1
Production: 37
MOVE UNSIGNEDNUMBER 1
EVAL VALUE.TAG
EVAL VALUE.TYPE
EVAL VALUE.VAL
MOVE VALUE 1
Production: 38
EVAL VALUE.TYPE
MOVE STRING 1
EVAL VALUE.TAG
MOVE VALUE 1
Production: 39
EVAL VALUE.TYPE
MOVE CHARACTER 1
EVAL VALUE.TAG
MOVE VALUE 1
Production: 40
EVAL VALUE.TYPE
MOVE BOOLEAN 1
EVAL VALUE.TAG
EVAL VALUE.VAL
MOVE VALUE 1
Production: 41
EVAL UNSIGNEDNUMBER.TYPE
MOVE UNSIGNEDINTEGER 1
EVAL UNSIGNEDNUMBER.TAG
EVAL UNSIGNEDNUMBER.VAL
MOVE UNSIGNEDNUMBER 1
Production: 42
EVAL UNSIGNEDNUMBER.TYPE
MOVE UNSIGNEDREAL 1
EVAL UNSIGNEDNUMBER.TAG
EVAL UNSIGNEDNUMBER.VAL
MOVE UNSIGNEDNUMBER 1
Production: 43
MOVE UNSIGNEDINTEGER 1
MOVE FRACTIONALPART 1
EVAL UNSIGNEDREAL.TAG
EVAL UNSIGNEDREAL.VAL
MOVE UNSIGNEDREAL 1
Production: 44
MOVE DIGITSEQUENCE 1
EVAL UNSIGNEDINTEGER.TAG
EVAL UNSIGNEDINTEGER.VAL
MOVE UNSIGNEDINTEGER 1
Production: 45
MOVE DIGITSEQUENCE 1
EVAL FRACTIONALPART.TAG
EVAL FRACTIONALPART.VAL
EVAL FRACTIONALPART.LEN
MOVE FRACTIONALPART 1
Production: 46
EVAL SIGN.TAG
EVAL SIGN.SVAL
MOVE PLUS 1
MOVE SIGN 1
Production: 47
EVAL SIGN.TAG
EVAL SIGN.SVAL
MOVE MINUS 1
MOVE SIGN 1
Production: 48
MOVE ID 1
EVAL IDENTIFIERLIST.IDLIST
MOVE IDENTIFIERLIST 1
Production: 49
MOVE ID 1
MOVE IDENTIFIERLIST2 1
EVAL IDENTIFIERLIST.IDLIST
COND 1
MOVE IDENTIFIERLIST 1
Production: 50
MOVE STRING 1
EVAL DESCRIPTION.TAG
COND 1
MOVE DESCRIPTION 1
Production: 51
MOVE ID2 1
MOVE IDENTIFIER 1
EVAL ID.TAG
REFERENCES


