

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

**SEMANTIC FORMALIZATION IN MATHEMATICAL
MODELING LANGUAGES**

Fernando Vicuna

**August 1990
CSD-900020**

UNIVERSITY OF CALIFORNIA

Los Angeles

Semantic Formalization in Mathematical Modeling Languages

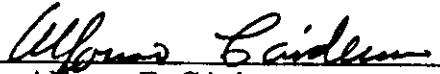
A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

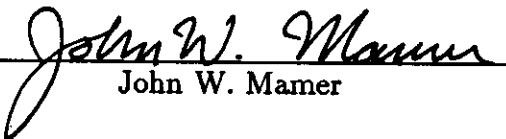
by

Fernando Vicuña

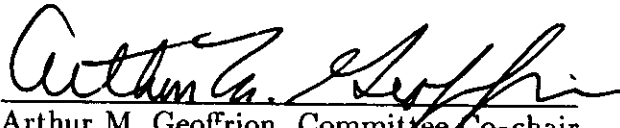
1990

The dissertation of Fernando Vicuña is approved.


Alfonso F. Cárdenas


John W. Mamer


Richard R. Muntz


Arthur M. Geoffrion, Committee Co-chair


Michel A. Melkanoff, Committee Co-chair

University of California, Los Angeles

1990

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	v
VITA	vi
ABSTRACT	vii
1 Introduction	1
1.1 Statement of Purpose	1
1.2 Motivation for this Dissertation	4
1.3 Dissertation Overview	8
2 Semantic Analysis in Mathematical Modeling Languages	11
2.1 Language Evaluation Issues	12
2.2 Systems Examined	18
2.2.1 AMPL	18
2.2.2 GAMS	29
2.2.3 LINGO	42
2.3 Discussion	49
3 Formalization of Static Semantics via Attribute Grammars	54
3.1 Attribute Grammars	55
3.2 SML: Structured Modeling Language	57
3.3 Formalization Methodology	60
3.4 Complexity	63
3.5 Examples of Schema Properties and Table Content Rules	64
3.5.1 Equations for Schema Overview	65
3.5.2 Equations for Generic Calling Sequence Sublanguage	75
3.5.3 Equations for Range Statement Sublanguage	77
3.5.4 Equations for Index Set Statement Sublanguage	79
3.5.5 Equations for Generic Rule Statement Sublanguage	79
3.5.6 Equations for Interpretation Sublanguage	80
3.5.7 Equations for Elemental Detail Tables	81
3.6 Importance of the Attribute Grammar Approach	83
3.6.1 Easy Modifiability	85
3.6.2 Declarative Approach	87
3.7 Difficulties and Lessons	88

4	Type Inferencing in Mathematical Modeling Languages	91
4.1	Areas Amenable to Inferencing in SML	93
4.1.1	Simple Variables	93
4.1.2	Type Declaration for Symbolic Parameters	94
4.1.3	Domain Statements	95
4.1.4	Calling Sequences	96
4.2	Type Inferencing in SML via Attribute Grammars	97
4.2.1	Inference of Types of Symbolic Parameters	98
4.2.2	Inference of Types Implied by Domain Statements	109
4.3	Discussion	112
5	Expression Evaluation in Mathematical Modeling Languages	114
5.1	Related Approaches for Semantic Evaluation	116
5.1.1	Attribute Grammar Methods	117
5.1.2	Denotational Semantic Methods	119
5.2	Immediate Expression Evaluation in SML	120
5.2.1	Example of Evaluation Code	121
5.2.2	An Attribute Grammar Based Compiler	127
5.3	Discussion	132
6	Implementation of a Syntax-Directed Editor Prototype	134
6.1	A Syntax-Directed Editor Description in SSL	135
6.2	Prototype Implementation	140
6.2.1	Example of Semantic Restrictions in Paragraphs	142
6.2.2	Example of Semantic Restrictions in Calling Sequences	144
6.2.3	Example of Semantic Restrictions in Index Set Statements	146
6.2.4	Example of Semantic Restrictions in Generic Rules	150
7	Conclusions	156
7.1	Summary of Contributions	157
7.2	Limitations	159
7.3	Further Research	161
7.3.1	Answering Queries about Model and Data	161
7.3.2	Support for a Complete Modeling Environment	164
7.3.3	Full Support for Immediate Evaluation	165
A	An Abstract Syntax for SML	167
	References	184

ABSTRACT OF THE DISSERTATION

Semantic Formalization in Mathematical Modeling Languages

by

Fernando Vicuña

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1990

Professor Arthur M. Geoffrion, Co-chair

Professor Michel A. Melkanoff, Co-chair

Mathematical modeling is notoriously error-prone because existing commercial languages and systems for Operations Research (OR) modeling lack semantic formalization. The absence of formal semantics has limited the amount of assistance that is provided by current mathematical modeling language environments to detect and prevent errors in models. For example, it has prevented the development of practical tools which could provide immediate feedback on the location and nature of semantic errors in a model, of tool extensions which could simplify model design by automatically deducing missing, or incomplete, fragments in the model, and of entirely new semantics-driven tools which could support model-analysis activities within the modeling environment itself. Even more importantly, this

deficiency has prevented the automatic generation of major components of the environments for OR modeling.

The purpose of this dissertation is to promote semantic formalization in mathematical modeling languages by showing the feasibility of employing an attribute grammar formalism as a design paradigm for some of the major components of an OR modeling environment. We begin by identifying the areas where semantic formalization is needed in selected modeling languages for mathematical programming. Then, using SML (Structured Modeling Language) as an example, we propose and show the feasibility of supporting the complete static semantics of mathematical modeling languages through attribute grammar equations. Next, we propose equations for various areas that we identify as amenable to language simplification via the automatic deduction of missing language constructs. Later, we show how we could support the numerical evaluation of expressions in immediate mode in SML. Finally, we validate our approach by implementing a prototype modeling environment which enforces the full syntax and semantics of SML, and by testing it using a large number of models from a variety of domains.

We conclude that it is possible to use attribute grammars to rigorously formalize a mathematical modeling language, and to automatically generate modeling environment tools. This kind of explicit semantic formalization can help to make the modeling activity in OR less error-prone, thereby shortening the length of time required to develop correct new models and maintain existing ones.

CHAPTER 1

Introduction

1.1 Statement of Purpose

The purpose of this dissertation is to promote semantic formalization in mathematical modeling languages and computer-based modeling environments for the Operations Research/Management Science (OR/MS) community, as an approach to ensure model correctness and to provide systematic methodologies for developing aids and tools to support the model formulation process. The particular direction we follow is to present a design and implementation of a development environment for OR/MS modeling based on the attribute grammar methodology.

The primary claim of this dissertation is that a declarative specification based on the attribute grammar foundation can and should be used as the definition paradigm for some of the major components of future modeling environments, such as language-based editors for building models, type checking and inferencing tools to simplify the writing of modeling statements, and code generators to support expression evaluation activities in immediate mode. The significance of the claim is that it should be possible to reduce the amount of time required to develop correct new models and to maintain existing ones in an attribute grammar

based modeling environment than in other currently existing ones. Furthermore, the generation of the semantic tools and aids for the modeling environment can be automated from the attribute grammar description of the modeling language. The benefits of this automation come not only from the time saved during the initial development of the modeling environment tools, but also from time savings achieved during enhancements, changes, and evolution of these systems. Thus, the attribute grammar approach supports “evolutionary flexibility”, a very desirable characteristic for modeling environments.

To examine the feasibility of our claim, the following hypotheses can be examined:

1. Current mathematical modeling languages which lack formal context-sensitive semantic definitions tend to make the modeling activity more error-prone, and the models written in these languages are more difficult to develop correctly and to maintain.
2. Complete static semantics of mathematical modeling languages can be supported via the attribute grammar framework.
3. Inferencing, or deduction, of mathematical modeling language constructs can be realized via the attribute grammar framework.
4. Evaluation of expressions in mathematical modeling languages can be supported in immediate mode via the attribute grammar framework.

5. Language-based tools, including syntax-directed editors that guarantee that model and data are correctly built and maintained, can be generated automatically from an attribute grammar framework.
6. Queries on the model structure and on its data can be answered with the support of an attribute grammar framework.
7. Tools to support structural relationships, such as configuration managers and version control systems, can also be supported with an attribute grammar framework.

One of the most challenging aspects of this dissertation is to investigate the feasibility of the primary claim. The first five hypotheses are the themes examined in subsequent chapters of the dissertation. The last two hypotheses are outside the scope of this work, but remain a topic of future research.

Attribute grammars is not the only approach to semantic formalization in mathematical modeling languages and environments. Among the other alternative paradigms for specifying a language or environment, the most popular ones include toolkit, ad hoc semantic-action, and method-based approaches; a complete taxonomy of software development environments, including toolkit, method-based, structure-oriented, and language-centered environments, can be found in [Dart87]. Other formal paradigms include logic programming approaches [Ster86], denotational semantic approaches [Gord79][Tenn76], and axiomatic semantic approaches in Hoare style [Hoar69]. An extensive bibliography of these systems and other

semantics-related activities can be found in [Gogu87].

1.2 Motivation for this Dissertation

In the Operations Research and Management Science communities there is an ever increasing number of modeling languages and systems available for mathematical modeling. To name a few examples, in the area of mathematical programming there are: AMPL [Four89], GAMS [Broo88], IFPS/OPTIMUM [Roy86], LINGO [Cunn89], LPL [Hü87], MIMI/LP [Bake86], MPL [Soft], and PAM [Welc87]. See [Matu87] for a list that includes over 20 mathematical programming systems together with a comparative analysis of some of them; see [Chil85] for an overview of the analytical computing environment ANALYTICOL; and see [Cuni87a] [Cuni87b] for a collection of special articles on mathematical programming modeling systems.

In spite of the numerous modeling languages and systems, mathematical modeling remains an activity that is notoriously error-prone for the following reasons. First, traditional modeling languages and systems have been weak in the area of detecting errors in models. Contemporary modeling languages, for instance, have not been designed around the idea of preventing errors. And although computer assisted systems to analyze linear programming models and diagnose errors do exist [Gree83], the approach usually taken has been to provide assistance after the “solvers” have been employed. (A solver is a piece of computer software for solving a given problem type, such as a linear or nonlinear programming problem.)

Next, although syntax and post-algorithm error checks are supported by some

systems, only a limited amount of assistance is provided by most existing systems to detect and prevent errors a priori in the model [Biss87]. For example, tools which could provide immediate feedback on the location and nature of semantic errors in the model are practically non-existent. Also missing are tool extensions which could simplify model design by automatically deducing missing, or incomplete, fragments in a model. Similarly absent are any class of entirely semantics-driven tools which could support model analysis activities within the modeling system itself.

Finally, in general, current popular modeling systems consist of a set of loosely integrated tools that only partially support the different aspects of the modeling life-cycle. Further, none of the existing systems has enough capabilities to support the entire life-cycle process, all the way from the conception of the model to its implementation, testing, maintenance, and modification. As a consequence, no system of today has the flexibility, level of integration, and broad scope required to be considered a “true” modeling environment in the sense given in [Geof89a].

The creation of a new generation of semantics-driven environments where mathematical modeling could be developed in a less error-prone fashion, and where environment tools could be automatically generated, is one of the most pressing challenges currently faced by the OR/MS community. The potential impact of such new modeling environments in terms of increased productivity, quality, and popularity of applied modeling work has already been argued in the literature (see e.g., [Geof89a], [Geof89b]). Thus, this dissertation is motivated in part as a

response to this challenge.

Our other main motivation is as follows. It is surprising that, even at the time of this writing, very little attention has been paid by the OR/MS community to the area of semantic formalization of modeling languages and modeling systems as an approach to improving model correctness, and as a way of automatically generating aids and tools to support the model formulation process. Few contemporary modeling languages and systems –and this is also true for programming environments– have requirements defined as precisely and formally as, say, selected programming languages that include Prolog [Jone84], Ada [Kini82], and LISP dialects [Much80]. A recent exception is the modeling language SML [Geof88], whose static semantics are completely formalized, and whose prototype implementation FW/SM [Geof90] offers many tools to avoid model errors.

The absence of precise, formal semantic definitions in modeling languages and systems has contributed to the lack of successful modeling environment paradigms in various important ways:

- It has prevented the automatic generation of major components of the environments for OR/MS modeling.

- It has increased the effort and cost of software development and maintenance. This negative impact has also been shared by the programming community, which has long recognized that ease of maintenance and modification plays a key role in controlling software costs and increasing productivity [Boeh87].

- It has influenced the evolution of modeling languages and systems in the di-

rection of data-centered approaches rather than toward semantics-driven methodologies. Needless to say, the software usually is developed with manual and ad-hoc procedures, and thus there is little possibility of software reusability.

Our recognition of the importance of the role that semantic formalization of modeling languages and systems plays in the control over model errors, coupled with our belief that the design of stronger semantics-driven modeling environments have the potential to influence the way model-based work will be carried out in the future, are the main motivation for this dissertation.

An immediate challenge of this work is to show the benefits of semantic formalization of modeling languages as a road to asserting model correctness. Another short term goal is to pioneer a methodology for an improved modeling environment that lends itself to rapid prototyping, so that new modeling studies may be quickly undertaken.

Following an analogy between the programming language and modeling language communities, our longer term goals are to develop uniform semantic methodologies that may be used to support not only “modeling in the small” activities (like small-scale model editing and model evaluation), but also “modeling in the large” activities (like large-scale modeling, model configuration, model management, and model sharing) as well. According to the views of DeRemer and Kron [DeRe76], very different languages should be used for these two apparently different activities. Integration, however, has benefited from developments on many fronts. Research in software design methodologies, like top-down design, has created an

effective way of handling both small-scale and large-scale problems. Integration has also been the goal set for some software environments [Clem86]. And integration is also (partially) supported by software development facilities such as UNIX [Ritc74].

1.3 Dissertation Overview

Each chapter in the dissertation is chartered to examine one of the hypotheses listed in Section 1.1. The chapters as a whole lend support to the primary claim. The different aspects of our work are now briefly explained.

Chapter 2 examines the first hypothesis. It surveys the capabilities for error detection and semantic analysis in commercial modeling languages for mathematical programming. The semantic restrictions of GAMS, LINGO, and AMPL are recognized and reviewed in this survey. These particular languages have been chosen because of their popularity and importance, and because they present designs that are representative of most of the options that are currently available in model definition languages for mathematical programming. The current shortcomings of these languages indicate the need for a more formal approach to semantic analysis and model correctness.

Chapter 3 tests the second hypothesis: that attribute-grammar based definitions can be used to represent the static semantics of modeling languages. Using the SML language as a test language (however, the same could be done for other modeling languages like GAMS, AMPL, and LINGO), we show the feasibility of

strictly enforcing with attribute grammar equations all the static properties of SML (i.e., the model constraints called “Schema Properties”, together with the data constraints called “Table Content Rules”). One objective is to show that the static semantics of even a complex model definition language like SML can be enforced in polynomial time.

Chapter 4 tests the third hypothesis: that inferencing, or deduction, of mathematical modeling language constructs can be supported via attribute grammar based definitions. The approach is illustrated by the design and implementation of an inferencing type checking mechanism for SML constructs that may be used as alternative to the default one in the language.

More specifically, an SML model either explicitly declares the types of the expressions that participate in an evaluation rule, or these expression types are implicit from the semantics. This requirement makes it difficult to maintain partially complete models (i.e., models which are not yet completely refined). It is possible, however, to have the modeling environment infer the types of all expressions from their use, even before all its variables are explicitly declared. For example, the environment can relieve the modeler from making explicit the type declaration of a construct. This declaration can be generated automatically from the use of the construct constituents.

Chapter 4 proposes specific changes in SML that allow these alternative type checking mechanisms. The approach is generic in nature, and thus may be used for purposes of automatic deduction of selected constructs in other mathematical

modeling languages.

Chapter 5 examines the fourth hypothesis. It presents the design of an extension to an attribute grammar based modeling environment that can be used to perform model analysis and evaluation activities. A code generation tool is shown which can be used in conjunction with a run-time evaluator to support the evaluation of expressions in SML in immediate mode. Naturally, similar approaches may be taken for other mathematical modeling languages.

Chapter 6 examines the fifth hypothesis. It describes a prototype syntax-directed editor of SML models. This prototype is shown to enforce the complete syntax and static semantics of SML, and is tested using a large number of models from a variety of domains. Since its inception, the prototype tool has been useful for developing correct new models and maintaining existing ones.

Chapter 7 gives the conclusions of this dissertation. It summarizes the contributions of this work, lists its limitations, and suggests future research directions. A main conclusion is that it is possible to use attribute grammars to rigorously formalize a mathematical modeling language, and to automatically generate modeling environment tools. This kind of explicit semantic formalization can help to make the modeling activity in OR/MS less error-prone, thereby shortening the length of time required to develop correct new models and maintain existing ones.

CHAPTER 2

Semantic Analysis in Mathematical Modeling Languages

The ability to trap modeling errors is one of the most important features of a modeling language and system aimed at mathematical programming. This chapter will examine the technologies with which existing commercial modeling languages and systems for mathematical programming express and enforce their syntactic and semantic restrictions, and thus prevent and detect modeling errors. The purpose for this examination is to test the hypothesis that mathematical modeling languages that do not offer a high degree of formalization of context-sensitive semantics tend to make modeling more error-prone, and the models written in these languages are more difficult to develop correctly and to maintain.

The mathematical programming modeling languages chosen for examination are AMPL [Four89], GAMS [Broo88], and LINGO [Cunn89]. We do not include SML [Geof88] in this discussion, since this model definition language for structured modeling will be covered in more detail in a separate chapter.

The approach taken here will be to study how semantic restrictions and other design-related issues are handled by the three modeling languages for mathematical programming. The issues under study are each language's semantic restrictions, run-time model checks, unsafe constructs, diagnostic system, readability

and writability, and implementability. These issues have been chosen for their contribution or importance to the technologies with which semantic analysis is performed in mathematical programming modeling languages.

The chapter is organized as follows. Section 2.1 explains the six language issues which are used to evaluate AMPL, GAMS, and LINGO. A simple transportation problem is also shown for convenience. In Section 2.2, each of the mathematical programming modeling languages is examined with respect to the evaluation issues. Finally, a discussion concerning the three languages studied is presented in Section 2.3.

2.1 Language Evaluation Issues

As a running example for discussing language issues, we will consider the following transportation problem adapted from [Geof89e], and an associated database given in [Geof89b] and stated here in English prose:

Model.

I: set of plants
S(I): supply capacity (in tons) for plant i
J: set of customers
D(J): nonnegative demand (in tons) for customer j
C(I,J): transportation cost (in \$/ton) from plant i to customer j
F(I,J): nonnegative flow (in tons) from plant i to customer j

The following definitional equations hold:

min $\sum(I)\sum(J)(C(I,J)*F(I,J))$

s.t.

$\sum(J) (F(I,J)) \leq S(I)$, for all I. supply constraint

$\sum(I) (F(I,J)) \geq D(J)$, for all J. demand constraint

Database.

The plants are {DALLAS, CHICAGO}.
The DALLAS plant has a supply capacity of 20,000 tons.
The CHICAGO plant has a supply capacity of 42,000 tons.
The customers are {PITTSBURG, ATLANTA, CLEVELAND}.
PITTSBURG has a demand of 25,000 tons.
ATLANTA has a demand of 15,000 tons.
CLEVELAND has a demand of 22,000 tons.
The {DALLAS,PITTSBURG} transportation cost is \$23.50/ton.
The {DALLAS,ATLANTA} transportation cost is \$17.75/ton.
The {DALLAS,CLEVELAND} transportation cost is \$32.45/ton.
The {CHICAGO,PITTSBURG} transportation cost is \$7.60/ton.
The {CHICAGO,CLEVELAND} transportation cost is \$22.75/ton.

The issues below are the basis for evaluating the modeling languages and systems for mathematical programming. Note that the viewpoint is that explicit semantic restrictions are of fundamental importance to the mathematical programming modeling languages, as a means of preventing and detecting modeling errors. This point of view influences all of the other issues by which the languages and their systems are evaluated.

- **Semantic Restrictions**

A modeling language for mathematical programming provides abstractions to specify a model class, and data associated with particular instances of the model class. To help catch errors, the modeling language defines a set of semantic restrictions that apply to the abstractions of the model and to its data. Typically, these restrictions are context-sensitive, i.e., cannot be generated by a context-free grammar; they are either explicitly declared in English prose or are left implicit. For each mathematical programming modeling language, this subsection will identify (in boldface) some of the most

important semantic restrictions that apply to the model (Model Restrictions flagged <MR>), and to the data supplied to the model (Data Restrictions flagged <DR>). Any violation of these restrictions ought to generate an error message from the language processing system.

The question is left open as to whether some semantic restrictions might still be checked by the translators (since they might appear on an exhaustive list of error messages), although they might not be recognized explicitly by the documentation of these languages. These kinds of implicit semantic restrictions are not considered in this discussion, as their appearance would show too-low attention paid by the language designers to making semantic restrictions explicit. Hence, there are no claims concerning “completeness”.

- **Run-Time versus Static Model Checking**

To understand how a model is processed in a language for mathematical programming, it is helpful to sketch in very general terms the steps that a model goes through in GAMS, from its initial definition stage to its final solution. The first phase in the processing of a model is the translation-time (also called static-time, or compilation) phase, where the syntax and the consistency of the model are checked. The second phase in the processing (also called the run-time, or execution phase) is divided into a generation-time phase and a solver-time phase. Generation-time is the initial stage of solving a SOLVE statement; transformations on the data are carried out,

and a problem description is generated for input to the solver. The solver is then invoked at solver-time, to solve the specific instance of the model that was generated during the generation-time phase. Finally, the solver's output is printed to a file.

This subsection refers to how conducive a modeling language for mathematical programming is to having errors caught by static-time rather than run-time checks. A typical "strongly-typed" language allows most of its syntactic and semantic checking to be done while the model and its data are being translated in the pre-solver phase(s), before the actual model generation and solver phases. Those semantic restrictions which cannot be processed at model translation-time in the language (i.e., static-time), have to be enforced at model generation-time or solver-time (i.e., run-time). For the three mathematical programming modeling languages under evaluation, some additional semantic restrictions that are only checked during run-time will be identified (in boldface).

- **Unsafe Constructs**

This subsection will highlight the areas where the individual mathematical programming modeling languages fail to include important semantic restrictions. In general, the lack of appropriate semantic restrictions leads to unsafe constructs, and the resulting models are more likely to contain errors that will go undetected.

Of course, the safety of a modeling language can be proven mathematically if the language is based on some formal semantic foundation. In less rigorously-founded modeling languages, testing can be used as a tool to detect model errors. However, no practical test criteria can guarantee the absence of errors. Hence, in this examination we cannot offer any claim of completeness of our findings.

- **Diagnostic System**

The friendlier the diagnostic facility in a modeling language for mathematical programming, the simpler it is to discover and correct model anomalies. Comprehensive error diagnostics can be provided in a mathematical programming modeling language as a consequence of explicit semantic restrictions. Indeed, after a language has identified its set of semantic restrictions, a good diagnostic system can be designed and implemented to enforce those design principles.

To describe the flavor of the diagnostic system, this subsection will give the output for the transportation example for each language, using a version of the original database that contains these erroneous modifications:

```
The customers are {PITTSBURG, ATLANTA, CLEVELAND, ATLANTA}.  
BOSTON has a demand of 25,000 tons.
```

Note that the ATLANTA customer is duplicated, and BOSTON does not exist.

- **Readability and Writability**

The reliability (or confidence in the correctness) of a model is strongly related to the rigorous definition of the semantics of the modeling language for mathematical programming. Indeed, explicit semantic restrictions affect the readability of a model (by eliminating anomalies, or making them detectable) and consequently, its reliability (the easier we can read a model the better we can reason about its correctness). Explicit semantic restrictions are also related to the writability of a model (by eliminating vagueness) and therefore, to its reliability (the easier we can write models the greater the confidence we can have of the correctness of what we write). Note that in this context, readability and writability refer to “model readability and writability by persons familiar with the mathematical programming modeling language”.

We note that, besides explicit semantic restrictions, another approach to increased reliability in mathematical programming modeling languages is offered by Bradley and Clemence’s “type calculus” system [Brad87]. This system adds units of measurement to an algebraic modeling language, and provides powerful checks on the completeness and consistency of a model.

This subsection will comment on the readability and writability aspects of a few selected features of the mathematical programming modeling languages; like the separation of model and data, self documentation features, and the nature of comments. These features, which are also very important in the

promotion of ‘good’ modeling style, are commented upon here from the viewpoint of their relationship to semantic restrictions and reliability.

- **Implementability**

The architecture of the system supporting the modeling language for mathematical programming dictates the way its semantic restrictions are implemented. Different architectural designs (pure model translation, or model translation combined with model solving) provide very different implementations of semantic checks and different efficiency. Because of the importance of designing semantic restrictions that can execute efficiently (in both time and space), this subsection will focus on how the mathematical programming modeling languages implement their semantic restrictions.

2.2 Systems Examined

In this section, the mathematical programming modeling languages AMPL, GAMS, and LINGO are examined in the context of semantic restrictions and related issues expounded in the previous section.

2.2.1 AMPL

AMPL (**A Mathematical Programming Language**) is a modeling language and processor currently being developed by AT&T for use in mathematical programming applications. The AMPL processor runs directly under DOS, receiving a

model written in the AMPL language as input, and sending the output of the execution directly to the screen.

The AMPL processor is invoked to translate an AMPL model into a format which can then be used by different optimizers. [Four89] reports that a translation routine is now available for the MPS format. This routine outputs a file suitable for the MINOS optimizer as well [Murt87,Rose86].

2.2.1.1 Semantic Restrictions

In AMPL, the semantic restrictions, or properties, apply either to the model itself (Model Restrictions), or to the data supplied to the model (Data Restrictions). These restrictions must be inferred from [Gay89a,Gay89b], since AMPL does not define them explicitly. Violations of Model Restrictions or Data Restrictions always generate a fatal error message from the AMPL processor.

An AMPL model includes declarations of entities (sets, parameters, variables, constraints, objectives), optional “check” statements (that supply assertions to verify that correct data have been read or generated), and a data set which begins with a “data” statement and ends either with the “end” statement or with the end-of-file. Although there is no standard format required in AMPL, two principles are followed: (1) model entity declarations, check statements, and data statements are terminated by a semicolon, and (2) **entities must be uniquely declared before they can be referenced** <MR1>; forward references are disallowed. Both the model declarations and the model data can be stored in either combined

or individual model and data files, which can be created with any ordinary text-editor.

Example. An AMPL model for the transportation problem will have the following structure:

```
#Model.
set plants;                #set of plants
set customers;            #set of customers
param supply {plants};    #supply capacity (in tons) for plant i
param demand {customers}; #demand (in tons) for customer j
param cost {plants,customers}; #transportation cost (in $ per ton)
                                #from plant i to customer j
var F {plants,customers}; #nonnegative flow (in tons)
                                #from plant i to customer j

minimize total_cost:
    sum{ i in plants, j in customers} cost[i,j]*F[i,j];
tsupply 'supply constraint' {i in plants}:
    sum{j in customers} F[i,j] <= supply[i] ;
tdemand 'demand constraint' {j in customers}:
    sum{i in plants} F[i,j] = demand[j] ;

#Data.
data;
set plants      := DALLAS CHICAGO;
set customers   := PITTSBURG ATLANTA CLEVELAND;
param supply    := DALLAS 20000 CHICAGO 42000;
param demand   := PITTSBURG 25000
                  ATLANTA  15000
                  CLEVELAND 22000;
param cost      :  PITTSBURG  ATLANTA  CLEVELAND :=
                  DALLAS    23.50   17.75   32.45
                  CHICAGO   7.60    0      25.75;

end;
```

A set in AMPL contains zero or more distinct $\langle DR1 \rangle$ members. All members must have the same number of elements $\langle DR2 \rangle$, which is the set's "dimension". AMPL also provides for the declaration of arbitrary subsets. The data declared for the subsets must be members of the original sets $\langle DR3 \rangle$.

Model entities may be subscripted by an indexing expression. The general structure of the indexing expression is a beginning left brace '{' followed by a list of one or more comma-delimited base sets, followed optionally by a colon and a “such-that” expression, and ended by a right brace '}'. Each base set may be preceded by a dummy member and the keyword “*in*”. A dummy member for a one-dimensional set is simply an **unbound** <MR2> name (i.e., an alphanumeric string that is different from previous model entity names). A dummy member for a multi-dimensional set is comma-delimited list, enclosed in parenthesis, of expressions or **unbound** <MR2> alphanumeric names; the list must **include at least one unbound name** <MR3>.

Arithmetic expressions in AMPL can include any of the following functions: *abs, acos, acosh, asin, asinh, atan, atan2, atanh, ceil, cos, cosh, exp, floor, log, log10, sin, sinh, sqrt, tan, tanh*. Of these, **only atan2 is binary, and the rest are unary** <MR4>.

The binary operator *in* checks set membership. Its left operand is an expression, or a comma-delimited set of expressions, **such that the number of expressions is equal to the *dimen* of the right operand** <MR5>, which must be a set expression.

Piecewise-linear terms have one of the following structures:

```
<< bkpts ; slopes >> var
<< bkpts ; slopes >> (expr)
<< bkpts ; slopes >> (var,expr)
```

where *bkpts* is a list of breakpoints and *slopes* is a list of slopes. After the list of

slopes and breakpoints are extended (by indexing over any indexing expressions), **there must be one more slope than breakpoints** <MR6>.

Parameter declarations permit a number of optional phrases, like *logical*, *integer*, *symbolic*, etc. The keyword *logical*, for example, requires the named parameter to be either 0 or 1. Hence, **the type of the input data must conform to the corresponding restrictions implied by the parameter declarations** <DR4>.

Recursive definitions of indexed parameters are allowed in AMPL. Of course, the value of a recursively defined parameter can be computed in sequences that refer to previously computed values, but the modeler does not need to worry about what the sequence is. **An attempt to evaluate an element that has not been previously computed will elicit a diagnostic error** <DR5>. (In the statement “*param p{i in A} := if i > 3 then p[4] else ...;*”, if no occasion arises for evaluating $p[i]$ for $i > 3$, then no error message is produced).

Some constraints have the following structure:

```
cexpr relop vexpr relop cexpr
cexpr relop tkexpr relop cexpr
```

where *relop* stands for one of the relational operators (\leq , \geq , $==$, $=$). When there are two *relop*'s, **both must be \leq or both must be \geq** <MR7>.

The special form of indexing expression, *if lexp*, is allowed for variable, constraint, and objective declarations (and also for piecewise-linear terms). If the logical expression *lexp* is true, then an unsubscripted entity results; otherwise the

entity is excluded from the model. **It is illegal to subsequently reference an excluded entity** <MR8>.

The phrases `:=` and `default` in set declarations are mutually exclusive. If the set is not defined in the data section, **either phrase must be present** <MR9>.

2.2.1.2 Run-Time versus Static Model Checking

The semantic restrictions identified in the previous subsection are all checked statically (i.e., at model translation-time) by the AMPL processor. Some model errors, however, like the specification of inconsistent constraints, cannot be detected by the AMPL processor, and are actually caught by a post-translation solver. AMPL does not include a run-time, or execution-time, processor.

In AMPL, there are no exception handling capabilities. Run-time errors cannot be trapped, so the execution immediately halts following the exception.

2.2.1.3 Unsafe Constructs

AMPL does not include semantic restrictions to enforce the concept of domain integrity in the traditional relational algebraic sense. The language has chosen to allow traditional binary set operators, like difference, intersection, and union, to draw their operands from any domains. Although such set operations are mathematically defined for operands drawn from distinct domains, it is generally not semantically meaningful, nor safe, to do so. For example, the following model

```
set part_number 'a set of unique part identifiers';
set fish        'some edible species';
```

```

set    aggregate within part_number union fish;
data;
set    part_number := 1 2;
set    fish        := BASS HALIBUT;
end;

```

is legal in AMPL, although the values of the *aggregate* set, {1, 2, BASS, HALIBUT}, do not come from a common domain. AMPL takes the viewpoint that the elements of this set belong to the universal domain. From a different perspective, however, this violates a basic principle of relational data structure. It prevents the language from performing automatic domain integrity checks, as in other more strongly typed modeling languages and systems. In AMPL, either the *within* clauses in *set* declarations or separate *check* statements are required to enforce domain integrity.

Another area where AMPL lacks appropriate semantic restrictions is in its construct for indexed expressions. The following statements, for example,

```

set    faulty1 := { f in fish : log(-f) };
set    faulty2 := { f in fish : f+1/0 };
set    faulty3 := { f in fish, p in part_number : f +2 <> p };

```

are all legal in AMPL, although they are meaningless. The language does not check at translation time for the domains of operands of arithmetic and logical operators if they occur in an expression that does not need to be evaluated. In particular, the translator only computes the elements of a set if it needs to iterate over the set; it can usually check membership in a set without computing all the elements, which sometimes yields a big time savings. Thus, the subprogram above does not elicit complaints since neither *faulty1*, nor *faulty2*, nor *faulty3* needs to be evaluated. However, if *var x{faulty1,faulty2,faulty3};* is inserted before a *data*

statement, an appropriate error message is generated.

2.2.1.4 Diagnostic System

The error reporting capabilities in AMPL are rather simple. During the translation process, if AMPL discovers errors, it sends an error message directly to the screen. This message, however, will only describe the first mistake found by the translator, and further mistakes may be sometimes lost. Although error recovery does not appear very powerful in AMPL, the translation of the model will continue as long as it is reasonable, or until the translator has printed the number of error messages specified by the *-e* option. It is necessary to correct the first mistake before processing can be resumed.

When AMPL encounters a translation error, it optionally outputs the name of the file and the line number of the offending error. Then an explanatory message is issued, optionally followed by a “context” line, which approximately locates the end of the offending statement. For data section errors, the context line marks the semicolon at the end of the statement with the symbols “>>>;<<<”. The reason why data-section error messages point to the semicolon at the end of the relevant statement is that, when reading a set in the data section, the translator first reads all its elements, and then checks whether they satisfy domain restrictions. For syntax errors, >>><<< surround a token near the one where the error was detected.

In the transportation problem, when the database contains the first mistake,

```
#the transportation example is stored in file 'diag.amp'  
set customers := PITTSBURG ATLANTA  
               CLEVELAND ATLANTA;
```

the translator issues the message:

```
diag.amp, line 18 (offset 667):  
      duplicate element ATLANTA for set customers  
context:      CLEVELAND ATLANTA >>> ; <<<
```

When the database contains the second mistake,

```
param demand := PITTSBURG 25000  
                ATLANTA   15000  
                CLEVELAND 22000  
                BOSTON    25000;
```

the translator discovers the unknown customer BOSTON and issues the message:

```
error processing param demand:  
      invalid subscript demand[BOSTON]
```

2.2.1.5 Readability and Writability

AMPL models are easy to read because the model section and the data section are kept separate. This principle of independence of model and data is strongly encouraged by AMPL, although instances of commingling of model structure with data are possible, as shown in [Geof89c]. An immediate advantage of this independence in AMPL is that Model Restrictions and Data Restrictions can be implemented separately.

AMPL includes two documentation features: comments and literals. Comments have the following syntax: they start with the symbol '#' and extend to the end of the line. Literals are strings delimited by single quotes or double quotes,

which are associated with the entity that is being declared. In the general model entity declaration:

```
Entity name    [alias]    [indexing]
```

“alias” is an optional literal. This literal is treated as a comment which AMPL makes available to solvers and post-optimal analyzers. However, since comments and literals are not subject to semantic restrictions, they cannot be used by specialized tools to verify model correctness.

2.2.1.6 Implementability

The AMPL processor, which is written in C++, takes the entity declaration file representing the model, and the input data (file) following the “data” statement, and proceeds to write a new representation of the model in a style suitable for optimization algorithms. The translation work is carried out in the following seven phases described in [Four89]:

- *Parse.* The lexical analyzer reads characters from the model file and sends them to the parser which generates suitable expression trees. The lexical analyzer is written in Lex [Lesk75] and the parser is written using YACC [John78].
- *Read Data.* This phase reads the input data and performs various on-line semantic checks. It checks that all entities are uniquely defined; that no set contains duplicate identifier values; that set elements have the correct number of dimensions; and that the given parameter values are of the correct type.

- *Compile*. This phase optimizes the expression trees created in the first phase, subject to the data read in the second phase. The optimization actions include removing invariants from loops, coalescing common subexpressions, and rearranging the retrieval of parameter values that are indexed over an identical set.
- *Generate*. In this phase, all the derived sets are computed and the list of all linear terms of the model are generated. Checks are performed on the data to verify the satisfaction of the conditions imposed by the model.
- *Collect*. In this phase, the repeated appearances of a variable in terms with the same objective or constraint are collected and merged into one appearance. The sorting of coefficients to match the order of the constraints is also performed.
- *Presolve*. This phase applies transformations on the linear program defined by the model and data files to make it simpler and smaller.
- *Output*. This final phase generates the original model translated into a file format suitable for optimizers.

From this architectural layout, we observe that the *Parse* phase implements the syntactic analysis, and the *Read Data* and *Compile* phases implement most of the context-sensitive semantic restrictions. This modular design provides enough separation between phases to make the implementation of additional semantic

restrictions an easy task. However, since a parallel processing of the phases does not look possible, possible improvements in the efficiency of implementation of the semantic restrictions may be rather limited.

2.2.2 GAMS

GAMS (**General Algebraic Modeling System**) is a mathematical modeling language developed originally at the World Bank for use as a modeling tool in a variety of strategic planning environments. The language has a mature mainframe implementation and is also available on personal computers.

2.2.2.1 Semantic Restrictions

In GAMS, some semantic restrictions, or properties, are applicable to the model (Model Restrictions), while others are applicable to the data (Data Restrictions). These properties must be inferred from [Broo88], since GAMS does not define them explicitly. The GAMS processor always halts with numerous error messages when a semantic restriction is violated.

The basic entities of a GAMS model are statements for sets, parameters, variables, and equations. Other entities may include commands that generate output; that use special operators which delimit the scope of a variable or equation; and that use the relational capabilities of the GAMS system. A typical GAMS model will include some of the following entities:

- * GAMS Input
SETS

```

PARAMETERS
VARIABLES
EQUATIONS
MODEL AND SOLUTION STATEMENTS
* GAMS Output
ECHO PRINT
REFERENCE MAPS
EQUATION LISTINGS
RESULTS

```

Example. The following GAMS statements describe the transportation problem:

```

*Model and Data.
SETS
  I set of plants /DALLAS,CHICAGO/
  J set of customers /PITTSBURG,ATLANTA,CLEVELAND/;
PARAMETERS
  S(I) supply capacity (in tons) for plant i
        /DALLAS 20000,CHICAGO 42000/
  D(J) nonnegative demand (in tons) for customer j
        /PITTSBURG 25000,ATLANTA 15000,CLEVELAND 22000/
  C(I,J) transportation cost ($ per ton) from plant i to customer j
        /DALLAS.PITTSBURG = 23.50, DALLAS.ATLANTA = 17.75
        DALLAS.CLEVELAND = 32.45, CHICAGO.PITTSBURG = 7.60,
        CHICAGO.CLEVELAND = 25.75/;
VARIABLES
  F(I,J) nonnegative flow (in tons) from plant i to customer j
  TOTCOST total cost (in $ per ton) ;
POSITIVE VARIABLE F ;
EQUATIONS
  OBJ
  TSUPPLY(I)
  TDEMAND(J) ;
OBJ .. TOTCOST =E= SUM((I,J), C(I,J)*F(I,J));
TSUPPLY(I) .. SUM(J, F(I,J)) =L= S(I) ;
TDEMAND(J) .. SUM(I, F(I,J)) =E= D(J) ;
MODEL TRANSPORT /ALL/ ;
SOLVE TRANSPORT USING LP MINIMIZING TOTCOST ;
DISPLAY TOTCOST.L, F.L, F.M;

```

The ordering of statements in a GAMS model must be such that an entity of

the model cannot be referenced before it is declared to exist <MR1>; i.e., forward references are not allowed. In addition, **any declaration of a model entity must be unique <MR2>**; i.e., parameter and sets may not be declared more than once. Although entities may not be declared more than once, entities may have values defined (i.e., assigned to them) multiple times. **Entity names and labels (set and parameter elements) must be distinct from a list of GAMS reserved words <MR3>**. Statements generally are terminated by semicolons.

GAMS offers three different formats for entering data: lists, tables, and direct assignments. **Every domain element in a list, table, or assignment, must be a member of the appropriate SET <DR1>**. The GAMS compiler performs this verification called “domain checking”, and issues an error message when an element does not belong to a set. For example, in the specification SET T(I), GAMS will check that every member of the set T is also a member of a previously defined set I.

However, if a parameter is not domain checked, the only restriction is that the dimensionality be kept constant; i.e., **identifiers must retain constant dimensionality <MR4>**. Once the number of labels per data item has been defined, it is illegal to refer to the parameter differently.

A TABLE, like a SET, can have up to 10 dimensions <MR5>. Of course, the number of labels associated with each number in the table must coincide with the number of domains in the domain list <MR6>.

GAMS includes various kinds of sets, namely static and dynamic, and ordered and unordered. Sets may have ALIAS names, **which must be unique** <MR7>. Static sets may have their members assigned directly in the model. The general principle followed is that **data must be non-redundant** <DR2>; i.e., each SET and PARAMETER data item (i.e., label or label-tuple combination) must be entered only once.

One way to assign members to a set, or parameter, is the asterisk notation. It applies to cases when elements follow a sequence, e.g.,

```
SET T time periods / 1991 * 2000 /;
SET M machines      / MACH-1 * MACH-24 /;
```

Obviously, for the data assignment to be meaningful, **both operands of the * operator may only differ in characters that are digits** <DR3>. Any non-numeric differences or other inconsistencies cause errors.

GAMS also provides decision VARIABLES. A variable has a lower bound and an upper bound, both of which are either set explicitly in the model, or take default values. At all times, **the lower bound cannot be greater than the upper bound** <DR4>. If the model violates this condition in an explicit bounds declaration, GAMS will exit with an error condition. Additional restrictions on variables are that **BINARY variables must have bounds of 0 and 1 or be fixed**, and that **INTEGER variables must have non-negative integer bounds or be fixed** <DR5>. If a variable appears more than once, **the suffix indices of the variable must match, in number and order, in all appearances** <MR8>.

The assignment statement (denoted by the '=' symbol) is the fundamental data manipulation statement in GAMS. An assignment statement can be indexed by a controlling index or controlling set. In an indexed assignment, **any controlling index on the right of the = sign must be matched on the left** <MR9>. In addition, **the domains of all controlling indices must be subsets of those declared for the corresponding sets, parameters, variables, or equations** <MR10>.

Standard arithmetic operations can be used on the right of the = sign. Indexed arithmetic operations are SUM, PROD, SMIN, and SMAX. The most common of these is the summation, written $SUM(\textit{index of summation}, \textit{summand})$. The *summand* is the scope of the controlling *index of summation*. **The controlling index cannot appear outside its scope** <MR11>.

GAMS provides common standard functions, like exponentiation, logarithms, and the trigonometric functions. **The number of arguments in use must be as stated by the GAMS standard function definition** <MR12>.

Also included in GAMS are binary set operations like union, difference, and intersection, and operators that determine the size of a set (CARD), that determine the position of an element in a set (ORD), and that perform the operation called "such that" ('\$') which can act as an assignment operator. The dollar operation can be controlled with a set (static or dynamic) or with a numeric valued parameter, but **cannot mix both** <MR13>. The ORD function can only be used **with a one-dimensional static, ordered SET** <MR14>.

Data types in GAMS are either “value” type or “set” type, and assignments must be of one type or the other. For all operators, with the exception of the dollar operator, **value type and set type operands cannot be mixed** <MR15>. For the dollar operator, some mixed mode operands are allowed.

GAMS also provides a LOOP statement. LOOPS may be nested and controlled by more than one set. **It is illegal to modify a controlling set inside the body of the loop** <MR16>.

The DISPLAY statement is used with parameters to write into the output file all values associated with identifiers. The format of this statement is the reserved word DISPLAY, followed by a comma-delimited list of “identifiers”, where identifier is the name of an **initialized or assigned** <MR17> set or parameter.

With regard to GAMS equations, by default, when the model class is unspecified, GAMS will examine the equations and determine the model class and solver to be used. However, when the model class is specified, **the mathematics in the equations must match the model class** <MR18>. Thus, linear programming models cannot contain nonlinear terms, nonlinear programming models cannot contain discrete variables, etc. Furthermore, **equations must be mathematically consistent** <MR19>. For example, all indices used in equations must match, in order and number, in all repeated appearances. More generally, all appearances of set names and labels must be consistent with the dimensionality and domains of the identifiers.

In GAMS, symbols may have values that are only determined subsequently

to their declaration. At compile time, however, **each symbol must have a value assigned to it before it can be referenced in some subsequent assignment statement** <DR6>. An attempt to use data associated with an unassigned symbol (like a parameter) will cause an error condition.

Other restrictions exist on GAMS. Some “discontinuous” functions (**CEIL, TRUNC, SIGN, etc.**) may not have endogenous arguments <MR20>; i.e., may not have “unknown”, or variable, arguments. In addition, **dollar operations cannot contain references to GAMS variables** <MR21>; only exogenous data or logic is permitted.

In the case of optimization, the **GAMS model must declare a scalar variable (i.e., with no domain) to serve as the quantity to be minimized or maximized** <MR22>. During compilation, GAMS checks that the objective variable is used in at least one of the equations.

2.2.2.2 Run-Time versus Static Model Checking

In GAMS, the model structure and the model instance (data) are mixed, although it is possible to maintain them separately. Because this separation between model class and model instance is not always explicit, many model and data checks must be deferred to run-time (generation-time). Hence, it is not possible to completely check a GAMS model statically. Some of the semantic restrictions checked at run-time are now identified.

Illegal arithmetic operations such as **division by zero, or taking the log of**

a **negative number** <DR7>, cause execution time errors. The exponentiation operator in $X^{**}Y$ computes a real number raised to a real power. **This operation will result in a run-time error if X has a negative value, or if Y is not a real number** <DR8>.

Although not illegal, to avoid the risk of having GAMS treat a value as undefined, the model should not create or use numbers larger than 1.0E+20, or smaller than 1.0E-20. Arbitrarily large or small numbers can be set to INF, or -INF.

Lag operations on the domain of an equation should not lead to **multiple definitions of the same single equation** <MR23>. This condition can only be checked at model generation time.

If a solver fails to find a solution, any following SOLVE is not attempted and GAMS treats this as a fatal error.

2.2.2.3 Unsafe Constructs

In general, GAMS allows its binary set operators, union, intersection, and difference, to take as operands dynamic sets drawn only from identical domains. (Dynamic sets, as opposed to static sets, may change their membership via assignments). However, this semantic restriction is not enforced for sets of two or more dimensions, as shown in the following set valued assignment:

```
SETS
  C      set of colors   /BLUE, ORANGE/
  CSUB(C) subset of colors /BLUE/
  P      set of plants   /100, 200/
  PSUB(P) subset of plants /200/
```

```
FAULTY;  
FAULTY(C,P) = CSUB(C) + PSUB(P);  
DISPLAY FAULTY;
```

The computed set FAULTY contains the tuples (BLUE,100), (BLUE,200), and (ORANGE,200). This set contains all tuples whose first component is BLUE (i.e., is a member of CSUB(C)) and second component is a member of P, together with all tuples whose second component is 200 (i.e., is a member of PSUB(P)) and first component is a member of C. Note that the tuple (ORANGE,200) is included in the set FAULTY, even though ORANGE is not a member of CSUB(C). GAMS takes the viewpoint that FAULTY is defined over the universe and actually stands for “set faulty(*,*)”, which allows expressions like faulty(p,c), faulty(*,csub), etc. It is apparent that this kind of set operation is not closed; therefore we consider it unsafe.

2.2.2.4 Diagnostic System

During the compilation process, GAMS provides a listing of the input file, a list of any error messages, and reference maps. During the execution process, the only normal output produced may come from a DISPLAY statement. Useful statistics, like model “generation time”, and model “execution time” are available in a summary.

When GAMS encounters a compilation error, it marks the offending position with a ‘\$’ sign and a number on a line that starts with 4 asterisks “****”. The section labeled ERROR MESSAGES explains the probable cause of the error.

Some errors will only be detected on lines following where they actually occurred; hence, the messages may sometimes not show exactly where the error has occurred.

Below is a fragment of the GAMS transportation problem which includes the duplicate ATLANTA customer and the invalid BOSTON demand.

```
*Database.
SETS
  J set of customers /PITTSBURG,ATLANTA,CLEVELAND,ATLANTA/;
PARAMETERS
  D(J) nonnegative demand (in tons) for customer j
      /PITTSBURG 25000,ATLANTA 15000,
      CLEVELAND 22000,BOSTON 25000/;
```

Both errors, the duplicate customer in the J set and the unknown demand in the D(J) parameter, are caught by the GAMS translator which, in addition to inserting the '\$' sign below each offending token, labels the ATLANTA line with the number 172 and the BOSTON line with the number 170. The ERROR MESSAGES section shows the following error codes:

```
170 DOMAIN VIOLATION FOR ELEMENT
172 SYMBOL IS REDEFINED
```

When GAMS encounters an execution error caused say, by an illegal instruction, GAMS writes an error message in the output file and continues execution. This behavior is possible because, notably, GAMS provides an extended algebra that contains all operations, including illegal ones. An “extended range” arithmetic is used to handle missing data, the results of undefined operations, and bounds regarded by the solver system as infinite. Hence, all run-time exceptions are trapped, and after writing an error message in the output file, GAMS treats the result as undefined and continues execution. However, under solver subsystem

control, the subsystem may report any arithmetic exception in the output file and interrupt the solution process.

There is another way of annotating the GAMS output file with useful information, and also ensuring data integrity. GAMS provides the ABORT statement to terminate a GAMS task if some logical or numerical condition does not hold. At run-time, when the stated condition does not hold, the identifiers and the message provided in the ABORT statement are output, and the processing is terminated with an error condition. Otherwise, if the condition holds, execution will continue normally.

The amount of diagnostic output can also be controlled with options of the SOLVE statement. Options can be chosen to provide debug and explanatory information, and to terminate the execution when a certain number of illegal instructions performed by the solver has been exceeded.

2.2.2.5 Readability and Writability

GAMS does not encourage the separation between model structure and data. For example, the same statement can be used for the declaration of data and its assignment of values in list-format. Hence, data tables, which are usually less-readable, get commingled together with the model. From the point of view of semantic restrictions, this commingling adds more complexity to their implementation.

GAMS provides two ways to include comments in a model. The first way is to

start a comment with a '*' symbol in the first character position. This comment may extend to the end of the line. The second way is to include any block of text between the delimiters \$ONTEXT and \$OFFTEXT. These delimiters must also be in the first character position, and there are no restrictions as to the length of the block.

GAMS also includes a self-documentation feature that facilitates the reading of the models. Explanatory text can be embedded within the GAMS statements immediately following a name declaration. The syntax of this optional text is the following: it is a sequence of characters not starting with “.” or ‘=’, and is separated by at least one blank space from the preceding symbol, which must be on the same line; the sequence is terminated by one of the three symbols, “, / ;”, or by the end-of-line, or by a quoted string not longer than 80 characters long. The explanatory text and model comments are not subject to semantic restrictions and consequently, cannot be used by a specialized tool to enhance model correctness.

2.2.2.6 Implementability

A GAMS program is compiled with the GAMS compiler, which produces a set of instructions that can be executed in the GAMS Symbolic Machine. The instructions generated by the GAMS compiler are not tied to any single computer hardware.

The operation of the GAMS Symbolic Machine has been described in [Eijk83]. This machine is composed of the following six components:

- *The Symbol Table.* It is created by the compiler and contains an entry for every symbol (name of set, parameter, variable, equation, etc.) listed in a GAMS source program. Every entry in the Symbol Table contains an index number and a record accessed through a pointer. This structure provides quick access to a Symbol Table Entry based on the index number which, if known, leads to the pointer in the Symbol Table. Access by name is made efficient by maintaining a hash table, which given a symbol name, can translate it to the corresponding index number.
- *The Unique Elements Table.* This table is created by the compiler and contains an entry for every unique alphabetic element (set element, parameter element, etc.) mentioned in a GAMS source program.
- *The Instruction Format.* The instructions in the GAMS Symbolic Machine are of fixed length and are all stored in an array of instructions.
- *The Program Counter.* It is an index which is used to indicate the current instruction being executed in the array code.
- *The Execution System Stacks.* The execution system operates with three stacks: the Control Stack (which is used for controlling loops and holds the current value of the loop index, the corresponding element, the controlling set information and the boundaries of the loop), the Index Stack (which contains unique element numbers), and the Value Stack (which contains the

real values for unary and binary operators as well as the fetch and store operations). All operations are executed on the Value Stack.

- *Miscellaneous Variables*. These are variables defined for diagnostic and options.

From this architectural layout, it is clear that the translation of GAMS code into an intermediate symbolic representation has been highly optimized. Hence, GAMS is able to implement its semantic restrictions very efficiently.

2.2.3 LINGO

LINGO is a recent linear programming modeling language from the developers of the popular LINDO [Schr86]. Versions of the language are available for different environments, including PC/DOS and Macintosh.

2.2.3.1 Semantic Restrictions

Very little has been published about the semantic restrictions enforced by LINGO. The Model Restrictions and Data Restrictions given below have been inferred from [Cunn89], and from the error messages encountered while using LINGO.

Two solvers are available in LINGO: the direct solver, and the simultaneous equation solver/optimizer which **can only solve linear models** <MR1>. The solver to be used is determined by LINGO from examination of the structure and mathematics of the model.

The direct solver tries to compute values for all the variables in the relations or constraints. It starts by computing values in relations with only one variable. When the values of all variables have been computed, the direct solver stops and the solution is printed. If the value of a variable in some constraint cannot be resolved, the simultaneous equation solver/optimizer is called. This solver first checks that all expressions are linear after variables have received their values. If the model is not linear, LINGO halts and an appropriate message is printed. If the solver determines that the model is linear, it attempts to find the solution. General information about the mathematical programming methods used by the solvers is found in [Schr86].

Two kinds of models exist in LINGO: basic “direct” models, and set-based systems of simultaneous equations and linear programs. Direct models have variables which are not interdependent and, hence, can be directly computed by LINGO’s direct solver. On the contrary, set-based systems of simultaneous equations and linear programs have variables whose values cannot be directly determined, but which can be solved by LINGO’s simultaneous equation/solver as long as the equations are linear.

A direct model in LINGO is an unordered list of statements; each statement is terminated by a semicolon, and may possibly span multiple lines. Since the order of statements in the direct model is irrelevant, the same results are obtained with a different order. The statements declare a condition that each variable must satisfy, as opposed to a rule for computing that variable; LINGO takes care of computing a

value for each statement variable. Apparently there are no implemented semantic restrictions applicable to variables in direct models.

A set-based model in LINGO is a collection of statements; they are split between one or more set definition sections, constraints section(s), and data section(s). A set section is delimited by the tokens "SETS:" and "ENDSETS". A constraints section is optional, and typically includes equations involving the model entities (i.e., its scalar and set variables). For an optimization model, the constraints section includes an equation beginning with "MAX=", or "MIN=", followed by the expression to be maximized or minimized. In any optimization model, **there can only be one expression to either minimize or maximize** <MR2>. Finally, a data section is delimited by the tokens "DATA:" and "ENDDATA". There are no reserved words in LINGO.

Example. A LINGO model for the transportation problem is:

```
!Model and Data;
SETS:
    PLANTS      /DALLAS, CHICAGO/ : SUPPLY ;
    CUSTOMER    /PITTSBURG, ATLANTA, CLEVELND/ : DEMAND ;
    LINK(PLANTS, CUSTOMER) : COST, FLOW ;
ENDSETS
! The objective function;
[TOTCOST] MIN = @SUM(LINK: COST * FLOW) ;
! The supply constraint;
    @FOR(PLANTS(I) :
        @SUM(CUSTOMER(J): FLOW(I,J)) < SUPPLY(I)) ;
! The demand constraint;
    @FOR(CUSTOMER(J) :
        @SUM(PLANTS(I): FLOW(I,J)) > DEMAND(J)) ;
DATA:
    SUPPLY = 20000, 42000 ;
    DEMAND = 25000, 15000, 22000 ;
```

```

        COST      = 23.50, 17.75, 32.45,
                   7.60,  0.0,  25.75 ;
ENDDATA
END

```

Sets are declared in the SETS section of a model. The syntax for a set declaration is:

```

set_name / identifiers / [:attribute(s)];

```

where the brackets around *attribute(s)* denote optionality. In LINGO, all constants, attributes, set names, and corresponding identifiers, must be globally unique <MR3>.

Set attributes are normally assigned values in the DATA section. The syntax for a data assignment is:

```

attribute = value_list;

```

Of course, the *value_list* must have the same number of numeric values as those of the attribute it initializes <DR1>.

A derived set (relation) is a set constructed from one or more other (primitive or derived) sets, a subset of another (primitive or derived) set, or combinations of elements in these other sets. Globally, i.e., throughout the model, sets must be declared before they can be referenced <MR4>.

An individual element of a derived set may be accessed with a set operator expression. The set operators in LINGO are: @FOR, @SIZE, @SUM, @MIN, and @MAX. A set operator expression is of the form:

```

set_operator(set_part:expression);

```

The *set_part* specifies a set name, optionally followed by a comma-delimited list of indices which are mentioned in *expression*. **The set's name and the number of indices, if present, in *set_part* must coincide with the name and dimension of a previously defined set <MR5>.** Any indices present in *set_part* are called bound indices in *expression*. Any indices used in *expression*, and which are not mentioned in *set_part*, are called free indices. In any indexed variable or indexed set within *expression*, **the number of indices used must coincide with the dimension of the corresponding declaration <MR6>.**

2.2.3.2 Run-Time versus Static Model Checking

It is difficult to distinguish between static checking and dynamic checking in an interpreted language like LINGO. In abstract terms, it is possible to think that most of the pre-solver activity corresponds to static checking, while the solver's actions correspond to run-time model checking. The two solvers in LINGO either compute a solution, or return an error message.

2.2.3.3 Unsafe Constructs

In at least two areas, LINGO does not have appropriate semantic restrictions, and thus has safety problems. The first area is in the treatment of identifiers in derived sets, and the second one is in the treatment of bound indices in set-operator expressions. The examples below illustrate each of these concerns.

In LINGO, identifiers in derived sets do not satisfy the equivalent of semantic

restriction <MR2> for their primitive constituents. For example, the model

SETS:

```
CU          / PIT, ATL / ;
PL          / 1..2 / ;
LINK(CU,PL) / PIT 1, PIT 2, ATL 1, PIT 1, ATL 2 / ;
```

is legal in LINGO, although the tuple (PIT 1) is duplicated. Hence, although this model contains a mistake, it is accepted by the language.

The treatment of bound indices in set-operator expressions may also be unsafe.

For example, the following statement may be added to the above model:

```
@FOR( PL(I): @SUM( LINK(I,J) :1 ) >1 );
```

In this legal statement, the indices in LINK are running over a set of (PL,PL) identifier-pairs, rather than over its defined set (CU,PL). This kind of liberty does not protect the modeler from making various kinds of mistakes.

2.2.3.4 Diagnostic System

In the LINGO environment, the GO command orders LINGO to start solving the current model. In the event of model errors, LINGO displays a line with an error message, then the offending line, and then it marks the offending position with a ' ^ ' symbol. LINGO stops at the first appearance of an error.

In LINGO, the database containing the duplicate ATLANTA customer in the transportation problem will be described as follows:

SETS:

```
CUSTOMER    /PITSBURG, ATLANTA, CLEVELND, ATLANTA/ : DEMAND ;
```

The duplicate customer in the CUSTOMER set is caught by the translator, which issues the following message:

```
INVALID PRIMITIVE SET ELEMENT NAME
4] CUSTOMER / PITSBURG, ATLANTA, CLEVELND, ATLANTA/ :DEMAND ; [
      ^
```

A demand for BOSTON can be added to the data section as follows:

```
DATA:
    DEMAND = 25000, 15000, 22000, 22000 ;
```

The invalid demand for BOSTON is caught by the translator, which issues the following message:

```
INVALID NUMBER OF ARGUMENTS IN DATA STATEMENT.
18] , 15000, 22000, 22000;
      ^
```

2.2.3.5 Readability and Writability

LINGO offers the modeler the choice to separate or to commingle model structure and data. For simple models, it is convenient to list the data immediately following each set declaration. For large-scale or more complex models, a DATA section can be used to store the data following the MODEL section. Alternatively, the *@FILE(file)* function can be used to input model text or data from a *file*. The *@FILE(file)* function reads the text from the input *file* until it finds the end-of-file or a LINGO end-of-record marker (`~`). *@FILE* behaves like the include command of programming languages, with the limitation that nesting is not allowed. From the viewpoint of semantic restrictions, the choices available for modeling both structure and data may add complexity to the implementation.

Comments in LINGO begin with a `!` symbol and are terminated by a semicolon. They can start anywhere in a line, and extend over multiple lines. Although

no other self-documentation features exist in LINGO, the models written in this language are easy to read. However, since comments are free-format and not subject to semantic restrictions, they cannot be used by a specialized tool to verify model correctness.

2.2.3.6 Implementability

There are no details available about the architecture of LINGO. From its user environment, however, it appears that the language is interpreted. Hence, notwithstanding some advantages of interpreted over compiled modeling languages in areas such as flexibility, the implementation of semantic restrictions may not be amenable to be optimized for speed of execution.

2.3 Discussion

The leitmotiv of this chapter has been that explicit context-sensitive semantic restrictions are of enormous importance to preventing and detecting modeling errors in mathematical programming modeling languages. Our beliefs are that, in modeling, errors are the norm rather than the exception, and that models cannot be assumed correct until proven so. The following comments are stimulated by the appearance of this recurring theme in Section 2.2.

Semantic Restrictions. None of these modeling languages for mathematical programming have published formal semantic definitions (neither in attribute grammar style nor in any other style, such as denotational semantics, operational

semantics, or axiomatic semantics), although almost all of them have published formal syntactic ones. The languages rank as follows, in order of greatest to least difficulty in identifying semantic properties: LINGO, AMPL, GAMS.

There is a need for rigorous and unambiguous specifications of the semantics of these mathematical programming modeling languages, so that confidence in the correctness of the models and their data can be strengthened, and so that the current approach to mastering these languages by experimentation can be abandoned. Consequently, we recommend that accurate context-sensitive semantic descriptions be included in language definition manuals of modeling languages for mathematical programming. Restrictions should be properly motivated and categorized, and the ones which cannot be properly classified, should be relegated to the group of pragmatics.

Run-Time versus Static Model Checking. None of these modeling languages for mathematical programming can detect all errors via static (translation-time) checks alone; some run-time (i.e., generation-time or solver-time) model checking is required. The languages rank as follows, in order of least to greatest conduciveness to having error checks performed at static time: LINGO, AMPL, GAMS.

For purposes of handling large models efficiently (i.e., having the model generator run as fast as possible and, at the same time, making the modeling task simple and less error-prone by providing feedback as early as possible), it's better to enforce semantic restrictions statically. Hence, to be able to improve error-checking

services, we believe strongly that designers should identify both the static and the dynamic (or run-time) semantic restrictions of their languages.

Unsafe Constructs. The languages rank as follows, in terms of largest to smallest number of unsafe constructs: AMPL, LINGO, GAMS. The highest degree of safety is offered by GAMS, since this language has defined a large number of semantic restrictions. LINGO places fewer semantic restrictions on its models than GAMS, leading to more safety problems. AMPL has the largest number of unsafe constructs, from our perspective, since it has not clearly defined semantic restrictions for some areas. However, we recognize that, on occasions, language designers make different design decisions deliberately.

Because any unsafe construct in a modeling language for mathematical programming makes modeling more error-prone and unreliable, we suggest that future language designers eliminate unsafe constructs through the identification of appropriate semantic restrictions and pragmatics (i.e., the system designer tells the modeler what practices are good and bad). Depending on the developers' intentions, this technique could be incorporated into the current mathematical programming modeling languages or their implementations as well.

Diagnostic System. The languages rank as follows in order of increasing friendliness of their diagnostic system: AMPL, LINGO, GAMS. The AMPL system has a simple diagnostic system: it lists error occurrences with sometimes understandable messages, and then stops since no error recovery mechanisms exist. Although LINGO stops at the first error occurrence, it provides better error

diagnostics than AMPL and, since the modeler does not have to leave the LINGO environment, it is easier to correct mistakes. Not surprisingly, considering the relative ease with which its semantic restrictions can be identified, GAMS has the most complete diagnostic and error recovery system of the three languages. At model translation-time, all model errors are clearly marked and, at model generation-time, all exceptions are trapped and written in the output file. Notwithstanding the fact that diagnostic systems are also a function of implementation strategies, this language illustrates the fact that a good diagnostic system can be constructed as a consequence of explicit semantic restrictions.

Readability and Writability. The languages rank as follows in order of increasing readability and writability: GAMS, LINGO, AMPL. In GAMS, the binding of data with model structure reduces the clarity of complex models, and adds complexity to the implementation of semantic restrictions.

Documentation can be extensively included throughout a model in all three languages, and hence they can be considered equally readable in this respect. However, since the documentation is not subject to semantic restrictions in any of the languages, it cannot be employed by automated tools to verify the correctness of a model and increase its reliability.

Implementability. The architecture of all three mathematical programming modeling languages is very different. AMPL and GAMS support batch mode operation, while LINGO supports an immediate computation mode. (It is possible, however, to use LINGO in batch mode by redirecting input from an input file,

and redirecting output to an output file). No data are available on the relative translation-time versus generation-time efficiency of the execution of semantic restrictions. Hence, it is not possible to compare how efficiently the different semantic restrictions are implemented on the current architectures.

CHAPTER 3

Formalization of Static Semantics via Attribute Grammars

The previous chapter examined the need to identify context-sensitive semantic restrictions in modeling languages for mathematical programming as a means of reducing modeling errors. This chapter examines a formal method of representing the context-sensitive aspects of mathematical programming modeling languages based on attribute grammars. We show how attribute-grammar based definitions can be used to enforce, in polynomial time, the complete static semantics of modeling languages for mathematical programming. From examples of formalization of various semantic restrictions of SML (**Structured Modeling Language**), we show that this kind of formalism is sufficiently perspicuous to facilitate the understanding of language semantics, and leads to complete, consistent, and correct specifications that are easy to maintain and modify.

The chapter is organized as follows. Section 3.1 provides a brief introduction to attribute grammar concepts. Then Section 3.2 defines some SML terminology which is necessary for reference purposes. Section 3.3 describes a methodology to derive an attribute grammar for SML. The time complexity of attribute evaluation is discussed in Section 3.4. Several examples, that show how specific semantic restrictions of model structures and model instances written in SML can be de-

scribed in attribute grammar terms, are given in Section 3.5. Section 3.6 discusses the importance of this methodology. Finally, Section 3.7 comments on some of the main difficulties and lessons learned from the description of SML via attribute grammar equations.

3.1 Attribute Grammars

Attribute grammars were first introduced by Knuth in [Knut68] and [Knut71] to assign meaning to derivation trees on context-free languages. Because of their clarity and brevity, attribute grammars have been used in an extensive range of applications, from compiler generation to language translation, theorem proving, and others (see e.g., [Aho86], [Farr82]). Language-based tools, like syntax-directed editors, can also be appropriately specified with attribute grammars.

An attribute grammar (AG) is a context-free grammar together with sets of semantic equations associated with the productions of the context-free grammar. Formally, $AG = (CFG, A, F)$, where

1. $CFG = (V_n, V_t, P, S)$ is a context-free grammar with V_n a set of nonterminal symbols, V_t a set of terminal symbols, and S the initial symbol. P is a set of production rules, $\{p_1, \dots, p_j\}$, where p_i ($1 \leq i \leq j$) is of the form: $p_i : X \rightarrow w_0 X_1 w_1 \dots X_n w_n$, $n \geq 0$, with w_0, \dots, w_n in V_t^* and X, X_1, \dots, X_n in V_n . The set of all terminal symbols in V_t , including the empty symbol, is denoted by V_t^* . CFG is called the underlying grammar for AG . We assume,

without loss of generality, that the initial symbol S never appears in the right hand side of any production rule.

2. A is a set of attribute symbols. Each nonterminal in production p_i has a (possibly empty) subset of these attributes A (there is no loss of generality in assuming that only nonterminals have attributes). An occurrence of an attribute symbol ' b ' of nonterminal X is written " $X.b$ ". Attributes have values from specific domains.
3. F is a set of semantic equations (analogous to a system of simultaneous algebraic equations) that specify the context-sensitive restrictions of the language generated by CFG . Semantic equations associated with a production p_i specify how the value of each attribute-occurrence in p_i is defined. The solution of these equations is an assignment of either a null value or a non-null value to every attribute-occurrence. This value can be computed from the value of other attribute-occurrences of the production.

Each attribute symbol is either *inherited* or *synthesized*. In a well-formed attribute grammar, the initial symbol S has no inherited attributes, and terminal symbols have no synthesized attributes. Inherited attributes can pass information "down" the tree from the root towards the frontier; synthesized attributes can pass information "up" the tree. The semantic equations associated with a production must define all synthesized attribute-occurrences of the left-hand-side (LHS) of the production and all inherited attribute-occurrences of the right-hand-side of

the production (RHS) - - these are called the output attributes. Furthermore, they cannot define any inherited attribute-occurrence of the LHS or any synthesized occurrence of the RHS - - called the input attributes. In a well-formed, reduced, attribute grammar, every nonterminal symbol, and no terminal symbol, is the left-hand-side of some production.

3.2 SML: Structured Modeling Language

SML, as described in detail in [Geof88], is a modeling language which provides a representation for the core concepts of SM (**Structured Modeling**). SM is a modeling approach intended to provide a foundation for the development of a new generation of modeling environments. It provides a framework of conceptual abstractions for representing a wide variety of models as “structured models”. An introduction to SM appears in [Geof87], and its “core” concepts and underlying theory are explained in [Geof89b].

Structured models consist of elemental structure together with qualifying generic structure and modular structure. Elemental structure is a collection of elements that is nonempty, closed, and acyclic. Generic structure is a partition of model elements that does not mix element types (possible element types are primitive entity, compound entity, attribute or variable attribute, function, and test). Each cell in the partition implied by generic structure is called a genus. A modular structure is an ordered tree whose leaves are the genera (plural of genus).

SML is composed of two parts: “Schema” and “Elemental Detail”. The Schema

is the text-oriented representation of general model structure. It is composed of the following five sublanguages:

1. Sublanguage for generic calling sequences. This language specifies the definitional dependencies among model elements.
2. Sublanguage for range statements. This language defines a set of permissible values of attribute and variable attribute elements for a whole genus.
3. Sublanguage for index set statements. This language helps to define the element population of a genus. Calling sequence feasibility also plays a major role in determining this population.
4. Sublanguage for generic rule statements. This language specifies genus-wide rules for determining the values of function and test elements.
5. Sublanguage for interpretation. This language provides explanatory comments about a typical element of a genus.

These sublanguages are specified by a context-free grammar augmented with context-sensitive restrictions given, in English prose, as Schema Properties. The Schema Properties include static semantics only.

Elemental Detail is the table-oriented data that supplements the Schema in order to specify a specific model instance. In a genuine (well-defined) structured model, the tables are properly structured, labeled, and loaded following Table Content Rules, which are restrictions on the data, also described in plain English.

An SML Schema consists of paragraphs which are denoted differently depending on the type of element represented. Each element type corresponds to one of the following choices (the square brackets denote optionality):

- Primitive entity. It is an atom (or undefinable “thing”) in SM. The corresponding genus paragraph is denoted:

```
GenusName [symbolic index] /pe/
[index set statement] [domain statement]
~| [interpretation] ~.
```

- Compound entity. It is a type of model element that is defined in terms of other primitive or compound entities. The corresponding genus paragraph is denoted:

```
GenusName [symbolic index] (generic calling sequence) /ce/
[index set statement] [domain statement]
~| [interpretation] ~.
```

- Attribute. It is a type of model element that bears a user-specified value. Some attributes classified as “variable” may bear a value that is solver-specified. The corresponding genus paragraph is denoted:

```
GenusName [symbolic index] (generic calling sequence) /a/
[index set statement] [domain statement] [range statement]
~| [interpretation] ~.
```

- Function. It is a type of model element that bears a numeric value computed by a mathematical rule. The corresponding genus paragraph is denoted:

```
GenusName [symbolic index] (generic calling sequence) /f/
[index set statement] [domain statement] ; generic rule
~| [interpretation] ~.
```

- Test. It is similar to a function element, but the value is of logical type. The corresponding genus paragraph is denoted:

```
GenusName [symbolic index] (generic calling sequence) /t/
[index set statement] [domain statement] ; generic rule
~| [interpretation] ~.
```

In the above denotations, *symbolic index*, together with possible alias indices, behave like a dedicated index symbol in conventional mathematics.

3.3 Formalization Methodology

To formalize the static semantic aspects of SML via an attribute grammar, the following four items must be specified:

1. Attributes attached to each nonterminal symbol in the productions of the context-free grammar that generates SML (specifying also attribute domains -- which may be structured -- and whether they are synthesized or inherited).
2. Semantic equations and their corresponding context-free production rules. (These are the attribute evaluation rules associated with each of the grammar's production rules. Attribute evaluation rules must include both the Schema Properties and the Table Content Rules).
3. Auxiliary functions, if necessary (to aid the evaluation of semantic equations).
4. Local attributes, if necessary. These attributes are associated with a particular production rather than to a nonterminal symbol, and are used in

place of synthesized or inherited attributes. For example, a local attribute *error*, corresponding to a domain of error messages, can be declared among a production's semantic equations. The value of *error* is determined by its corresponding semantic equation (*error* = ...). Then in a modeling environment, *error* would be represented in the output display as a string containing an error message, or as the empty string.

For a number of practical reasons, rather than employing the concrete syntax implied by SML's context-free grammar to specify an attribute grammar, it is preferable to employ an abstract syntax generated by an abstract grammar. Abstract syntax mimics concrete syntax, except that it usually ignores details like terminal symbols, operator precedence, and similar syntactic adornments. In abstract syntax, a production rule is written in prefix notation, with an operator symbol used as a label and a list of space-delimited arguments enclosed in parenthesis. (In a modeling system, parse trees are internally manipulated using abstract syntax, where operators are the nodes of a tree, and the arguments of the operator are the children of the tree). In addition, every abstract syntax rule has a distinguished production called the "completing production", which plays a special role in the specification of syntax-directed editors. This role is explained in Section 6.1, "A Syntax-Directed Editor Description in SSL". Usually, the first operator in every production is taken as the completing production.

The following example shows some of the notational differences between concrete and abstract representations in SML.

Example. Whereas the concrete BNF syntax for a numeric-valued *expression* in the *generic rule* of a function or test genus paragraph may be given by

```
Expression ::= Term
             | MINUS_SIGN Term
             | Expression PLUS_SIGN Term
             | Expression MINUS_SIGN Term
```

its abstract counterpart might be represented by

```
expr:      ExprNull()
           | TermSingle(term)
           | TermPair(MINUS_SIGN term)
           | PlusPair(expr PLUS_SIGN term)
           | MinusPair(expr MINUS_SIGN term)
```

One of the main reasons for maintaining an abstract syntax is to handle incomplete model fragments. A Schema that may be partially complete, from the point of view of the modeler, is always a complete derivation from the point of view of a modeling system founded on attribute grammars. The following example illustrates this distinction for SML.

Example. The concrete incomplete expression, *PLUS_SIGN Term*, will exhibit the complete abstract representation:

```
PlusPair(ExprNull PLUS_SIGN term).
```

To determine which kind of attribute (i.e., synthesized or inherited) should be associated with a particular nonterminal symbol, the following criteria may be considered:

- If the value of the current attribute depends on the value of the same attribute at another left sibling node in the abstract tree, then the attribute should be synthesized by the left sibling node, and *inherited* by the current node.
- If the value of the current attribute depends on the value of the same attribute at another right sibling node in the abstract tree, then the attribute should be synthesized by the right sibling node, and *inherited* by the current node.
- If the value of the current attribute depends on the value of the same attribute at the parent node in the abstract tree, then the attribute should be *inherited* by the current node.
- If the value of the current attribute depends on the values of attributes at the children of the node in the abstract tree, then the attribute should be *synthesized* by the current node.

3.4 Complexity

An attribute grammar for SML may be *evaluated* on a particular input model by:

- building a parse-tree for the given input model in which every parse-tree node for a grammar symbol X is implemented as a record, and the fields correspond to the attributes of X ,

- traversing over this parse-tree and evaluating the fields of the records by calling the semantic equations, using the values of other node fields as arguments.

The different evaluation strategies correspond to different ways of visiting the nodes of the tree. The result of the attribute evaluation process is the value of synthesized attribute field(s) of the root node.

It has been established that the update of a distinguished attribute of an n -attribute syntax tree can be computed in a linear number of instructions, yet at a cost of storing only $O(\sqrt{n})$ attribute values at any stage [Reps83]. Hence, the process of evaluating each attribute (also called attribution) can be performed in a polynomial number of recomputations, for any modeling language.

3.5 Examples of Schema Properties and Table Content Rules

The complete attribute grammar for SML is given in [Vicu90]; there are around 3900 lines of attribute grammar equations, which are embedded in more than 7800 lines of pseudo attribute grammar code for the syntax-directed editor prototype described in Chapter 6. The prototype's attribute grammar equations are tested on a set of suitable test models in Section 6.2. The following subsections include various examples of the application of the formalization methodology. Several quotes, taken from [Geof88], of SML's Schema Properties and Table Content Rules are provided, followed by the attribute grammar equations enforcing these semantic

restrictions. Schema Properties are flagged either $\langle\langle Pn.i \rangle\rangle$, or $\langle\langle Pi \rangle\rangle$. In the former case, the label indicates that Schema Property i was defined in Appendix n of [Geof88], while in the latter case, the label refers to the i 'th Schema Property defined in Section 2. Table Content Rules are tagged $\langle\langle TCR.x \rangle\rangle$, where x is an upper-case alphabet letter.

A formal context-free grammar for SML, with its rules rendered as a Schema, appears in Appendix 6 of [Geof88]. An abstract syntax for SML, derived from the concrete syntax implied by the context-free grammar, is given here in Appendix A. The notation is basically unextended BNF, with the root node labeled “*schemafile*”, and leaf nodes in all upper case letters (such as the module name *MNAME*). The left-hand side of every production rule is to the left of the colon symbol (‘:’), and the corresponding right-hand side is the set of grammar symbols enclosed in parenthesis, and labeled with an operator name. Right-hand side rules associated with the same left-hand side nonterminal are separated by a bar (‘|’) symbol. Individual production rules are separated from each other by a semicolon symbol (‘;’).

3.5.1 Equations for Schema Overview

Example 1: Three restrictions dictated by the semantics of SML:

$\langle\langle P1 \rangle\rangle$ *The indentation must be in multiples of K blanks, where K is a positive integer.*

<<P2>> For each genus paragraph, the indentation of the following paragraph, if any, must not exceed the indentation of the genus paragraph.

<<P3>> For each module paragraph, the indentation of the paragraph immediately following must be exactly one K -multiple greater than that of the module paragraph.

The (mandatory) indentation is defined by the production rules:

```

schema_par:
  SchemaNil()
  | SchemaPair(paragraph schema_par) ;
paragraph:
  SubschemaNil()
  | SubschemaModPar(indent_or_null modparagraph)
  | SubschemaGenPar(indent_or_null genparagraph) ;
indent_or_null:
  IndentOrNullEmpty()
  | IndentOrNullNonEmpty(BLANKSPACE) ;

```

An inherited attribute *env*, corresponding to a domain of environments *ENV* (see below), can be associated with the nonterminals *schema_par*, *paragraph*, and *indent_or_null*. A synthesized attribute *ind* representing indentation, and belonging to the domain of integers, can be associated with the nonterminals *paragraph* and *indent_or_null*. The constraints can be specified by the following equations:

```

schema_par:
  SchemaNil { }
  | SchemaPair {
    paragraph.env      = schema_par$1.env;
    schema_par$2.env  = Binding(paragraph.ind,paragraph.ids,*,*,
    paragraph.typ,*,*,*,*,*,*,*,*,*)::schema_par$1.env; } ;
paragraph:

```

```

SubschemaNil() {
  paragraph.ind      = 0;
  paragraph.ids      = "";
  paragraph.typ      = '?'; }
| SubschemaModPar {
  indent_or_null.env = paragraph.env;
  modparagraph.env   = paragraph.env;
  paragraph.ind      = indent_or_null.ind;
  paragraph.ids      = modparagraph.ids;
  paragraph.typ      = 'm'; }
| SubschemaGenPar {
  indent_or_null.env = paragraph.env;
  genparagraph.env   = paragraph.env;
  paragraph.ind      = indent_or_null.ind;
  paragraph.ids      = genparagraph.ids;
  paragraph.typ      = genparagraph.typ; } ;
indent_or_null:
  IndentOrNullEmpty {
    local ERR error;
    error              = ErrP1();
    indent_or_null.ind = 0; }
| IndentOrNullNonEmpty {
  local ERR error;
  local INT initspaces;
  local INT tempspaces;
  tempspaces = STRlen(BLANKSPACE)-RSTRINGindex(BLANKSPACE,'\n');
  indent_or_null.ind = tempspaces;
  initspaces        = with(indent_or_null.env)(
    NullEnv: tempspaces,
    default: indent_lookup(indent_or_null.env));
  error              = (
    ((tempspaces-initspaces)%K != 0) || (tempspaces<initspaces) ?
    ErrP1(): with(indent_or_null.env) (
      NullEnv: NoErr(),
      EnvConcat(b, e):
      with (b)( Binding(i,s,*,*,*,*,*,*,*,*,*,*,*,*):
        /* is the previous paragraph a module? */
        (s[1]=='&' && (i>=tempspaces || tempspaces-i!=K)) ?
        ErrP3():
        (s[1]!='&'&&i<tempspaces)?ErrP2():NoErr() )); } ;

```

The environment is the most important attribute in the specification, since it

contains data about all Schema paragraphs. The domain of environments *ENV* is a list of *Binding*'s, one for each paragraph. Each *Binding* is a structure of type *BINDING*, which holds information about an individual paragraph, such as its indentation, name, and other relevant data. The environment type *ENV* is defined as follows:

```
ENV:
    NullEnv()
    | EnvConcat(BINDING ENV) ;
BINDING:
    Binding(INT STR STR ALIAS CHAR DEP COMS STR
            INT RANGE ISS COMPS SYMPARAM INTERP) ;
```

We ignore all components in a *Binding* except for the first two and the fifth: the *INT*eger-valued indentation level, the *STR*ing-valued paragraph name, and the *CHAR*acter-valued paragraph type.

In the attribute equations, the local attribute *error*, corresponding to a domain of error messages, is assigned the appropriate error operator: *ErrP1* for a violation of <<P1>>, *ErrP2* for a violation of <<P2>>, *ErrP3* for a violation of <<P3>>, or the null operator *NoErr* in case of no violation of these semantic restrictions. The workings of these equations are now explained.

The lexical value of the terminal *BLANKSPACE* is a collection of one or more blank spaces, new lines, or tabs. The number of blank spaces in the first line of the current *paragraph* is extracted in *tempspaces*, a local attribute of *IndentOr-NullNonEmpty*, and then immediately assigned to *indent_or_null.ind*. This value is transmitted by synthesis to *paragraph.ind* in *SubschemaGenPar*, and finally stored in the first component of the current environment's *Binding* in *schema_par*. The

current *Binding* is appended at the head of the list *schema_par\$1.env*, and transmitted through inheritance to the rest of the paragraphs by *schema_par\$2.env*.

The local attribute *initspaces* is assigned the value of the indentation of the first paragraph in the schema. This value is obtained by calling an auxiliary function *indent_lookup*, which receives the environment as an actual parameter and returns the integer-valued indentation level of the first *Binding* in the environment. This auxiliary function is defined as follows:

```
INT indent_lookup(ENV env){
  with (env) (
    NullEnv: 0,
    EnvConcat(b1, e1): with (e1)(
      NullEnv: with (b1)
        (Binding(i,*,*,*,*,*,*,*,*,*,*,*,*,*) : i),
      default: indent_lookup(e1) )) } ;
```

If the difference between *tempspaces*, the value of the current indentation, and *initspaces*, the value of the initial indentation, is not a multiple of *K*, or if the value of the current indentation is less than the value of the initial indentation, a violation of <<P1>> has occurred, and *error* receives the value *ErrP1*. If no error occurs at this point, the inherited environment is inspected once more. The second component *s* of the previous *Binding* *b* contains the name of the previous paragraph. If the previous paragraph is a module, its name starts with an ampersand ('&'), and its indentation *i* must be greater or equal than the current indentation level stored in *tempspaces*. The difference between *tempspaces* and *i* must also be a multiple of *K*. If neither condition holds, *error* receives the value *ErrP3*.

Finally, if none of the previous two violations have occurred, the next semantic

restriction is checked. In this case, if the previous paragraph is a genus paragraph, its name s does not start with an ampersand, and its indentation level i should be greater or equal than $tempspaces$, the indentation value of the current paragraph. If this condition is violated, $error$ receives the value $ErrP2$. Otherwise, $error$ receives the value $NoErr$.

$Error$ also receives the value $ErrP1$, in $IndentOrNullEmpty$. This condition may occur if the indentation of the current paragraph is null.

Example 2:

$\langle\langle P_4 \rangle\rangle$ *Module names must be distinct.*

The module names are defined by the production rules:

```

modparagraph:
  ModParNull()
  | ModParNonNull(modname TBAR interp TPERIOD) ;
modname:
  ModNameNull()
  | ModNameNonNull(MNAME) ;

```

A synthesized attribute ids corresponding to the domain of strings can be associated with the nonterminals $modname$, $modparagraph$, and $paragraph$. The inherited attribute env corresponding to the domain of ENV 's can be associated with the nonterminals $modparagraph$ and $modname$. The constraints can be specified by the following equations:

```

modparagraph:
  ModParNull {
    modparagraph.ids = ""; }
  | ModParNonNull {
    modparagraph.ids = modname.ids;

```



```

    modname.env      = modparagraph.env; } ;
modname:
  ModNameNull {
    modname.ids      = ""; }
| ModNameNonNull {
  local ERR error;
  local BINDING b;
  b      = gname_lookup(MNAME, modname.env);
  error = with(b)(Binding(*,s,*,*,*,*,*,*,*,*,*,*,*,*): s=="?"?
              NoErr(): ErrP4());
  modname.ids = with(b) (Binding(*,s,*,*,*,*,*,*,*,*,*,*,*,*):
                        s=="?"?MNAME:"?"); } ;

```

The lexical value of the module name *MNAME* is assigned, in *ModNameNon-Null*, to the attribute *ids*. This value is then synthesized by *modparagraph*, next by *paragraph*, and eventually recorded in the current *Binding* in *schema-par*.

The auxiliary function *gname_lookup* returns the first *Binding* in the environment such that its corresponding paragraph name, stored as a string in the second component *s*, is equal to the identifier provided as first argument to the function. If no such *Binding* exists, the function returns a *Binding* whose *s* name is equal to the string “?” - - a non-valid genus or module paragraph name. In this latter condition, the local attribute *error* is assigned the null operator *NoErr*. In the former condition, a *Binding* of name *MNAME* already exists, hence *error* is assigned the operator *ErrP4*.

Example 3:

<<P5>> *Genus names must be distinct.*

<<P6>> *No genus names should be “INTERP” or “INTERPRETATION” or end in “_VAL”.*

A genus paragraph name is defined by the production rules:

```

genparagraph:
  GenParNull()
  | PePar(peparagraph)
  | CePar(ceparagraph)
  | AVaPar(avapagraph)
  | FPar(fparagraph)
  | TPar(tparagraph) ;
peparagraph:
  PeParNull()
  | PeNode(gname opindices PE_TYPE_DECL
           op_iss op_dom_stat TBAR interp TPERIOD) ;
ceparagraph:
  CeParNull()
  | CeNode(gname opindices LPAREN calls RPAREN CE_TYPE_DECL
           op_iss op_dom_stat TBAR interp TPERIOD) ;
avapagraph:
  AVaParNull()
  | AVaNode(gname opindices LPAREN calls RPAREN A_VA_TYPE_DECL
            op_iss op_dom_stat op_range_stat TBAR interp TPERIOD) ;
fparagraph:
  FParNull()
  | FNode(gname opindices LPAREN calls RPAREN F_TYPE_DECL
           op_iss op_dom_stat SEMICOLON modfunexpr TBAR interp TPERIOD) ;
tparagraph:
  TParNull()
  | TNode(gname opindices LPAREN calls RPAREN F_TYPE_DECL
           op_iss op_dom_stat SEMICOLON modtstexpr TBAR interp TPERIOD) ;
gname:
  GenNameNull()
  | GenNameNonNull(GNAME) ;

```

The synthesized attribute *ids* corresponding to the domain of strings can be associated with the nonterminals *gname*, *tparagraph*, *fparagraph*, *avapagraph*, *ceparagraph*, *peparagraph*, *genparagraph*, and *paragraph*. The inherited attribute *env* corresponding to the domain of *ENV*'s can be associated with the nonterminals *genparagraph*, *peparagraph*, *ceparagraph*, *avapagraph*, *fparagraph*, *tparagraph*, and *gname*. The constraints can be specified by the following equations:

```

genparagraph:
  GenParNull {
    genparagraph.ids = "";
    genparagraph.typ = '?'; }
  | PePar {
    genparagraph.ids = peparagraph.ids;
    genparagraph.typ = peparagraph.typ;
    peparagraph.env = genparagraph.env; }
  | CePar {
    genparagraph.ids = ceparagraph.ids;
    genparagraph.typ = ceparagraph.typ;
    ceparagraph.env = genparagraph.env; }
  | AVaPar {
    genparagraph.ids = avaparagraph.ids;
    genparagraph.typ = avaparagraph.typ;
    avaparagraph.env = genparagraph.env; }
  | FPar {
    genparagraph.ids = fparagraph.ids;
    genparagraph.typ = fparagraph.typ;
    fparagraph.env = genparagraph.env; }
  | TPar {
    genparagraph.ids = tparagraph.ids;
    genparagraph.typ = tparagraph.typ;
    tparagraph.env = genparagraph.env; } ;
peparagraph:
  PeParNull {
    peparagraph.ids = "";
    peparagraph.typ = '?'; }
  | PeNode {
    gname.env = peparagraph.env;
    peparagraph.ids = gname.ids;
    peparagraph.typ = 'p';
    calls.typ = 'p'; } ;
ceparagraph:
  CeParNull {
    ceparagraph.ids = "";
    ceparagraph.typ = '?'; }
  | CeNode {
    gname.env = ceparagraph.env;
    ceparagraph.ids = gname.ids;
    ceparagraph.typ = 'c';
    calls.typ = 'c'; } ;
avaparagraph:

```

```

    AVaParNull {
    avaparagraph.ids = "";
    avaparagraph.typ = '?'; }
  | AVaNode {
    gname.env       = avaparagraph.env;
    avaparagraph.ids = gname.ids;
    avaparagraph.typ = (STRINGindex(A_VA_TYPE_DECL,'v')==0?'a':'v');
    calls.typ       = (STRINGindex(A_VA_TYPE_DECL,'v')==0?'a':'v'); } ;
fparagraph:
  FParNull {
  fparagraph.ids   = "";
  fparagraph.typ   = '?'; }
  | FNode {
  gname.env        = fparagraph.env;
  fparagraph.ids   = gname.ids;
  fparagraph.typ   = 'f';
  calls.typ        = 'f'; } ;
tparagraph:
  FParNull {
  tparagraph.ids   = "";
  tparagraph.typ   = '?'; }
  | FNode {
  gname.env        = tparagraph.env;
  tparagraph.ids   = gname.ids;
  tparagraph.typ   = 't';
  calls.typ        = 't'; } ;
gname:
  GenNameNull {
  gname.ids        = ""; }
  | GenNameNonNull {
  local ERR error;
  local BINDING b;
  b      = gname_lookup(GNAME, gname.env);
  error = with(b)(Binding(*,s,*,*,*,*,*,*,*,*,*,*,*): s=="?"?
    (GNAME=="INTERP"||GNAME=="INTERPRETATION"||
    ((STRlen(GNAME)>4)&&(GNAME[STRlen(GNAME)-3:]=="_VAL"))
    ?ErrP6():NoErr()):
    ErrP5());
  gname.ids = with(b)(
    Binding(*,s,*,*,*,*,*,*,*,*,*,*,*):s=="?"?GNAME:"?"); } ;

```

The workings of these equations are similar to those of the previous example.

The attribute *ids* is synthesized by each type of genus paragraph, after *gname.ids* has been assigned the name of the current genus, *GNAME*.

If *GNAME* matches the name of the *Binding* returned by the function *gname-lookup*, the local attribute *error* is assigned the operator *ErrP5*. Otherwise, *GNAME* is matched against all of the three forbidden strings. If a match occurs, *error* is assigned the operator *ErrP6*. If none of the above occurs, *error* is assigned the null operator *NoErr*.

3.5.2 Equations for Generic Calling Sequence Sublanguage

Example 4:

<<P1.1>> *Closedness and monotonicity of modular structure require that G must be defined in a prior genus paragraph.*

<<P1.2>> *Moreover, if GNAME is of type compound entity or attribute (including variable attribute), then G may not be of type attribute, variable attribute, function, or test.*

The calling sequence components are defined by the production rules:

```
calls:
    CallComps(components) ;
components:
    CompSingle(component)
    | CompPair(component COMMA components) ;
component:
    CompSimple(simplecomp)
    | CompGeneral(generalcomp) ;
simplecomp:
    SimpleCompNull()
```

```

    | SimpleComp(GNAME) ;
generalcomp:
    GeneralCompNull()
    | GeneralComp(GNAME indexcells) ;

```

The attribute *typ* corresponding to the domain of characters can be synthesized for the nonterminals *paragraph*, *genparagraph*, *peparagraph*, *ceparagraph*, *avapagraph*, *fparagraph*, and *tparagraph*. The same attribute can be inherited by the nonterminals *calls*, *components*, *component*, *simplecomp*, and *generalcomp*. The inherited attribute *env* corresponding to the domain of *ENV*'s can be associated with the nonterminals *calls*, *components*, *component*, *simplecomp*, and *generalcomp*.

The constraints can be specified by the following equations:

```

calls:
    CallComps {
        components.env = calls.env;
        components.typ = calls.typ; } ;
components:
    CompSingle {
        component.env = components.env;
        simplecomp.typ = components.typ; }
    | CompPair {
        component.env = components.env;
        component$2.env = components.env;
        component.typ = components.typ;
        component$2.typ = components.typ; } ;
component:
    CompSimple {
        simplecomp.env = component.env;
        simplecomp.typ = component.typ; }
    | CompGeneral {
        generalcomp.env = component.env;
        generalcomp.typ = component.typ; } ;
simplecomp:
    SimpleCompNull { }
    | SimpleComp {
        local ERR error;

```

```

local BINDING b;
b      = gname_lookup(GNAME, simplecomp.env);
error  = with(b)(Binding(*,s,*,*,k,*,*,*,*,*,*,*,*,*):
          s==""? ErrP1_1(): (
            (simplecomp.typ=='c' || simplecomp.typ=='a' ||
             simplecomp.typ=='v') && (k=='a' || k=='v' || k=='f' ||
             k=='t')) ? ErrP1_2(): NoErr()); } ;

generalcomp:
  GeneralCompNull { }
| GeneralComp {
  local ERR error;
  local BINDING b;
  b      = gname_lookup(GNAME, generalcomp.env);
  error  = with(b)(Binding(*,s,*,*,k,*,*,*,*,*,*,*,*,*):
                s==""? ErrP1_1(): (
                  (generalcomp.typ=='c' || generalcomp.typ=='a' ||
                   generalcomp.typ=='v') && (k=='a' || k=='v' || k=='f' ||
                   k=='t')) ? ErrP1_2(): NoErr()); } ;

```

3.5.3 Equations for Range Statement Sublanguage

Example 5:

<<P2.4>> $lo \leq hi$ is necessary to avoid an empty subrange.

The range options are defined by the production rules:

```

single_string_range:
  SingleStringRangeNull()
| SingleStringRangeOption1(string_options)
| SingleStringRangeOption2(QUOTED_STRING ltle string_options)
| SingleStringRangeOption3(string_options ltle QUOTED_STRING)
| SingleStringRangeOption4(QUOTED_STRING ltle string_options
  ltle QUOTED_STRING) ;

single_integer_range:
  SingleIntegerRangeNull()
| SingleIntegerRangeOption1(integer_options)
| SingleIntegerRangeOption2(integer LT_LE_INTEGER optsign)
| SingleIntegerRangeOption3(integer_options ltle integer)
| SingleIntegerRangeOption4(integer LT_LE_INTEGER optsign

```

```

    ltle integer) ;
single_real_range:
    SingleRangeRangeNull()
  | SingleRangeRangeOption1(real_options)
  | SingleRangeRangeOption2(real_int ltle REALLY optsign)
  | SingleRangeRangeOption3(real_options ltle real_int)
  | SingleRangeRangeOption4(real_int ltle REALLY optsign
    ltle real_int) ;

```

The synthesized attribute *val* corresponding to the domain of reals can be associated with the nonterminals *integer* and *real*, and the constraint can be specified by the following equations:

```

single_string_range:
    SingleStringRangeNull{ }
  | SingleStringRangeOption1{ }
  | SingleStringRangeOption2{ }
  | SingleStringRangeOption3{ }
  | SingleStringRangeOption4{
    local ERR error;
    error = QUOTED_STRING$1>QUOTED_STRING$2? ErrP2_4():NoErr(); } ;
single_integer_range:
    SingleIntegerRangeOption4{
    local ERR error;
    error = integer$1.val>integer$2.val?
      ({UnqualRangeBodyInteger.in_sp_decl,
        QualRangeBodyInteger.in_sp_decl,
        QualSymParTypeInteger.in_sp_decl,
        UnqualSymParTypeInteger.in_sp_decl} ?
        ErrP4_34():ErrP2_4()) : NoErr(); } ;
single_real_range:
    SingleRealRangeOption4{
    local ERR error;
    error = real_int$1.val>real_int$2.val?
      ({UnqualRangeBodyReal.in_sp_decl,
        QualRangeBodyReal.in_sp_decl,
        QualSymParTypeReal.in_sp_decl,
        UnqualSymParTypeReal.in_sp_decl} ?
        ErrP4_34:ErrP2_4()) : NoErr(); } ;

```


3.5.4 Equations for Index Set Statement Sublanguage

Example 6:

<<P3.1>> Each kind of genus has its own variety of index set statement that may not be used for any other kind of genus.

The index set statement options are defined by the production rules:

```
op_iss:
  OpIssNull()
| OpUnindexedIss(unindexed)
| OpSelfIss(self_iss)
| OpExternalIss(external_iss) ;
```

The inherited attribute *sym* corresponding to the domain of strings can be associated with the nonterminal *op_iss*, and the constraints can be specified by the following equations:

```
op_iss:
  OpIssNull{ }
| OpUnindexedIss{
  local ERR error;
  error = (STRlen(op_iss.sym)==0&&STRlen(op_iss.git)==0
    ?NoErr():ErrP3_1()); }
| OpSelfIss{
  local ERR error;
  error = (STRlen(op_iss.sym)==0?ErrP3_1():NoErr()); }
| OpExternalIss{
  local ERR error;
  error = (STRlen(op_iss.sym)==0?NoErr():ErrP3_1()); } ;
```

3.5.5 Equations for Generic Rule Statement Sublanguage

Example 7:

<<P4.2>> The generic rule of a genus of type /f/ must be a numeric-valued expression.

A paragraph corresponding to a function element was defined by the production rule *fparagraph* in Example 3.

The synthesized attribute *is_num* corresponding to the domain of booleans can be associated with the nonterminal *modfunexpr*, and the constraint can be specified by the following equations:

```
fparagraph:  
  FParNull { }  
| FNode {  
  local ERR error;  
  error = (modfunexpr.is_num ? NoErr() : ErrP4_2()); } ;
```

3.5.6 Equations for Interpretation Sublanguage

Example 8:

<<P5.1>> Defined key phrases must all be distinct.

The full paragraph interpretation can be maintained as a (possibly empty) set of interpretation lines (or logical units). Each interpretation line is either a defined key phrase, a referenced key phrase, or a non-key phrase. An interpretation line is defined by the production rules:

```
interp_line:  
  InterNilKeyPhrase()  
| InterpNonKeyPhrase(NON_KEY_PHRASE)  
| InterpDefKeyPhrase(DEF_KEY_PHRASE)  
| InterpRefKeyPhrase(REF_KEY_PHRASE) ;
```

The inherited attribute *env* corresponding to the domain of ENV can be associated with the nonterminal *interp_line*, and the constraint can be specified by the following equations:

```
interp_line:
  InterpNonKeyPhrase,InterpNilKeyPhrase{
    interp_line.phrase = ""; }
| InterpDefKeyPhrase{
  local ERR error;
  error = def_key_phrase_intro(DEF_KEY_PHRASE,interp_line.env) ?
    ErrP5_1() : NoErr();
  interp_line.phrase = DEF_KEY_PHRASE; }
| InterpRefKeyPhrase{
  interp_line.phrase = ""; } ;
```

3.5.7 Equations for Elemental Detail Tables

The following production rules describe the modifications that have been made to the original SML context-free syntax to incorporate the Elemental Detail Tables and the Table Content Rules.

```
schemafile:
  SchemaFileList(schema_par EOF opt_element_detail) ;
opt_element_detail:
  OptElementDetailNil()
| OptElementDetailTable(ed_table_list) ;
ed_table_list:
  EDTableListSingle(ed_table TPERIOD)
| EDTableListCompound(ed_table TPERIOD ed_table_list) ;
ed_table:
  EDTableEmpty(table_struct)
| EDTableFull(table_struct line_list) ;
table_struct:
  TableStructOption(QUOTED_STRING COMMA QUOTED_STRING
  COMMA AN_INTEGER) ;
line_list:
  LineListOption1(data_line)
| LineListOption2(data_line line_list) ;
```

```

data_line:
    DataLineOption1(QUOTE_BAR_BAR_QUOTE COMMA data_list)
  | DataLineOption2(data_list COMMA QUOTE_BAR_BAR_QUOTE)
  | DataLineOption3(data_list COMMA QUOTE_BAR_BAR_QUOTE
    COMMA data_list) ;
data_list:
    DataListOption1(data)
  | DataListOption2(data_list COMMA data) ;
data:
    DataOptionNull()
  | DataOption1(QUOTED_STRING)
  | DataOption2(A_REAL)
  | DataOption3(AN_INTEGER) ;

```

The structure above describes elemental detail tables appended after the Schema section. To accommodate models with no tables, the nonterminal *opt_elemental_detail* has the optional nil operator *OptElementalDetailNil*.

Example 9:

<<TCR.A>> Each stub must consist of a non-empty, finite collection of tuples composed of valid identifiers.

In SML, an identifier is “valid” if it is defined in the Elemental Detail Table for the associated self-indexed genus. A data item is defined by one of the production rules:

```

data:
    DataOptionNull()
  | DataOption1(QUOTED_STRING)
  | DataOption2(A_REAL)
  | DataOption3(AN_INTEGER) ;

```

The inherited attribute *in_stub* corresponding to the domain of booleans can be associated with the nonterminal *data*, and the constraint can be specified by the following equations:

```

data:
  DataOptionNull{
    local ERR error;
    error      = (data.in_stub==false) ? NoErr(): ErrTCR_A();
    data.stub = ""; }
| DataOption1{
  local ERR error;
  error = (data.in_stub==false) ? NoErr():
    (data.tabtype!=1) ? NoErr():
    ((data.domtype==0)&&(STRlen(QUOTED_STRING)==2))?ErrTCR_A():
    NoErr() ;
  data.stub = ((data.in_stub==true)&&(data.tabtype==1)) ?
    QUOTED_STRING[2:STRlen(QUOTED_STRING)-1] : ""; }
| DataOption2{
  local ERR error;
  data.stub = "";
  error      = (data.in_stub==true)?ErrTCR_A():NoErr(); }
| DataOption3{
  local ERR error;
  error = data.in_stub==false? NoErr():
    (data.domtype<3)?ErrTCR_A():
    ((data.domtype==4)&&(STRtoINT(AN_INTEGER)<0))?ErrTCR_A():
    ((data.domtype==5)&&(STRtoINT(AN_INTEGER)>0))?ErrTCR_A():
    NoErr();
  data.stub = ((data.in_stub==true)&&(data.tabtype==1)) ?
    AN_INTEGER[2:STRlen(AN_INTEGER)-1] : ""; } ;

```

3.6 Importance of the Attribute Grammar Approach

It is surprising that, at the time of this writing, very little has been published in the area of semantic formalization of modeling languages and modeling environments. Although formal definitions of programming languages and programming environments exist in the literature, we believe this is the first attempt to formally specify the semantics of a modeling language via attribute grammars. Moreover, even though the examples of attribute grammar specification that were shown in

the previous section dealt exclusively with the static semantics of SML, the fact that this language is so large and complex, and that it contains so many of the desirable design features of the new generation of modeling languages, suggest that similar approaches may be applicable to other modeling languages of equal or less complexity.

For practical purposes, the approach is restricted to languages that have a complete syntactic and context-dependent semantics specification, and where all attribute instances can be defined without circularities. Algorithms for detecting if an attribute grammar is circular do exist, e.g., [Knut68]. Noncircularity can be defined as a requirement of acyclicity on the dependency graph for every possible derivation tree. A dependency graph is a directed graph which includes an edge from 'a' to 'b' if 'a' is used to determine the value of 'b', and a vertex 'c' for every attribute instance 'c'.

It is important to note that, while there are similarities, modeling languages are different from programming languages. While programming languages can be viewed as tools for writing algorithms (“programming”), mathematical modeling languages are more concerned with describing the structural relationships between the facts of a problem. Modeling leaves the programming tasks (like model optimization, evaluation, and solving) to the “solvers” which can be invoked independently in the modeling environment. Hence, common abstractions, structures, rules, and methodologies adopted by programming languages are not necessarily relevant to modeling languages. For example, the context-dependent restrictions

of SML of existential nature are not found in typical procedural programming languages.

Two important benefits of the attribute grammar methodology are discussed below: its ease of modification and its non-procedural approach. Of course, these benefits can only be derived when there is a high degree of completeness in the specification of context sensitivities in the modeling language.

3.6.1 Easy Modifiability

Ease of maintenance and modification, also called “extendibility” [Mey88], plays a key role in controlling software costs and increasing productivity [Boeh87]. For mathematical programming modeling languages, and environments, the situation is similar: an easily modifiable formalization technique saves programming time and expense. Hence, to improve productivity, a formal specification methodology should be easily amenable to change when faced with requirement modifications. The ultimate objective is the practical elimination of the traditional software maintenance: when changes are made to any of the language’s, or environment’s, requirements, only the formal specifications need to be changed.

One approach to achieving easy modifiability is via the use of automated tools which take high-level input specifications and produce programs as output. Program generation tools, like Lex [Lesk75] (a regular expression based lexical analyzer generator) and YACC [John78] (an LALR(1) parser generator), for example, have been extensively used to minimize the cost impact of software changes. However,

because SML, like other languages for model description, has numerous context-sensitive aspects, table-driven generators like Lex and YACC cannot be used to generate a specification for SML that will support all of its context-sensitive aspects and, at the same, possess a definition style that can be easily modified. Of course, the static semantics of SML may be written in *C* and included as “action routines” in an environment generated from YACC. (In fact, this has been done in the prototype structured modeling environment FW/SM [Geof90]). However, this approach is a hand tailored method of semantic evaluation and is not easy to maintain and modify. Indeed, in this approach only syntax (without including semantics) is described in a modular fashion, and all implementation details are explicitly shown.

The advantages of the attribute grammar approach over toolkit, ad-hoc routines, and method-based approaches for ease of maintenance make it a preferred specification technique. Moreover, this kind of definition is sufficiently clear to facilitate the understanding of language semantics. And because the semantic equations in the attribute grammar can be easily tested and validated, this definition style can lead to complete, consistent, and correct specifications.

Our personal experience in building an attribute grammar-based syntax-directed editor for SML lends support to the arguments of easy modifiability. This ease of maintenance is a byproduct of the modularity of the specifications of syntax and semantics, and also of declarativeness which hides all implementation details, as discussed in Section 3.6.2. We have found that the construction of a working pro-

prototype editor has not been too time-consuming. Furthermore, the prototype has been able to evolve quickly in response to evolution in SML.

3.6.2 Declarative Approach

An important characteristic of attribute grammar specifications is that they are declarative without side effects, as opposed to procedural; they are not programs. Unlike procedural programming, there are no explicit loops, no control structures, and attributes cannot be incremented or reassigned. Attribute instances are defined by the semantic equations. The order in which attribute instances are evaluated does not depend on the order in which the semantic equations are listed; they only depend on the evaluation strategy employed. The concern is with “what” needs to be done, rather than with “how” it is done. This fact concedes a high degree of abstraction to the description of context-dependent semantics in modeling languages.

The declarative nature of attribute grammar specifications has numerous implications, extending from the areas of attribute evaluation to applications. In terms of the attribution process, for example, the declarative approach makes it suitable for exploiting parallelism in advanced machines. In terms of applications, for instance, the declarative approach allows for the generation of syntax-directed editors, a topic which is discussed in Chapter 6.

The other important aspect of the declarative nature of attribute grammar specifications is that the functions described by the semantic equations do not

have side effects. Side effects, for example, are a typical objective of syntax-directed translations, where an external procedure is specifically called during the translation process to create the side effect.

Syntax-directed editing is one case of an application where side effects in semantic equations are undesirable. In a syntax-directed editor for SML, for example, an editing change in the source model may result in a fragment of the parse tree for the model being deleted, and thus leave the attributes of the tree with inconsistent values. As long as there no side effects, the attribute values for the new tree can be updated incrementally, and thus the evaluator can perform less work than would be required otherwise [Aho86].

3.7 Difficulties and Lessons

A great deal of experimental work was required to implement many of the context-dependent restrictions in SML. Only after an individual and detailed inspection of each semantic restriction was it possible to show that all of SML's context-dependencies could be formalized via attribute grammar equations. Numerous functions used in the semantic equations had to be newly defined to propagate information across the semantic tree. Occasionally, functions that were used in certain semantic equations had to be defined procedurally in *C*, but they were referenced as if they were primitive within the semantic equation. It would be desirable if the attribute grammar evaluator provided many more primitive functions, instead of having to build these as foreign functions.

An important difficulty in dealing with Table Content Rules relates to representing completeness and conformity. The appearance of a tuple in a certain table is typically controlled by the satisfaction of both integrity constraints and existential constraints. To test conformity to an index set statement, for example, a complete understanding of the relational algebraic expressions involved is required. The interpretation of relational expressions can only be achieved by executing an algorithm designed to interpret the index set statement when it is viewed as a query on the tables. These algorithms fall more properly into the area of run-time evaluation, than into the area of static semantics. Strategies of query optimization, such as those used in high level database query languages (e.g., [Ullm82]) are relevant to this task.

Another difficulty, not normally found with programming languages, relates to preserving the original structure of the formal and informal parts of an SML model, in both the concrete and the abstract syntax representation. Whereas white space (i.e., blanks and newlines) and comments normally can be discarded in programming languages because they have no significant structure, in SML they need to be preserved because they add clarity to a model, or because they are an integral part of the (informal) description of the model, and are subject to some minimal structure (expressed via the language of keyphrases). Hence, the concrete and abstract syntax representations need to carry, in their syntax trees, all the white space that is contained in both the formal and informal parts of a model. This demand adds much complexity to the modeling environment designer's task

of representing a generic model in both concrete and abstract syntax terms. This complexity, of course, is a consequence of having to reproduce, for any modeling language, the model in the output display in a form that is identical to its original form. However, it also slows down the performance of the attribution process, since the abstract syntax tree has to be almost identical in size and decoration to the concrete syntax tree.

CHAPTER 4

Type Inferencing in Mathematical Modeling Languages

Mathematical modeling languages, in general, have abstractions to permit the declaration of typed elements. To illustrate, consider GAMS, for example, where each identifier (called symbol) is declared to be one of six data types, which are SET, PARAMETER, VARIABLE, EQUATION, MODEL, and ACRONYM. After the initial specification of the data type of an identifier, the values (called labels) assigned to the identifier have to be of the proper type. Or consider also the Generic Rule Sublanguage of SML, which allows for the declaration of classes of typed elements that are evaluated via generic function and test rules. Function rules must return numeric values, while test rules must return logical values. A goal of an “integrated” - - in the sense examined in [Geof89d] - - modeling system that supports model building ought to be that both model classes and model instances always be well-typed during the entire model development process.

To maintain strong-typing, languages for model description usually make default type assumptions when the explicit type declaration for an element is missing. Note that this type information is sometimes intentionally absent from the schema to maintain clarity in the model’s description. However, it would be possible to endow a modeling system with an element type inferencing mechanism that in-

serts the inferred type in the schema while freeing the modeling language from (1) making any default assumptions, and (2) from requiring explicit type declarations. Furthermore, the principles which would be used in an inference mechanism for SML, for example, would be applicable to other mathematical modeling languages that have a structure such that those same conclusions, which could be reached about a model from information that is explicitly declared, could be implicitly reached from an analysis of references.

This chapter discusses the design of such an alternative element type inferring mechanism for SML. This design, which is based on attribute grammars, tests capabilities of attribute-grammar-based definitions which go beyond those of traditional context-sensitive semantic descriptions. As an additional motivation, it suggests a way of handling incompletely specified models, namely, that of inferring incomplete or missing details.

The chapter is organized as follows. To begin, some of the areas where inferring is appropriate in SML are described in Section 4.1. Then Section 4.2 provides attribute grammar designs (including changes to the syntax of SML, and corresponding attribute equations) that can eliminate, through inferring, the explicit type declaration for symbolic parameters, and explicit domain statements. Finally, a discussion follows in Section 4.3.

4.1 Areas Amenable to Inferencing in SML

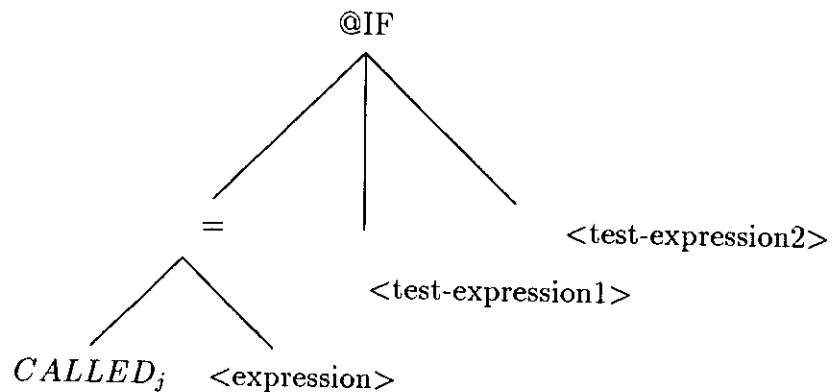
In SML, there are at least three separate areas where explicit element type declarations are necessary, or are assumed by default, and hence which could be simplified by inferencing: (1) the area of simple variables in numeric-valued and logical-valued generic rules, (2) the area of domain statements, and (3) the area of symbolic parameter type declarations. In addition, the area of calling sequences in genus paragraphs could also benefit from an inferencing approach. The following subsections formulate the problems which would be solved in each area if inferencing were provided.

4.1.1 Simple Variables

Consider the following (incomplete) fragment of a structured model:

```
GENUS(CALLEDj) /t/ ;  
@IF(CALLEDj = <expression>, <test-expression1>, <test-expression2>)
```

with a generic rule (abstract) syntax-tree:



At this point in the model development process, the selection (highlight) could be either at $\langle expression \rangle$, $\langle test-expression1 \rangle$, or $\langle test-expression2 \rangle$. The modeler has not yet refined any of these expressions, and perhaps might not even have introduced the corresponding genus paragraph for the simple variable $CALLED_j$ (assuming we can suspend the associated semantic errors). Nevertheless, we would like the modeling system to infer the type of each (numeric- or test-valued) expression, and warn the developer with an error message on the screen as soon as any expression has a type mismatch.

For example, when the simple variable $CALLED_j$ is of type primitive entity or compound entity, it cannot participate in this context. When the simple variable $CALLED_j$ is of type attribute (or variable attribute) with real or integer range, $\langle expression \rangle$ must evaluate to a numeric value. But, when $CALLED_j$ is of type attribute (or variable attribute) with a string range, $\langle expression \rangle$ must then evaluate to a string value. All these type inferences should be made by the modeling system on the fly, as soon as $\langle expression \rangle$ is refined. Similarly, the system should type check any occurrences of $CALLED_j$ in $\langle test-expression1 \rangle$ and $\langle test-expression2 \rangle$.

4.1.2 Type Declaration for Symbolic Parameters

The data types of the symbolic parameters mentioned in a generic rule are usually provided by explicit type declarations, at the end of the generic rule. If a symbolic parameter type declaration is omitted, a default type is assumed.

It would be desirable, however, to eliminate the explicit symbolic parameter type declaration, as symbolic parameter types can be normally inferred from the contexts where each symbolic parameter is used. For example, consider the following (incomplete) fragment of a structured model:

```
T:ZIP(CITYj) /t/ ;
@IF(j = 5, %ZIP > 9000, %ZIP = "#TRUE")
```

In this fragment, the symbolic parameter *%ZIP* is not used consistently. In *<test-expression1>*, *%ZIP* is compared against a numeric value, while in *<test-expression2>*, *%ZIP* is compared against a string value. It would be desirable that a modeling system detect this kind of type violation on the fly, as soon as it occurs.

4.1.3 Domain Statements

The data type of the identifiers used for the elements of a self-indexed genus is specified by a domain statement. If the statement is omitted, a default type is assumed. Naturally, the modeling system takes care of enforcing that all element identifiers be consistent with the declared, or assumed, domain statement. It would be desirable, however, to completely eliminate the domain statement from the SML syntax, as it is not pertinent to the core concepts of Structured Modeling.

The domain statement may be deduced by inspecting the identifiers for all the elements stored in Elemental Detail Tables, and then subsequently propagating this type information to the relevant sections of the Schema. If the elements are inconsistent, a distinguished error type is propagated. Hence, the modeling system could be flexible enough to check, at the same time, that all the identifiers in

Elemental Detail Tables are *valid* (in the SML sense), and that they are consistent with Schema declarations. The modeler would then be immediately notified on the screen of any element type violation.

4.1.4 Calling Sequences

The calling sequence of a compound entity, attribute, function, or test genus lists all the definitional dependencies of the current genus on genera that appear previously in the Schema. Some restrictions exist, like the restriction that a compound entity genus may not call an attribute, function, or test genus. Normally, it is the responsibility of the modeler to include all the definitional dependencies in the calling sequence of each genus. SML has specific Schema Properties designed to flag errors when explicit definitional dependencies are omitted.

It is possible, however, to have an inferencing system that will deduce mandatory parts of the calling sequences of a function or test genus from the actual use of the definitional dependencies in generic rules. For instance, the indices used in index supporting functions could provide a clue about the genera that define those indices. This inferencing approach could provide economy in the writing of calling sequences. Additionally, it could provide a useful service to the modeler, who could check if those calling sequences that are inferred match those that were intended when the model was formulated. Even though it would not be desirable to provide this service as a replacement for the explicit declaration of the calling sequences, this service would be very useful to complement other functions provided

in a modeling environment.

4.2 Type Inferencing in SML via Attribute Grammars

The previous section provided examples of areas where type information in some of SML's Schema paragraphs might not be available (e.g., because it is not explicitly declared for clarity considerations), and yet where immediate incremental type inference is desired. Indeed, it is desirable that the modeling system annotate the screen with error messages, as soon as a structural piece is incorrectly refined; that is the least expensive moment to do it.

Support for this kind of incremental type checking is difficult in the attribute grammar approach. In the classical attribute grammar style of semantic evaluation, type declarations are collected via synthesized attributes which are attached to the nonterminals closest to the declarations of the identifiers. The lexical value of a declared identifier is then propagated to the head of the declaring production where it is appended to a global symbol table. This modified global symbol table is later broadcast down the syntax-tree, thus making the declared name visible to other productions. This method requires that all identifiers be declared first, before the type checking step is begun.

To derive type information when an SML model Schema is incomplete (i.e., when not all typed paragraph constructs are explicitly declared), type inferencing becomes necessary. The following subsections provide attribute grammar schemes which can perform the needed inferencing of symbolic parameters types, and of

types implied by domain statements.

4.2.1 Inference of Types of Symbolic Parameters

In this approach we do not deal with symbolic parameters that are constrained to some *range*. In principle, however, this approach may be extended to cover a range by attaching lower and upper bounds to each inferred data type, and then carrying the bounds information together with the type name in a structured-valued attribute.

The first step is to eliminate the explicit symbolic parameter type declaration from the syntax in the Generic Rule Sublanguage. Indeed, after removing the *opt_sp_type_decl* operand from the operators *MFunExpr* and *MTstExpr*, function and test expressions are defined by the production rules:

```
modfunexpr:
    ModFunNull()
  | MFunExpr(funexpr) ;
modtstexpr:
    ModTstNull()
  | MTstExpr(testexpr) ;
```

Symbolic parameters can now be defined from the expansion of the nonterminals *funexpr* and *testexpr* via the path: $\{funexpr, testexpr\} \rightarrow expr \rightarrow term \rightarrow power \rightarrow factor \rightarrow variable \rightarrow sympar$. The explicit abstract syntax is as follows:

```
funexpr:
    FuncExprNonNull(expr) ;
testexpr:
    TExpNull()
  | LitConst(LITERAL)
  | TestPair(expr relop expr)
  | TestTriple(expr relop expr relop expr)
```

```

| LogIFc(log_index_sup_fun)
| AndPair(AT_AND LPAREN testexpr COMMA testexpr_list RPAREN)
| OrPair(AT_OR LPAREN testexpr COMMA testexpr_list RPAREN)
| NotPair(AT_NOT LPAREN testexpr RPAREN)
| IfTPair(AT_IF LPAREN testexpr COMMA testexpr COMMA testexpr RPAREN)
| TParen(LPAREN testexpr RPAREN)
| TExist(AT_EXIST LPAREN exist_arg COMMA testexpr COMMA testexpr
  RPAREN) ;
testexpr_list:
  TExpList1(testexpr)
  | TExpList2(testexpr_list COMMA testexpr) ;
expr:
  ExprNull()
  | TermSingle(term)
  | TermPair(MINUS_SIGN term)
  | PlusPair(expr PLUS_SIGN term)
  | MinusPair(expr MINUS_SIGN term) ;
term:
  PowerSingle(power)
  | ProdPair(term MULT_SIGN power)
  | QuotPair(term DIVIDE_SIGN power) ;
power:
  FactorSingle(factor)
  | ExpPair(factor EXP_SIGN power) ;
factor:
  Const(constant)
  | Var(variable)
  | ParenExpr(LPAREN expr RPAREN)
  | IfFTPair(AT_IF LPAREN testexpr COMMA expr COMMA expr RPAREN)
  | FExist(AT_EXIST LPAREN exist_arg COMMA expr COMMA expr RPAREN) ;
constant:
  ConstNum1(nninteger)
  | ConstNum2(NN_REAL)
  | ConstNum3(QUOTED_STRING) ;
variable:
  VarNull()
  | Variable1(symvar)
  | Variable2(simplevar)
  | Variable3(builtin_function)
  | Variable4(pp_index)
  | Variable5(functional_dependency)
  | Variable6(arith_index_sup_fun) ;
log_index_sup_fun:

```

```

    LogIndexSupFunctionNonNull(it_log_fun_unit LPAREN testexpr
    RPAREN) ;
arith_index_sup_fun:
    ArithIndexSupFunctionNonNull(it_arith_fun_unit LPAREN expr
    RPAREN) ;
simplevar:
    SimpVarConst(GNAME)
    | SimpVarName(GNAME grindices) ;
builtin_function:
    BuiltinFunctionNonNull( AT_SIGN GNAME expr_pack ) ;
expr_pack:
    ExprPackNonNull( expr_head expr RPAREN ) ;
expr_head:
    ExprHeadNull( LPAREN )
    | ExprHeadNonNull( expr_head expr COMMA ) ;
sympar:
    SymParNull()
    | SymPar1(S_P_STEM)
    | SymPar2(S_P_STEM sym_par_indices) ;

```

Example 1. The test expression

$@IF(j = 5, \%ZIP > 9000, \%ZIP = "\#TRUE"),$

discussed in Section 4.1.2, can be derived via the following paths:

$modtstexpr \rightarrow tstexpr \rightarrow @IF(testexpr_1 , testexpr_2 , testexpr_3) .$

$testexpr_1 \rightarrow expr_1 = expr_2 .$

$expr_1 \rightarrow term \rightarrow power \rightarrow factor \rightarrow variable \rightarrow pp_index \rightarrow j .$

$expr_2 \rightarrow term \rightarrow power \rightarrow factor \rightarrow constant \rightarrow 5 .$

$testexpr_2 \rightarrow expr_3 > expr_4 .$

$expr_3 \rightarrow term \rightarrow power \rightarrow factor \rightarrow variable \rightarrow sympar \rightarrow \%ZIP .$

$expr_4 \rightarrow term \rightarrow power \rightarrow factor \rightarrow constant \rightarrow 9000 .$

$testexpr_3 \rightarrow expr_5 = expr_6 .$

$expr_5 \rightarrow term \rightarrow power \rightarrow factor \rightarrow variable \rightarrow sympar \rightarrow \%ZIP .$

$expr_6 \rightarrow term \rightarrow power \rightarrow factor \rightarrow constant \rightarrow \#TRUE .$

The productions of abstract syntax shown above do not carry type information for the symbolic parameter *sympar*. Thus, the type “NoType” can be initially assigned to *sympar*. After type inferencing, *sympar* will be assigned the legal type “Logical”, “Integer”, “Real”, or “String”, or the error or unknown type “Illegal”. *sympar* may also be given a temporary (intermediate) type, “Number”, which stands for either “Integer”, or “Real”. The domain of symbolic parameter types, *ATYPE*, can now be syntactically defined by the operators:

```
ATYPE:
  NoType()
  | Logical()
  | Integer()
  | Real()
  | Number()
  | String()
  | Illegal() ;
```

The names and types of all the symbolic parameters in the current function expression (*funexpr*) or test expression (*testexpr*) can be collected in a type environment, *TYPENV*, defined by the following equations:

```
list TYPENV;
TYPENV:
  NullSymParTypeEnv()
  | SymParTypePair(TypedMember TYPENV) ;

TypedMember:
  TypedMemberOp(STR ATYPE) ;
```

Thus, the domain *TYPENV* is a list of *TypedMembers*, where each list member is a pair with components (*STR ATYPE*), with first component in the domain of

*STR*ings, and second component in the set of *ATYPE*. The empty list is denoted by *NullSymParTypeEnv*.

At the core of an inferencing scheme is a function *Narrow* that, given two type operands as input, say *t1* and *t2*, returns a narrower type according to the coercion rules of SML. The function *Narrow* is defined as follows:

```
ATYPE Narrow(ATYPE t1, ATYPE t2){
with(t1)(
  NoType: t2,
  Logical: with(t2)(
    NoType: Logical, Logical: Logical, Integer: Illegal,
    Real: Illegal, Number: Illegal, String: Illegal,
    Illegal: Illegal,),
  Integer: with(t2)(
    NoType: Integer, Logical: Illegal, Integer: Integer,
    Real: Real, Number: Integer, String: Illegal,
    Illegal: Illegal,),
  Real: with(t2)(
    NoType: Real, Logical: Illegal, Integer: Real,
    Real: Real, Number: Real, String: Illegal,
    Illegal: Illegal,),
  Number: with(t2)(
    NoType: Number, Logical: Illegal, Integer: Integer,
    Real: Real, Number: Number, String: Illegal,
    Illegal: Illegal,),
  String: with(t2)(
    NoType: String, Logical: Illegal, Integer: Illegal,
    Real: Illegal, Number: Illegal, String: String,
    Illegal: Illegal,),
  Illegal: Illegal ) } ;
```

The merging and sorting of two symbolic parameter type environments, say *e1* and *e2*, can be performed by the recursive function *MergeTypeEnv*, defined as follows:

```
TYPENV MergeTypeEnv(TYPENV e1, TYPENV e2) {
with(e1)(
```



```

NullSymParTypeEnv: e2,
SymParTypePair(TypedMemberOp(m1,t1),tail1): with(e2)(
  NullSymParTypeEnv: e1,
  SymParTypePair(TypedMemberOp(m2,t2),tail2): m1 < m2 ?
    TypedMemberOp(m1,t1)::MergeTypeEnv(tail1,e2): m1 == m2 ?
    TypedMemberOp(m1,Narrow(t1,t2))::MergeTypeEnv(tail1,tail2) :
    TypedMemberOp(m2,t2)::MergeTypeEnv(e1,tail2)) ) } ;

```

Finally, a type environment, say $e1$, can have the type of all of its members narrowed to a different type, say $t2$, via the function *NarrowTypeEnv* defined as follows:

```

TYPENV NarrowTypeEnv(TYPENV e1, ATYPE t2){
with(e1)(
  NullSymParTypeEnv: e1,
  SymParTypePair(TypedMemberOp(m1,t1),tail1):
    TypedMemberOp(m1,Narrow(t1,t2))::NarrowTypeEnv(tail1,t2)) } ;

```

To gather and broadcast type information, the synthesized attributes *SymParTypeEnv*, corresponding to the domain of symbolic parameter type environments *TYPENV*, and *type*, corresponding to the domain of types *ATYPE*, can be associated with the nonterminals *funexpr*, *testexpr*, *testexpr_list*, *expr*, *term*, *power*, *factor*, *constant*, *variable*, *log_index_sup_fun*, *arith_index_sup_fun*, *simplevar*, *builtin_function*, *expr_pack*, *expr_head*, and *sympar*. The first attribute carries the symbolic parameter type bindings that are visible to the current nonterminal, while the second attribute carries the particular inferred type of the nonterminal. The attribute *nam*, corresponding to the domain of *STRings*, can also be associated with the nonterminal *sympar*. This attribute holds the lexical name of the symbolic parameter *sympar*.

The attribute equations which generate the type environment for the complete

paragraph (i.e., *funexpr.SymParTypeEnv* and *testexpr.SymParTypeEnv*) are as follows:

```

funexpr:
  FuncExprNonNull{
    funexpr.SymParTypeEnv    = expr.SymParTypeEnv;
    funexpr.type             = expr.type; } ;
testexpr:
  TExpNull{
    testexpr.SymParTypeEnv    = NullSymParTypeEnv;
    testexpr.type            = NoType; }
  | LitConst{
    testexpr.SymParTypeEnv    = NullSymParTypeEnv;
    testexpr.type            = Logical; }
  | TestPair{
    testexpr.SymParTypeEnv    = MergeTypeEnv(
      NarrowTypeEnv(expr$1.SymParTypeEnv, expr$2.type),
      NarrowTypeEnv(expr$2.SymParTypeEnv, expr$1.type)) ;
    testexpr.type            = Logical; }
  | TestTriple{
    testexpr.SymParTypeEnv    = MergeTypeEnv(
      MergeTypeEnv(NarrowTypeEnv(expr$1.SymParTypeEnv, expr$2.type),
        NarrowTypeEnv(expr$2.SymParTypeEnv, expr$1.type)),
      MergeTypeEnv(NarrowTypeEnv(expr$2.SymParTypeEnv, expr$3.type),
        NarrowTypeEnv(expr$3.SymParTypeEnv, expr$2.type)));
    testexpr.type            = Narrow(
      Narrow(expr$1.type, expr$2.type), Narrow(expr$2.type, expr$3.type)); }
  | LogIFc{
    testexpr.SymParTypeEnv    = log_index_sup_fun.SymParTypeEnv;
    testexpr.type            = log_index_sup_fun.type; }
  | AndPair, OrPair{
    testexpr$1.SymParTypeEnv  = MergeTypeEnv(
      testexpr$2.SymParTypeEnv, testexpr_list.SymParTypeEnv);
    testexpr$1.type          = Logical; }
  | NotPair{
    testexpr$1.SymParTypeEnv  = testexpr$2.SymParTypeEnv;
    testexpr$1.type          = Logical; }
  | IfTPair{
    testexpr$1.SymParTypeEnv  = MergeTypeEnv(
      MergeTypeEnv(testexpr$2.SymParTypeEnv, testexpr$3.SymParTypeEnv),
      testexpr$4.SymParTypeEnv);
    testexpr$1.type          = Logical; }

```

```

| TParen{
  testexpr$1.SymParTypeEnv = testexpr$2.SymParTypeEnv;
  testexpr$1.type          = Logical; }
| TExist{
  testexpr$1.SymParTypeEnv = MergeTypeEnv(
    testexpr$2.SymParTypeEnv, testexpr$3.SymParTypeEnv);
  testexpr$1.type          = Logical; } ;
testexpr_list:
  TExpList1{
    testexpr_list.SymParTypeEnv = testexpr.SymParTypeEnv;
    testexpr_list.type          = Logical; }
  | TExpList2{
    testexpr_list$1.SymParTypeEnv = MergeTypeEnv(
      testexpr_list$2.SymParTypeEnv, testexpr.SymParTypeEnv);
    testexpr_list$1.type          = Logical; } ;
expr:
  ExprNull{
    expr.SymParTypeEnv          = NullSymParTypeEnv;
    expr.type                    = NoType; }
  | TermSingle{
    expr.SymParTypeEnv          = term.SymParTypeEnv;
    expr.type                    = term.type; }
  | TermPair{
    expr.SymParTypeEnv          = NarrowTypeEnv(term.SymParTypeEnv, Number);
    expr.type                    = Narrow(Number, term.type); }
  | PlusPair, MinusPair{
    expr$1.SymParTypeEnv        = MergeTypeEnv(
      NarrowTypeEnv(expr$2.SymParTypeEnv, Narrow(Number, term.type)),
      NarrowTypeEnv(term.SymParTypeEnv, Narrow(Number, expr$2.type)));
    expr$1.type                  = Narrow(expr$2.type, term.type); } ;
term:
  PowerSingle{
    term.SymParTypeEnv          = power.SymParTypeEnv;
    term.type                    = power.type; }
  | ProdPair, QuotPair{
    term$1.SymParTypeEnv        = MergeTypeEnv(
      NarrowTypeEnv(term$2.SymParTypeEnv, Narrow(Number, power.type)),
      NarrowTypeEnv(power.SymParTypeEnv, Narrow(Number, term$2.type)));
    term$1.type                  = Narrow(term$2.type, power.type); } ;
power:
  FactorSingle{
    power.SymParTypeEnv         = factor.SymParTypeEnv;
    power.type                   = factor.type; }

```

```

| ExpPair{
  power$1.SymParTypeEnv    = MergeTypeEnv(
    NarrowTypeEnv(factor.SymParTypeEnv,Narrow(Number,power$2.type)),
    NarrowTypeEnv(power$2.SymParTypeEnv,Narrow(Number,factor.type)));
  power$1.type=Narrow(Number,Narrow(factor.type,power$2.type)); } ;
factor:
  Const{
    factor.SymParTypeEnv    = constant.SymParTypeEnv;
    factor.type              = constant.type; }
| Var{
  factor.SymParTypeEnv    = variable.SymParTypeEnv;
  factor.type              = variable.type; }
| ParenExpr{
  factor.SymParTypeEnv    = expr.SymParTypeEnv;
  factor.type              = expr.type; }
| IfFTPair{
  factor.SymParTypeEnv    = MergeTypeEnv(
    MergeTypeEnv(testexpr.SymParTypeEnv,expr$1.SymParTypeEnv),
    expr$2.SymParTypeEnv);
  factor.type              = Narrow(expr$1.type,expr$2.type); }
| FExist{
  factor.SymParTypeEnv    =
    MergeTypeEnv(expr$1.SymParTypeEnv,expr$2.SymParTypeEnv);
  factor.type              = Narrow(expr$1.type,expr$2.type); } ;
constant:
  ConstNum1{
    constant.SymParTypeEnv = NullSymParTypeEnv;
    constant.type          = Integer; }
| ConstNum2{
  constant.SymParTypeEnv = NullSymParTypeEnv;
  constant.type          = Real; }
| ConstNum3{
  constant.SymParTypeEnv = NullSymParTypeEnv;
  constant.type          = String; } ;
variable:
  VarNull{
    variable.SymParTypeEnv = NullSymParTypeEnv;
    variable.type          = NoType; }
| Variable1{
  variable.SymParTypeEnv =
    TypedSetOp(sympar.nam,NoType)::NullSymParTypeEnv;
  variable.type          = sympar.type; }
| Variable2{

```

```

    variable.SymParTypeEnv = simplevar.SymParTypeEnv;
    variable.type          = simplevar.type; }
| Variable3{
    variable.SymParTypeEnv = builtin_function.SymParTypeEnv;
    variable.type          = builtin_function.type; }
| Variable4{
    variable.SymParTypeEnv = NullSymParTypeEnv;
    variable.type          = Integer; }
| Variable5{
    variable.SymParTypeEnv = NullSymParTypeEnv;
    variable.type          = Integer; }
| Variable6{
    variable.SymParTypeEnv = arith_index_sup_fun.SymParTypeEnv;
    variable.type          = arith_index_sup_fun.type; } ;
log_index_sup_fun:
    LogIndexSupFunctionNonNull{
    log_index_sup_fun.SymParTypeEnv = testexpr.SymParTypeEnv;
    log_index_sup_fun.type          = Logical; } ;
arith_index_sup_fun:
    ArithIndexSupFunctionNonNull{
    arith_index_sup_fun.SymParTypeEnv = expr.SymParTypeEnv;
    arith_index_sup_fun.type          = Number; } ;
simplevar:
    SimpVarConst{
    simplevar.SymParTypeEnv = NullSymParTypeEnv;
    simplevar.type = get_type(gname_lookup(GNAME,simplevar.env)); }
| SimpVarName{
    simplevar.SymParTypeEnv = NullSymParTypeEnv;
    simplevar.type = get_type(gname_lookup(GNAME,simplevar.env)); } ;
builtin_function:
    BuiltinFunctionNonNull{
    builtin_function.SymParTypeEnv = expr_pack.SymParTypeEnv; }
    builtin_function.type          = NoType; } ;
expr_pack:
    ExprPackNonNull{
    expr_pack.SymParTypeEnv =
        MergeTypeEnv(expr_head.SymParTypeEnv,expr.SymParTypeEnv);
    expr_pack.type = NoType; } ;
expr_head:
    ExprHeadNull{
    expr_head.SymParTypeEnv = NullSymParTypeEnv;
    expr_head.type = NoType; }
| ExprHeadNonNull{

```

```

    expr_head.SymParTypeEnv =
      MergeTypeEnv(expr_head.SymParTypeEnv,expr.SymParTypeEnv);
    expr_head.type = NoType; } ;
sympar:
  SymParNull{
    sympar.SymParTypeEnv      = NullSymParTypeEnv;
    sympar.nam                = "";
    sympar.type                = NoType; }
  | SymPar1{
    sympar.SymParTypeEnv      = NullSymParTypeEnv;
    sympar.nam                = S_P_STEM;
    sympar.type                = NoType; }
  | SymPar2{
    sympar.SymParTypeEnv      = NullSymParTypeEnv;
    sympar.nam                = S_P_STEM;
    sympar.type                = NoType; } ;

```

The equations above make use of an additional function *get_type*, in the domain of *ATYPE* values, that is applied to attributes of the nonterminal *simplevar*. The value of *get_type* is the type value *Number* if *simplevar* corresponds to a numeric-valued simple variable, or is the logical value *String* if *simplevar* corresponds to a string-valued simple variable. The argument to *get_type* is the *Binding* returned from a call to the function *gname_lookup* (described in Section 3.5.1), with input arguments *simplevar.nam* and *variable.env*. Thus, it is necessary that all nonterminals also inherit the environment attribute *env*, in a manner similar to that described in Section 3.5. We omit these details, which can be found in [Vicu90].

Example 2. The values of the attributes *SymParTypeEnv* and *type*, for each of the nonterminals listed in Example 1, are as follows:

```

testexpr.SymParTypeEnv:
  SymParTypePair(TypedMemberOp(%ZIP Illegal) NullSymParTypeEnv)
texpr.type:          Logical
testexpr1.SymParTypeEnv: NullSymParTypeEnv

```

```

testexpr1.type:      Logical
expr1.SymParTypeEnv: NullSymParTypeEnv
expr1.type:         typeof CITY
expr2.SymParTypeEnv: NullSymParTypeEnv
expr2.type:         String
testexpr2.SymParTypeEnv:
  SymParTypePair(TypedMemberOp(%ZIP Integer) NullSymParTypeEnv)
testexpr2.type:      Logical
expr3.SymParTypeEnv:
  SymParTypePair(TypedMemberOp(%ZIP NoType) NullSymParTypeEnv)
expr3.type:          NoType
expr4.SymParTypeEnv: NullSymParTypeEnv
expr4.type:          Integer
testexpr3.SymParTypeEnv:
  SymParTypePair(TypedMemberOp(%ZIP String) NullSymParTypeEnv)
testexpr3.type:      Logical
expr5.SymParTypeEnv:
  SymParTypePair(TypedMemberOp(%ZIP NoType) NullSymParTypeEnv)
expr5.type:          NoType
expr6.SymParTypeEnv: NullSymParTypeEnv
expr6.type:          String

```

The screen display can then be annotated with the following output representation of the attribute *SymParTypeEnv*, at the place of nonterminal *testexpr*:

```

@IF(j = 5, %ZIP > 9000, %ZIP = "#TRUE"),
Where %ZIP Is Illegal

```

4.2.2 Inference of Types Implied by Domain Statements

The first step is to eliminate the optional domain construct from the SML syntax. Indeed, after removing the *op_dom_stat* operand from all genus paragraph operators, primitive entity, compound entity, attribute or variable attribute, function, and test paragraphs are defined by the production rules:

```

peparagraph:
  PeParNull()

```

```

    | PeNode(gname opindices PE_TYPE_DECL op_iss TBAR interp
      TPERIOD) ;
ceparagraph:
    CeParNull()
    | CeNode(gname opindices LPAREN calls RPAREN CE_TYPE_DECL op_iss
      TBAR interp TPERIOD) ;
avapagraph:
    AVaParNull()
    | AVaNode(gname opindices LPAREN calls RPAREN A_VA_TYPE_DECL op_iss
      op_range_stat TBAR interp TPERIOD) ;
fparagraph:
    FParNull()
    | FNode(gname opindices LPAREN calls RPAREN F_TYPE_DECL op_iss
      SEMICOLON modfunexpr TBAR interp TPERIOD) ;
tparagraph:
    TParNull()
    | TNode(gname opindices LPAREN calls RPAREN T_TYPE_DECL op_iss
      SEMICOLON modtstexpr TBAR interp TPERIOD) ;

```

Without any domain statements in the model Schema, the type of each of the elements of a self-indexed genus can now be inferred solely from the Elemental Detail Tables. The approach is similar to that shown in Section 3.5.7, where Table Content Rule A was enforced. The main difference is that the paragraph domain type characterized by the attribute *domtype*, which was previously inherited from the model Schema, now has to be synthesized within the productions that define the Elemental Detail Section. We now sketch an outline of the equations.

An attribute *domtype*, corresponding to the domain of *INTegers*, can be synthesized by the nonterminal *data*, according to the type of element. The attribute equations are the following:

```

data:
    DataOptionNull{
      data.domtype = 0; }
    | DataOption1{

```



```

    data.domtype = 1; }
| DataOption2{
    data.domtype = 2; }
| DataOption3{
    data.domtype = 3; } ;

```

The attribute *domtype* must then be synthesized via the chain of nonterminals: $data \rightarrow data_list \rightarrow data_line \rightarrow line_list$. At the same time, the attribute *in_stub*, corresponding to the domain of logical values, needs to be synthesized by the nonterminal *data_list*. The equations for *domtype* are as follows:

```

line_list:
    LineListOption1{
        line_list.domtype = data_line.domtype; }
| LineListOption2{
    line_list$1.domtype =
        Narrow(data_line.domtype,line_list$2.domtype); }
data_line:
    DataLineOption1{
        data_line.domtype = data_list.domtype; }
| DataLineOption2{
    data_line.domtype = data_list.domtype; }
| DataLineOption3{
    data_line.domtype =
        Narrow(data_list$1.domtype,data_list$2.domtype); }
data_list:
    DataListOption1{
        data_list.domtype = data.domtype; }
| DataListOption2{
    data_list$1.domtype =
        Narrow(data_list$2.domtype,data.domtype); }

```

The function *Narrow*, when given two numeric type operands, will return a narrower type according to the coercion rules of SML. The value of the synthesized attribute *domtype*, at the nonterminal *line_list*, will be a legal type if the elements are consistent, or an illegal type if they are inconsistent.

4.3 Discussion

Type inferencing has been applied in a variety of contexts. For example, it has been dealt with in polymorphic languages, like ML [Gord78], and explored in the PSG system [Bahl86], developed at the Technical University of Darmstadt. In those contexts, a unification algorithm [Robi79] is used for inferencing a construct type from its use. In the modeling language context, we show here how missing or incomplete information can be inferred in SML via an attribute grammar formalism. Many related ideas about expression type checking can be found in [Aho86].

The examples of inferencing with attribute grammars given in this chapter suggest that mathematical modeling languages may be simplified, and become easier to use, if appropriate inferencing mechanisms are integrated into the modeling environment. The modeling environment can then fill in the missing details.

Type inference makes models in the modeling language easier to write, as there are fewer details to write. The gained simplicity and clarity does not come at the expense of not performing checks; they are just deferred to a later time of the model development process. Although it might just be better to perform all type checking as early as possible via explicit declarations, the inferencing approach suggested here may perfectly well be the basis of dialects of the original modeling language. These complementary dialects can help users who might want to learn a new language, and also help experienced modelers who might want to confirm

hypotheses.

While inferencing offers many advantages, it naturally has some limitations. Among them, a principal one is the fact that inferencing only offers coarse distinctions between values. It is not possible, for example, to infer from context that symbolic parameter values should be restricted to a fixed number of digits. An explicit declaration could easily make this restriction available and enforceable by the modeling environment. Similarly, although it would be possible to determine by inference if an element's domain is either integer valued or string valued, it would not be possible to determine that it is positive integer, or short string. These kinds of fine statements can only be done via explicit declarations. Finally, explicit data type information is also useful for documentation purposes, in particular when the designer(s) and the user(s) of the model are not the same individuals.

The equations shown in Section 4.2.1 were implemented and tested in the syntax-directed editor prototype described in Chapter 6. The ideas shown in that section can also be used as a guideline to develop implementations for the other areas where it is suggested that inferencing could be applied in SML. The most important initial aspect of an implementation is to define the function *Narrow*, according to the semantic rules of the modeling language. The next concern is then to describe how these inferred conclusions can be broadcast along a parse tree of the model being described.

CHAPTER 5

Expression Evaluation in Mathematical Modeling Languages

Mathematical modeling languages, in general, can model elements whose values are computed by indexed functions from some domain space to a range space. A natural objective is to have a language-specific translator in the modeling environment automatically generate (without any imperative programming) the code to evaluate general classes of indexed families of expressions, as the model is designed and/or modified. Furthermore, if this translator can be combined with a resident execution tool (an interpreter), an immediate evaluation capability can be provided by the modeling environment.

To illustrate this objective, consider for example a structured model that represents the sequence of Fibonacci numbers. The numbers in this sequence, which are denoted by F_n , are defined as $F_0=0$, $F_1=1$, $F_{n+2}=F_{n+1}+F_n$, $n \geq 0$. A general model structure (Schema), written in the recursive version of SML, is shown in Figure 5.1. A particular model instance, comprising the first five numbers in the sequence, is given in the Elemental Detail Table of Figure 5.3. That table is structured according to the labels depicted in Figure 5.2. Referring to the model instance, the function elements could have their corresponding values in column *FIB_VAL* of Elemental Detail Table *FIB* calculated automatically, immediately

after the indexed function expressions are written or modified in the Schema.

```

FS /pe/ ~| FIBONACCI SEQUENCE. ~.

FIBn (FS) /f/ ;
@IF(n=1, 0, @IF(n=2, 1, FIB<n-1> + FIB<n-2>))
~| Each ~/FIBONACCI~/ number is the sum of
the preceding two FIBONACCI numbers. ~.

```

Figure 5.1: Schema for a Model of Fibonacci Numbers.

Table Name	Column Names
FIB	FIB INTERP FIB_VAL

Figure 5.2: Elemental Detail Table Structure.

FIB	INTERP	FIB_VAL
=====		
0	First number	0
1	Second number	1
2	Third number	1
3	Fourth number	2
4	Fifth number	3

Figure 5.3: Elemental Detail Table FIB.

The capability to evaluate indexed families of mathematical expressions of great complexity, for example those which can be written in SML, is an important requirement for the next generation of modeling environments. In this chapter, we show how to provide a compiler for SML's indexed families of expressions via an

attribute grammar framework. This compiler could be combined with a resident execution tool (interpreter) to endow a syntax-directed editor for SML, like the prototype presented in Chapter 6, with the integrated ability to support immediate evaluation of expressions. With changes to account for differences in surface syntax, the attribute grammar-based approach to compiling that is introduced for SML is adaptable to other mathematical modeling languages as well.

The organization of the chapter is as follows. To place this work in perspective, Section 5.1 reviews several methods for semantic description of programming languages, which are relevant to expression evaluation in mathematical modeling languages. Two particular methods are examined in more detail: attribute grammars, and denotational semantics. Section 5.2 sketches the architecture of a compiler of SML expressions into executable program fragments. The paradigm for expressing this language-specific compiler is based on attribute grammar equations. Finally, Section 5.3 provides a discussion.

5.1 Related Approaches for Semantic Evaluation

For the purpose of implementing an evaluator of indexed families of expressions, properties like the maintenance of the focus of control of the execution, and the assignment of values to specific elements, need to be incorporated in the evaluation tool. However, a simple inspection of the model's Schema and its static semantics is insufficient for deriving these kinds of properties, which are known as “dynamic”, or “behavioral” semantics. Indeed, after the first-stage translation of a subject

model into an intermediate fragment, its actual evaluation by the execution tool (interpreter and/or symbolic debugger) requires execution time (run time) support from the modeling environment. Unfortunately, most language-based system generators do not provide this kind of support.

Recent research in language-based environment generators that support the description of both static and dynamic semantics falls typically in the class of action routines, attribute grammars, and denotational semantics. Action routines are either written in a procedural programming language as, for example, in the parser generation system YACC, or in a special-purpose action-routine language. Action routines are always associated with the particular productions of the grammar, and are invoked automatically when an editing command is applied to a node in the syntax tree representing the model. The other two approaches to semantic description are discussed in the balance of this section.

5.1.1 Attribute Grammar Methods

Attribute grammars are an alternative semantic processing paradigm used in various language-based generator systems. This declarative approach, however, is generally limited to a small subset of the processing aspects of dynamic semantics. Indeed, as shown by the review made in [Dera86], most systems based on, or related to, attribute grammars have no special provision for code generation. Furthermore, the interpretation of generated code is not generally supported by attribute grammar based generator systems.

For the code generation phase, an approach is to emit “code templates” during the traversal of the syntax-tree. For example, in the MUG 1 and MUG 2 systems [Dera86], there is a code template for each operator (node) of the abstract syntax tree. Each code template is an action to either visit a (son) subtree, compute some local values, or output some intermediate code using the node’s attributes and local values. This customized approach needs special routines, outside the attribute grammar, to translate the intermediate code to machine language code.

For the specification of run time semantics, a generic approach is to take a semantic definition written in a lambda-calculus notation, and to translate the definition into an equivalent, directly executable, intermediate language. This intermediate language, together with a subject test program written in the modeling language, say SML, are then given to an interpreter which executes the test program to obtain its dynamic semantics. Arbab [Arba86], for instance, suggests taking an attribute grammar specification of a programming language, translating it into Prolog code, and then supplying the subject program plus the Prolog code to a Prolog interpreter or compiler to produce object code in the form of lambda expressions. These expressions can then be given to a lambda machine to produce the output corresponding to the execution of the subject program.

In the Synthesizer Generator developed at Cornell [Reps89a], which proposed the use of attribute grammars for generating language-based environments, the syntax and static semantics of the subject language are defined using SSL (Synthesizer Specification Language), but the specification of run time semantics is not

supported. Hence, the attribute evaluation scheme(s) employed by the Synthesizer Generator are not completely adequate for the purposes of code interpretation, as required for immediate evaluation of indexed families of expressions.

A recent extension to attribute grammars, action equations [Kais89], embeds rules similar in form to semantic equations, which reflect the interaction between the user and the environment, in an event-driven architecture. This extension has been developed to permit the generation of language-based environments which support the specification of both static and dynamic semantics.

5.1.2 Denotational Semantic Methods

Denotational semantics is another paradigm for specifying dynamic semantics processing: interpretation, in particular. The basic problem, however, is that as in attribute grammar paradigms, denotational specifications are restricted to only small aspects of dynamics processing.

The central concept behind denotational semantics is that the meaning of a language can be computed in terms of the meaning of the constituents of the language [Gord79]. The meaning of more primitive constituents can be expressed as mappings from the sentences (or abstract trees) to sets of functions written in a lambda-calculus. It has been shown that a denotational semantic specification of a language can be mapped into a synthesized-only attribute grammar definition [Chir79]. Thus, both denotational semantics and attribute-grammars are equally powerful semantic definitions. However, this equivalence comes at a cost: in the

synthesized-only attribute grammar definition, attributes have functional values, making the definition more complex and harder to read.

In the approach taken in [Kini82], the ADA formal definition, written in denotational semantic style, is translated with a software tool into a directly executable intermediate language. Candidate test programs are mapped into corresponding abstract syntax-trees. The translated definition is applied to the abstract trees to obtain the semantics of the candidate programs. The execution phase is performed with a special-purpose interpreter written in Interlisp. Wand [Wand84] describes related software tools which have been developed to translate a denotational semantic specification of a language into terms of a lambda-calculus. The terms are actually functions of Scheme 84, a version of Franz Lisp, which serves as the lambda-calculus interpreter. An interface to the YACC parser generator permits the use of concrete input syntax, rather than abstract syntax for its input programs.

5.2 Immediate Expression Evaluation in SML

Several alternatives are available for the formal specification of the evaluation properties of indexed families of expressions in SML. A generic, although complex, approach would be to produce both a denotational semantic definition for SML, and a translation of subject structured models into executable representations in lambda-calculus form. Then, the evaluation of function and test genera could be carried out by a LISP interpreter which would take as inputs the executable

representation of the denotational semantic specification of SML and the abstract syntax-tree representing the model, and then would execute the model (traverse the abstract tree) to produce new output values for the affected model elements. This approach would require execution time support from the modeling environment to handle aspects like the order of evaluation of functions, and the assignment of values to elements.

An alternative architecture is to consider designing a compiler of SML expressions directly within an attribute grammar framework, and to incorporate an execution tool into the modeling environment that will interpret the compiled code. This is the strategy adopted in the remainder of this section. Among the advantages derived from this selection are the modularity of the generated code, the readability of this code, and the compact size of constituent code fragments.

5.2.1 Example of Evaluation Code

This subsection shows the type of code that is required to evaluate the function *genera* in the example of the structured model of Fibonacci numbers shown in Figure 5.1. The code is written in FRED, Framework III's built-in, interpreted, language [Asht88]. Framework III is a popular DOS multi-function program that supports many modeling system functions, including word processing, outlining, spreadsheets, databases, graphs, and telecommunications. One reason for choosing FRED to describe this example is that the language contains many of the primitive functions that are required to support tabular databases and spreadsheets. If

Elemental Detail Tables are stored in databases, the data handling aspects can be completely specified in FRED. Another reason for selecting FRED is that the language is highly readable. Nevertheless, FRED is only used here as a vehicle to explain the kinds of operations which are required to evaluate function and test genera and, therefore, any other procedural language might have been chosen as well. In this latter case, it is assumed that Elemental Detail data can be stored within some database management system which can be programmed in the chosen procedural language.

For evaluation purposes, the meaning of the generic rule of paragraph *FIB* in Figure 5.1 is the equivalent code fragment of Figure 5.4. The essentials of this

```

1  f:="@if(@~Calc.x.[is n]([FIB]),
2    [FIB_VAL] := @if(@~Calc.x.[ir n] = 1 ,
3      0 ,
4      @if(@~Calc.x.[ir n] = 2 ,
5        1 ,
6        @~Calc.x.[g FIB_VAL]("""n""",-1) +
7        @~Calc.x.[g FIB_VAL]("""n""",-2)
8      )
9    )
10 ",
11 @setdefining("[FIB].[FIB_VAL]",f),

```

Figure 5.4: Defining Formula

fragment may be understood in the following.

In general, each function or test genus has an associated defining formula f . In this particular model with only one function genus, the defining formula f (lines 1

through 10) is assigned to the *FIB_VAL* column of table *FIB* via the *@setdefining* function in line 11. Since the genus for *FIB_VAL* only has one symbolic index (there is a single column to the left of the “||” field in Figure 5.3), there is only one Index Setup function call in line 1. The Index Setup function, *@ Calc.x[is n]*, is called with the argument *[FIB]*, which evaluates to the current value of the stub (“0” for the first row, “1” for the second row, etc.).

The generic Index Setup function code, given in Figure 5.5, creates the global variable *@ Calc.x[ir n]* via the *@makevariable* function in line 12, and assigns to it (in lines 4 and 7) the row number in the table that introduces the symbolic index (i.e., *[FIB].[FIB]*) where the actual parameter value matches the table identifier value. If a match is found, the function returns *#TRUE* but, if the actual parameter value is not a valid identifier, a match will not be found and the Index Setup function returns *#FALSE*.

```

1  f:=";is n
2  @local(i),
3  i:=@reset([FIB].[FIB]),
4  x.[ir n] := 1,
5  @while(@and(i<>#NULL!,i<>@item1),
6         i:=@next([FIB].[FIB]),
7         x.[ir n] := x.[ir n] + 1
8  ),
9  @return(@and(i<>#NULL!,i<>#N/A!))
10 ",
11 @makeformula("~Calc.x.[is n]",f),
12 @makevariable("~Calc.x.[ir n]"),

```

Figure 5.5: Index Setup Function

Returning to Figure 5.4, the `@if` function in line 1 causes the calculation to be aborted if the return value of `@ Calc.x[is n]` is `#FALSE`, which would indicate that the stub value is incorrect. Otherwise, the true clause of the `@if` function is executed, and column `[FIB_VAL]` is assigned the value of the expression to the right of the `:=` sign in line 2.

The expression to evaluate (lines 2 through 9 of Figure 5.4) is practically a literal translation of the model's generic rule. Indeed, the value of the current row position in the stub is maintained by variable `@ Calc.x[ir n]`. For the first two row positions, this variable evaluates to 1 and 2, in this order; hence, genus `FIB_VAL` evaluates to 0 and 1, respectively. For row positions greater than two, the value of genus `FIB_VAL` is determined in line 6 by the sum of the return values of the two calls to the generic Genus Value function listed in Figure 5.6. The first call passes the pair $(n,-1)$ as actual parameters, corresponding to the index identifier parameters of the previous row. The second call passes the pair $(n,-2)$ as arguments, corresponding to the index identifier parameters of the next to the previous row.

```

1  f:=";g FIB_VAL
2  @choose(@ Calc.x.[ts FIB](@item1,@item2), [FIB].[FIB_VAL])
3  ",
4  @makeformula("~Calc.x.[g FIB_VAL]",f),

```

Figure 5.6: Genus Value Function

A Genus Value function returns the value of genus `FIB_VAL`, given the current

settings of the variable `@Calc.x[ir n]`, and appropriate offset modifications. The `@choose` function in line 2 is used to extract a value from the `[FIB].[FIB-VAL]` column, at the current rank settings. This current rank setting is determined after a call to the Table Search function `@Calc.x[ts FIB]`, whose generic code is shown in Figure 5.7.

```
f:=";ts FIB
;Lookup index:
@x.[il n](@item1,@item2),
;Check if same as last search:
@if(x.[iv n] = x.[ti FIB n],
    @return(x.[tp FIB])),
;Remember we did this search:
x.[ti FIB n] := x.[iv n],
;Initialize search loop:
@reset([FIB].[FIB]),
x.[tp FIB] := 1,
;Search:
@while(@get([FIB].[FIB]) <> #NULL!,
    @if(@get([FIB].[FIB]) = x.[iv n],
        @return(x.[tp FIB])),
    @next([FIB].[FIB]),
    x.[tp FIB] := x.[tp FIB] + 1),
;Not found, return #N/A!
@return(x.[tp FIB] := #N/A!)
",
@makeformula("~Calc.x.[ts FIB]",f),
@makevariable("~Calc.x.[ti FIB n]"),
@makevariable("~Calc.x.[tp FIB]"),
```

Figure 5.7: Table Search Function

The Table Search function `@Calc.x[ts FIB]` searches the table `FIB` that contains genus `FIB-VAL`, for the record number such that its stub values match the position specified in the formal parameters. It employs the auxiliary Index Lookup

function, whose generic code is listed in Figure 5.8.

```
f:=";il n
;Simple index case:
@if(@or(@and(@item1=#NULL!,@item2=#NULL!),
          @and(@item1=#N/A!,@item2=#N/A!)),
    @list(x.[iv n] := @choose(x.[ir n], [FIB].[FIB]),
          @return(x.[ir n]))),
;Get index letter offset:
@local(macro,rank),
@if(@isalpha(@item1),@list(
    macro := "x.[ir "&@item1&"",
    @iserr(rank := @macro),
    @if(@not(@isnumeric(rank)),0)),
    rank := 0),
;Add constant offset:
@if(@isnumeric(@item2),
    rank := rank + @item2),
;Fixup negative indices
@if(rank < 0,
    rank := @panel2("[FIB].[FIB]") + rank + 1),
x.[iv n] := @choose(rank, [FIB].[FIB]),
@return(rank)
",
@makeformula("~Calc.x.[il n]",f),
@makevariable("~Calc.x.[iv n]"),
```

Figure 5.8: Index Lookup Function

The generic Index Lookup function $\text{@Calc.x}[il\ n]$ sets up the value of the variable $\text{@Calc.x}[iv\ n]$ calculated from the current rank $\text{@Calc.x}[ir\ n]$.

This completes the explanation of the code fragments. It is necessary to add that in the code generation scheme above, the supporting spreadsheet @Calc. is used to store formulas and to hold the values of variables. Figure 5.9 shows the row and column labeling structure for this spreadsheet.

Row Names	Column Name
	x
[is n]	
[ir n]	
[g FIB_VAL]	
[ts FIB]	
[ti FIB]	
[tp FIB]	
[il n]	
[iv n]	

Figure 5.9: Spreadsheet “~@Calc”.

5.2.2 An Attribute Grammar Based Compiler

The code fragments required to evaluate function and test genera in FRED can be generated by an attribute grammar based compiler. This subsection shows how to specify such a compiler in order to generate lines 2 through 9 of the Defining Formula fragment of Figure 5.4. This approach can be easily extended to generate the complete code fragments given by Figures 5.4 through 5.8, if table structuring information is available. Issues about actual code evaluation are deferred to the discussion presented in the next section.

The code generation strategy suggested here automatically recalculates the evaluation code when the Schema is significantly altered. If the data in Elemental Detail Tables are changed, the code is not regenerated; it only ought to be reexecuted to assign new values to elements.

In the attribute grammar equations, the single synthesized attribute *code*, from the domain of *STRings*, is attached to the generic rule nonterminals. Only the

nonterminals affected in the generation of the Defining formula code fragment are shown below.

```
fparagraph:
  FNode{
    fparagraph.code = "["#gname.code#" := "#modfunexpr.code; }
;
tparagraph:
  TNode{
    tparagraph.code = "["#gname.code#" := "#modtstexpr.code; }
;
gname:
  GenNameNonNull{
    gname.code = GNAME; }
;
modfunexpr:
  MFunExpr{
    modfunexpr.code = funexpr.code; }
;
modtstexpr:
  MTstExpr{
    modtstexpr.code = testexpr.code; }
;
testexpr:
  | LitConst{
    testexpr.code = LITERAL; }
  | TestPair{
    testexpr.code = expr$1.code#" "#relop.code#" "#expr$2.code; }
  | IfTPair{
    testexpr$1.code = "@if("#testexpr$2.code#" ,\n"#
      testexpr$3.code#" ,\n"#testexpr$4.code#" \n)"; }
;
relop:
  RelOpLt{
    relop.code = LT; }
  | RelOpLe{
    relop.code = LE; }
  | RelOpEq{
    relop.code = EQ; }
  | RelOpGt{
    relop.code = GT; }
  | RelOpGe{
```

```

        relop.code = GE; }
    | RelOpNe{
        relop.code = NE; }
;
funexpr:
    FuncExprNonNull{
        funexpr.code    = expr.code; }
;
expr:
    | TermSingle{
        expr.code       = term.code; }
    | TermPair{
        expr.code       = term.code; }
    | PlusPair{
        expr$1.code     = expr$2.code#" "#PLUS_SIGN#" "#term.code; }
    | MinusPair{
        expr$1.code     = expr$2.code#" "#MINUS_SIGN#" "#term.code; }
;
term:
    PowerSingle{
        term.code       = power.code ; }
    | ProdPair{
        term$1.code     = term$2.code#" "#MULT_SIGN#" "#power.code; }
    | QuotPair{
        term$1.code     =
            term$2.code#" "#DIVIDE_SIGN#" "#power.code; }
;
power:
    FactorSingle{
        power.code      = factor.code; }
    | ExpPair{
        power$1.code    = factor.code; }
;
factor:
    Const{
        factor.code     = constant.code; }
    | Var{
        factor.code     = variable.code; }
    | ParenExpr{
        factor.code     = expr.code; }
    | IfFTPair{
        factor.code = "@if(\n"#"#testexpr.code#" ,\n"#"#expr$1.code#"
            " ,\n"#"#expr$2.code#"#"\n)"; }

```

```

;
constant:
  ConstNum1{
    constant.code      = nninteger.code; }
  | ConstNum2{
    constant.code      = NN_REAL; }
;
variable:
  | Variable2{
    variable.code      = simplevar.code; }
  | Variable4{
    variable.code      = "@~Calc.x.[ir "#pp_index.code#" ] "; }
;
simplevar:
  SimpVarConst{
    simplevar.code     = "@~Calc.x.[g "#GNAME#" ](,)"; }
  | SimpVarName{
    simplevar.code     =
      "@~Calc.x.[g "#GNAME#" ]("#grindices.code#" ); }
;
grindices:
  GrIndices1{
    grindices.code = grule_index.code; }
  | GrIndices2{
    grindices$1.code = grule_index.code#" ,"#grindices$2.code; }
;
grule_index:
  | GrInd1{
    grule_index.code = "\"\\""#pp_index.code#"\"\","; }
  | GrInd2{
    grule_index.code = ", "#replaced_index.code; }
  | GrInd3{
    grule_index.code = offset_index.code; }
;
replaced_index:
  ReplacedIndex1{
    replaced_index.code = ", "#optsign.code#pinteger.code; }
;
offset_index:
  OffsetIndexNonNull{
    offset_index.code =
      "\"\\""#pp_index.code#"\"\", "#sign.code#pinteger.code; }
;

```

```

optsign:
    OptSignNonNull{
        optsign.code = sign.code; }
;
sign:
    SignPlus{
        sign.code = PLUS_SIGN; }
    | SignMinus{
        sign.code = MINUS_SIGN; }
;
pinteger:
    One{
        pinteger.code = STR; }
    | Pinteger{
        pinteger.code = P_INTEGER; }
;
nzinteger:
    NzInteger1{
        nzinteger.code = pinteger.code; }
    | NzInteger2{
        nzinteger.code = sign.code#pinteger.code; }
;
nninteger:
    NnInteger1{
        nninteger.code = pinteger.code; }
    | NnInteger2{
        nninteger.code = STR; }
;
integer:
    Integer1{
        integer.code = nninteger.code; }
    | Integer2{
        integer.code = sign.code#nninteger.code; }
;
index:
    Index1{
        index.code = INDEX; }
    | Index2{
        index.code = C_P_TOKEN; }
;
pp_index:
    PpIndex1{
        pp_index.code = index.code; }

```

```

| PpIndex2{
  pp_index.code = S_P_INDEX; }
| PpIndex3{
  pp_index.code = D_P_INDEX; }
;

```

Example. The value of the attribute *code*, for the nonterminal *fparagraph*, is shown below. As expected, it is exactly that code of lines 2–9 of Figure 5.4.

fparagraph.code →

```

@if(@~Calc.x.[ir n] = 1 ,
0 ,
@if(@~Calc.x.[ir n] = 2 ,
1 ,
@~Calc.x.[g FIB_VAL] ("n",-1) + @~Calc.x.[g FIB_VAL] ("n",-2)
)
)

```

5.3 Discussion

Several approaches exist for evaluating the code generated in the way shown in Section 5.2. For example, FRED code could be generated by an attribute grammar based compiler, exported to an output file, and then read and executed by a built-in interpreter in a Framework III environment. Alternatively, if a database management system is interfaced to a syntax-directed editor in an attribute grammar based modeling environment, the generated code would contain function calls against the database. These functions would form a library that would be compiled and run under some solver control. The database would be immediately updated by the solver's output.

If the data is maintained in the syntax-directed editor described in Chapter 6, for example, it is possible to provide direct, but inefficient, data access for updating and evaluation. In this case, the syntax-directed editor would need to be provided with an interpreter to run the compiled code fragments. This direct execution approach could be considered a better design approach if the code is generated and compiled incrementally, so as to reduce the work performed by the attribution process.

An important application of the ideas of code generation presented in this chapter can be made to the translation processes that are required in optimization modeling. Indeed, an approach to integrate a mathematical programming solver into a modeling environment is to have a translator convert the model described in the modeling environment to the representation suitable for the solver. The main tasks performed by this translator are equivalent to the lexical, syntactic, and semantic phases performed by a compiler. Thus, the actions of a mathematical programming translator are naturally suited to the attribute grammar approach, since all information required about the model and its data can be kept and manipulated globally via attributes. For example, the techniques of code synthesis via the simple synthesized attribute *code*, described in Section 5.2.1, can serve to illustrate similar parts of the semantic analysis phase of the translator. It remains open to further research how the other semantic tasks performed by the translator, such as algebraic manipulations, symbol replacements, and detection of special structures, might be performed with attribute grammars.

CHAPTER 6

Implementation of a Syntax-Directed Editor Prototype

This chapter describes the implementation of a syntax-directed editor for SML. A prototype editor, which satisfies all the syntactic constraints and Schema Properties of SML, together with some Table Content Rules, has been built with the Synthesizer Generator—an attribute grammar based generator. This prototype fulfills the modeling environment goals of easy modifiability and of non-procedural specification advocated in Chapter 3.

To implement a modeling environment for SML that is easily modifiable, and that is also specified non-procedurally, different language-based generator approaches were considered. However, among several language-based generator systems, only a few prototypes, like MENTOR [Donz80] and GANDALF [Elli85], are not tailored for a particular application. The GANDALF system, for example, uses action routines written in GC, a dialect of C. The PSG system [Bahl86], developed at the Technical University of Darmstadt, uses the concept of “context relations” to generate language-specific environments. A so-called data attribute grammar is used specify the context analysis.

Attribute grammar-based generators are another popular approach to achieving easy modeling environment modifiability. In the Synthesizer Generator, de-

veloped at Cornell University [Deme81,Reps89a,Reps89b], for example, context-sensitive aspects of a modeling language are declared in SSL (Synthesizer Specification Language), a specification language which is based on attribute grammars and closely resembles other specification languages like YACC.

The advantages of the attribute grammar approach over ad-hoc action routines for ease of maintenance influenced our decision to choose the Synthesizer Generator to generate a desired syntax-directed editor for SML. An immediate benefit of this choice is that, via the generated prototype editor, we are able to independently validate the attribute grammar specification of Schema Properties and Table Content Rules developed for SML.

The chapter is organized as follows. Section 6.1 explains the Synthesizer Generator's specification language SSL. The implementation environment is then briefly described in Section 6.2. Examples of four erroneous subject models, with the corresponding error messages displayed on the screen by the syntax-directed editor, are shown in the balance of the chapter.

6.1 A Syntax-Directed Editor Description in SSL

The description of a syntax-directed editor in SSL requires a specification of the following items:

Concrete Syntax Declarations. A concrete syntax is provided in order to read and write a file containing a subject test model. The notation employed is basically BNF. In SML, for instance, there is a root node labeled *SchemaFile*, and leaf nodes

denoted in all upper-case letters, by convention. A typical production is written

```
SchemaFile ::=
    (Schema EOF OptElementDetail)
    { $$ .t = SchemaFileList(Schema.t, EOF, OptElementDetail.t); }
;
```

The left-hand side of every production is located to the left of the “::=” symbol, and the corresponding right-hand side is a set of grammar symbols enclosed in parenthesis ‘(’ and ‘)’. Right-hand side rules associated with the same left-hand side nonterminal are separated by a bar (‘|’) symbol. Associated with each production are equations enclosed in left brace (‘{’) and right brace (‘}’) symbols. These equations translate the concrete input syntax into an abstract syntax used for internal manipulation. Individual productions are separated from each other by a semicolon symbol (‘;’).

Lexical Syntax Declarations. Terminal symbols in the attribute grammar are either predefined, or are denoted using a regular expression language. While some primitive types (like INT, BOOL, and STR) are predefined in the system, the rest of the tokens need to be explicitly declared. By convention, every terminal name is listed in uppercase, and its corresponding lexical rule is enclosed by less-than and greater-than symbols (‘<’ and ‘>’). The syntax for regular expressions is almost identical to that of LEX, with some minor restrictions. For example, a lexical rule for a genus name GNAME

```
< (([A-Z] | "$") ([&@\\:#!\[%\] ' _] | [0-9] | [A-Z])*) >
```

states that a genus name is any of the upper case letters or the dollar sign, followed by zero or more occurrences of those characters listed inside the first set of square

brackets, or the digits 0–9 or the letters A–Z.

Abstract Syntax Declarations. The abstract syntax must match the concrete input syntax. As described in Chapter 3, abstract syntax mimics concrete syntax, but it usually ignores “syntactic sugaring”. For example, for the concrete syntax declaration shown earlier, the corresponding abstract syntax is written

```
root schemafile;  
schemafile:  
    SchemaFileList(schema_par EOF opt_element_detail)  
;
```

The complete abstract syntax for SML is included in Appendix A.

It remains to explain the role of completing productions in abstract syntax declarations. This distinguished production is usually a 0-ary production which is used to construct default tree representations in the case of incomplete concrete syntax. In this fashion, it is always possible to maintain a complete abstract syntax tree in the presence of incomplete model fragments. An example in Section 3.3 showed how the completing production *ExpNull* could be used to represent an incomplete model fragment.

Type Definitions. One of the best features of the SSL language is that its derived types (those which are not the primitive types STRing, BOOLEan, etc.) can be expressed in the same notation that is used for declaring abstract syntax. Thus, for example, in Chapter 3 the new type BINDING was declared as a subclass of type STR, and type ENV was declared as a list of BINDINGs. Type BINDING was returned by *gname_lookup*. This lookup function takes the identifier “*id*” and the environment “*env*” as formal parameters, and recursively searches the list of

BINDINGS looking for a match with the formal parameter *id*. If such a match exists, that BINDING is returned. Otherwise, lookup returns the '?', or unknown, BINDING. Type definitions are generally required for the attribution of abstract syntax.

Attribute Declarations. Attribute declarations define the domain of values of every synthesized and inherited attribute associated with every abstract nonterminal symbol. These declarations must precede attribute use in the attribution equation scheme. Every nonterminal in the abstract syntax which uses an attribute declares it by specifying the type of the attribute and whether it is inherited or synthesized. It is not necessary that nonterminals have both inherited and synthesized attributes, nor it is required for a nonterminal to have any attribute declared at all.

Parse Syntax. We can now explain the meaning of the equations enclosed in braces in the concrete input syntax. These equations represent the translation scheme from concrete to abstract syntax. The left hand side of each equation assigns the value of the operator of the right hand side to the synthesized (syntax tree) attribute '*t*' of the left hand side. For the concrete syntax production shown earlier, for example, *t* is a synthesized attribute for *schemafile*. The declaration states that every time the cursor is in location *SchemaFile* in the concrete syntax, the input should be parsed as a *schemafile* in the abstract syntax.

Unparsing Declarations. The display representation of the abstract syntax tree is provided by the unparsing declarations. Normally, the screen display will con-

tain the representation of terminal strings found in a left to right traversal of the abstract syntax tree. However, because not all terminal symbols are carried in the abstract tree, additional tokens to be displayed are explicitly indicated in the unparsing declaration. Strings enclosed in quotes will be displayed interspersed with the terminal symbols derived from each nonterminal. Attributes can also be made to have visible display representations, like the *error* attribute, for example, which either evaluates to the null string or to a nonempty error message to appear in place.

Attribution of Abstract Syntax. The core of a description in SSL is provided by the attribute grammar equations. For SML, the attribute grammar equations are provided in [Vicu90]. Equations have been provided to enforce all Schema Properties, and various Table Content Rules. Some of these equations have already been described in Chapter 3.

Entry Declarations. A final aspect of an attribute grammar definition in the SSL language consists of entry point declarations. These points declare those nonterminals which are editable by the system. Thus, when the selection is at the editable nonterminal in the concrete syntax, input is to be parsed as given in the abstract syntax and should be inserted appropriately in the syntax tree denoted by *t*. Grammar symbols which are not declared as entry points become immutable, and as such are not subject to change.

6.2 Prototype Implementation

A prototype of a syntax-directed editor has been developed and implemented on an RT PC workstation, running X-Windows under UNIX Berkeley 4.3 [Ritc74]. Portability has been one of the design requirements; this prototype is capable of running on any environment to which the Synthesizer Generator system can be ported. The languages and tools used are standard ones provided in UNIX environments: C, LEX, YACC, make, and sed.

The Schema Properties of SML have been implemented in about 3400 lines of attribute-grammar style equations. The complete environment (syntactic analyzer, parser, and unparser) is written in about 7100 lines of pseudo-code.

Table Content Rules A, B, E, I, and L of SML have been implemented in about 500 lines of additional attribute-grammar style equations. The complete environment (Schema Properties and implemented Table Content Rules, including syntactic analyzer, parser, and unparser) now includes about 7800 lines of special-purpose code; all details are rendered in [Vicu90]. The non-implemented Table Content Rules can only be verified by the user (Rules D and K), or require understanding a relational algebraic expression implied by an index set statement (Rule F), or demand cognizance of element feasibility (Rules C, G, H, and J).

The prototype environment has been tested with a collection of problems extracted from various domains, including Engineering and Operations Research. Over 100 SML models collected in [Geof89e] and elsewhere have been used as

subject test models.

Response-time on the RT PC is rather slow when large models are tested. Complex models with 40–50 paragraphs, few tables, and few elements per table, are handled quickly, with few seconds response time. Larger and more complex models, with many tables and many element rows per tables, however, bring the performance down very quickly. Performance of attribution evaluation is viewed as a limitation, but some performance enhancements can be achieved with other attribute evaluation algorithms, and faster workstation hardware.

The syntax-directed editor has successfully handled the general structure of a model as represented by the Schema section. The specific instance of a model is represented by the Elemental Detail section. In the syntax-directed environment, Elemental Detail is given in tables, in a section that follows the Schema Section. Elemental Detail is represented in structured tables. The Table data are input interactively, or read from files and stored in internal abstract trees. The Elemental Detail Table Content Rules have been cast in an attribute grammar framework.

The following subsections show some representative output generated by the syntax-directed editor for SML. Only semantic errors are displayed. These semantic errors are displayed on the screen as soon as the modeler types in the faulty model fragment, or immediately after the input file containing the model is read. Because the editor operates in an *immediate evaluation* mode, the first syntax error in the model is shown directly on the screen, and processing is suspended until the syntax error is fixed. Hence, in the tested models, all the syntax errors have

been corrected, and thus are not shown. Normally, a syntax error is marked by the cursor, which positions itself immediately below the location of the error. All syntax errors have to be fixed before a model can be successfully parsed and all its semantic errors displayed, if any.

6.2.1 Example of Semantic Restrictions in Paragraphs

The following model contains violations of semantic restrictions <<P1>>-<<P8>>. The error messages are shown in uppercase.

Aa /pe/ ~| ~.

&MOD1 ~| ~.

B /pe/ ~| ~.

&MOD1<--ERROR <<P4>>: MODULE NAME PREVIOUSLY DEFINED-->
~| Duplicate module name ~.

C /pe/ ~| ~.

C<--ERROR <<P5>>: GENUS NAME PREVIOUSLY DEFINED-->
/pe/ ~| Duplicate genus name ~.

D /pe/ ~| new paragraph ~.

<--ERROR <<P2>>: INCORRECT INDENTATION FOR GENUS-->
EE /pe/ ~| This paragraph is indented (3 blanks) more than last. ~.

Ja<--ERROR <<P7>>: DUPLICATED SYMBOLIC INDEX-->
/pe/ ~| Duplicate index (see A) ~.

Kb,c,b<--ERROR <<P7>>: DUPLICATED SYMBOLIC INDEX-->
/pe/ ~| Not all aliases distinct. ~.

L /pe/ :: Char <--ERROR <<P8>>: DOMAIN STATEMENT CANNOT APPEAR-->
~| Unindexed genus can't have domain statement. ~.

NN /pe/ ~| ~.

Nn /pe/ :: Str ~| Spelled Str or String ~.

NN<--ERROR <<P5>>: GENUS NAME PREVIOUSLY DEFINED-->
/pe/ ~| Duplicate declaration of genus NN. ~.

QQ (B) /ce/ ~| No errors. ~.

&MOD!\$%&':@[\]_ ~| All the acceptable special characters ~.

YYY!\$%&':@[\]_ /pe/ ~| genus name has all possible special
characters ~.

XXXX (YYY!\$%&':@[\]_) /ce/ ~| ~.

&MOD2 ~| ~.

<--ERROR <<P3>>: INCORRECT INDENTATION FOR MODULE-->
ZZZ /pe/ ~| Violates indentation. ~.

R /pe/ ~| ~.

&MOD3 ~| ~.

<--ERROR <<P1>>: INCORRECT NUMBER OF INDENTATION BLANKS-->
S /pe/ ~| Indentation is 3, while &MOD3 is indented 1. ~.

T /pe/ ~| ~.

U_VAL<--ERROR <<P6>>: GENUS NAME IS INVALID-->
/pe/ ~| Forbidden genus name. ~.

V /pe/ ~| ~.
~.~.

6.2.2 Example of Semantic Restrictions in Calling Sequences

The following model contains violations of semantic restrictions <<P1.1>>-<<P1.11>>, and <<P2.4>>. The error messages are shown in uppercase.

Aa /pe/ ~| ~.

Bb /pe/ ~| ~.

Cc /pe/ ~| ~.

CC (AA<--ERROR <<P1.1>>: GENUS NAME IS UNDEFINED-->
) /ce/ ~| AA not defined in previous paragraph ~.

D (A) /va/ ~| ~.

E (D<--ERROR <<P1.2>>: INAPPROPRIATE GENUS TYPE-->
) /ce/ ~| /ce/ can't call /a/ ~.

G (Aa, Bb) /ce/ ~| ~.

H (A<4:3><--ERROR <<P1.3>>: NUMERIC RANGE IS INVALID-->,B<-3:-4>
<--ERROR <<P1.3>>: NUMERIC RANGE IS INVALID-->
) /ce/ ~| 2 violations of numeric range. ~.

I (A<1:2>) /ce/ ~| N:M are nonzero integers. ~.

J (A<a+1>) /ce/ ~| P is a positive integer ~.

K (A<a+4:4><--ERROR <<P1.4>>: NUMERIC RANGE IS INVALID-->
) /ce/ ~| Violates numeric range. ~.

M (A<a+4:a+3><--ERROR <<P1.5>>: NUMERIC RANGE IS INVALID-->
) /ce/ ~| Violates range limits. ~.

N (A<a-4:a-5><--ERROR <<P1.5>>: NUMERIC RANGE IS INVALID-->
) /ce/ ~| Violates range limits. ~.

Q (Aa3(b,c,b<--ERROR <<P1.6>>: DUPLICATE INDEPENDENT INDICES-->
) ,Bb,Cc) /ce/ ~| Duplicate independent indices
(violates minimality and non--duplication of indices). ~.

U (Aa5(b),Bb) /ce/ ~| ~.

V (Aa5(b,c)<<P1.7>>: INVALID INDEX REPLACEMENT OPTION-->
,Bb,Cc) /ce/ ~| Different a5 from U's. ~.

VVv,w (Aa6(w)<<P1.8>>: INVALID INDEX REPLACEMENT
OPTION-->) /ce/ ~| Independent index in a6() not first among
aliases for VV's indices. ~.

GNAME1g (Ga<<P1.9>>: INVALID INDEX REPLACEMENT OPTION-->
) /ce/ ~| (a unreplaced). ~.

GNAME2e (G.<b+1><<P1.9>>: INVALID INDEX REPLACEMENT
OPTION-->) /ce/ ~| (Option c used). ~.

GNAME3 (Gab) /ce/ ~| OK ~.

GNAME4 (Gc<<P1.10>>: INVALID INDEX REPLACEMENT OPTION-->
b) /ce/ ~| Violates index replacement. ~.

GNAME6 (G<2:b><<P1.10>>: INVALID INDEX REPLACEMENT
OPTION-->.) /ce/ ~| Violates index replacement. ~.

GNAME7 (G.b.<<P1.10>>: INVALID INDEX REPLACEMENT
OPTION-->) /ce/ ~| Too many replacements. ~.

T (Ab1(c)<<P1.10>>: INVALID INDEX REPLACEMENT OPTION-->
,Cc) /ce/ ~| Illegal functional dep name (A's index is not
b). Option d used incorrectly. ~.

GNAME8 (G.<<P1.10>>: INVALID INDEX REPLACEMENT OPTION-->
) /ce/ ~| One G--index unaccounted for. ~.

X (Cc1(a),A) /ce/ Filter (a>1) {A} ~| ~.

Y (Cc1(a),A<<P1.11>>: INVALID INDEX REPLACEMENT OPTION-->
) /ce/ Filter (a>2) {A} ~| Need to call X, but not <<P1.12>>. ~.

Z1(Aa) /a/ : "C" <= String 5 <= "A"<<P2.4>>: INVALID
SUBRANGE LIMITS--> ~| ~.

Z2(Aa) /a/ : 30 <= Int <= 5<<P2.4>>: INVALID SUBRANGE
LIMITS--> ~| ~.

```
Z3(Aa) /a/ : 3.0 <= Real <= .5<--ERROR <<P2.4>>: INVALID
SUBRANGE LIMITS--> ~| ~.
~.~.
```

6.2.3 Example of Semantic Restrictions in Index Set Statements

The following model contains violations of semantic restrictions <<P3.1>>-
<<P3.26>>. The error messages are shown in uppercase.

```
A1 /pe/ Size {A1} <= 5<--ERROR <<P3.1>>: INVALID INDEX SET
STATEMENT--> ~| A1 is unindexed, not self--indexed. ~.
```

```
A2 /pe/ Select <--ERROR <<P3.1>>: INVALID INDEX SET
STATEMENT--> ~| A1 is unindexed, not externally--indexed. ~.
```

```
A3d /pe/ Select <--ERROR <<P3.1>>: INVALID INDEX SET
STATEMENT--> ~| A3 is self--indexed, not externally--indexed. ~.
```

```
A4a /pe/ 1<--ERROR <<P3.1>>: INVALID INDEX SET STATEMENT-->
~| A4 is self--indexed, not unindexed (a singleton). ~.
```

```
A5 (A4a) /ce/ 10 <= Size{A5}<--ERROR <<P3.1>>: INVALID INDEX SET
STATEMENT--> ~| A5 is externally indexed, not self--indexed. ~.
```

```
A6 (A4a) /ce/ 1<--ERROR <<P3.1>>: INVALID INDEX SET STATEMENT-->
~| A6 is externally indexed, not unindexed. ~.
```

```
B1b /pe/ Size{A1}<--ERROR <<P3.2>>: INVALID GENUS NAME--> <= 10
~| A1 is not the current genus. ~.
```

```
B2u /pe/ 1 <= Size{B2} <= 10 ~| P and Q must be
positive integers. ~.
```

```
Cc /pe/ 10 <= Size{C} <= 1<--ERROR <<P3.3>>: INVALID SIZE LIMITS-->
~| P must be less than or equal to Q. ~.
```

```
D (A4a) /ce/ {A6}<--ERROR <<P3.4>>: INVALID GENUS CALL-->
~| {A6} is not called directly or indirectly by current genus. ~.
```

E1 (Cc1(a)) /ce/ ~| ~.

E2 (A4a) /ce/ Project (a) {c1} <--ERROR <<P3.5>>: INVALID
DEPENDENCY CALL-->~| c1 is not defined by a
genus that is called directly or indirectly by the current genus. ~.

F (B1b,Cc) /ce/ Project (b)({B1}x{C}) <--ERROR <<P3.6>>:
INCOMPATIBLE INDEX SET STATEMENT-->~| The relation
algebra expression of index set statement must be compatible with
the generic index tuple of the current genus. ~.

G g, z, y /pe/ ~| ~.

GG (Gz) /ce/ ~| ~.

G1 (Cc, Gg) /ce/ ({C} Union {G}<--ERROR <<P3.7>>: DIFFERENT
SYMBOLIC INDICES-->) x {G} ~|
The union operator must apply to two generalized index sets
with identical symbolic indices. ~.

G2 (Gg, GGz) /ce/ ({G} Union {GG}) x {GG} ~| The union
operator must apply to two generalized index sets with identical
symbolic indices (alias not considered distinct).
This test must pass. ~.

G3 (Cc, Gg) /ce/ ({C} Minus {G}<--ERROR <<P3.7>>: DIFFERENT
SYMBOLIC INDICES-->) x {G} ~|
The minus operator must apply to two generalized index sets
with identical symbolic indices. ~.

G4 (Cc, Gg) /ce/ ({C} Intersect {G}<--ERROR <<P3.7>>: DIFFERENT
SYMBOLIC INDICES-->) x {G} ~| The Intersect operator must apply
to two generalized index sets with identical symbolic indices. ~.

H (Gg, GGz, Gg1(c)) /ce/ ~| ~.

H1 (Cc,Gg) /ce/ (Project (c) {G}<--ERROR <<P3.8>>: ILLEGAL SYMBOLIC
INDICES-->) x {G} ~|
c is not a subset of the symbolic indices of the operand {G}. ~.

H2 (Cc) /ce/ Project(c) (Project (c,c) {C}<--ERROR <<P3.8>>:
ILLEGAL SYMBOLIC INDICES-->)
~| (c, c) are not a distinct subset of the symbolic indices of

the operand {C}. ~.

I (A4a, Cc) /ce/ {A4} * {C} <--ERROR <<P3.9>>: MISSING IDENTICAL SYMBOLIC INDEX--> ~| The natural Join operator must apply to two generalized index set statements with 1 or more identical symbolic indices. ~.

J (Cc, Gg) /ce/ (Filter (g = -1) <--ERROR <<P3.10>>: INDEX NOT IN OPERAND-->{C})x{G} ~| Index g does not appear in the operand. ~.

K (Cc, Gg) /ce/ (Filter (g1 <--ERROR <<P3.11>>: UNREACHABLE FUNCTIONAL DEPENDENCY-->(c) > 1) {g1} <--ERROR <<P3.5>>: INVALID DEPENDENCY CALL-->) ~|

Functional dependency g1 is not defined in the the current genus or in one that reaches it. ~.

L (B1b, Cc, Gg2(c)) /ce/ (Filter (g2(b <--ERROR <<P3.12>>: INCORRECT NUMBER OF SYMBOLIC INDICES-->) > 1) {B1}) x {C} ~|

Use of functional dependency g2 does not match its definition arguments. ~.

M (Lbc) /ce/ Filter ((b + c) <--ERROR <<P3.13a>>: INDICES NOT ALIASES OF ONE ANOTHER-->= 1) {L} ~|

b and c and not aliases of one another. ~.

N (Gz) /ce/ Select Where g <--ERROR <<P3.14>>: INDEX NOT IN GENERIC INDEX TUPLE OF CURRENT GENUS--> Covers {G}

~| Index 'g' should be from the generic index tuple (should be a 'z'). ~.

O (Gz) /ce/ Select Where z Covers {N} <--ERROR <<P3.15>>: INVALID GENUS CALL--> ~| N is not directly or indirectly called by O. ~.

P (Gg, B1b) /ce/ Select Where (g, b) <--ERROR <<P3.16>>: INDICES NOT DISTINCT ALIASES--> Reflexive

~| Reflexive must apply to a single domain ('g' and 'b' are not from the same domain). ~.

Q (Gg, Gz, Gy) /ce/ Select Where Reflexive <--ERROR <<P3.17>>: ILLEGAL USE OF QUALIFIER-->~|

The short version can be used only when the generic index tuple of the current genus has exactly 2 indices in all and they are alias of one another. (we have three indices in this example). ~.

T (Gg, Cc) /ce/ Select Where g<--ERROR <<P3.20>>: INDEX NOT IN
GENERIC INDEX TUPLE OF GENUS--> Covers {C} ~| g is in the
generic index tuple of T, but not C. ~.

U (Gg, B1b) /ce/ ~| ~.

U1 (Ugb) /ce/ Select Where (g, g) <--ERROR <<P3.21>>: INDICES MUST
BE DISTINCT-->Covers {U}
~| Multiple indices must be distinct. ~.

V1 (Gg, Gz, Gy) /ce/ Where (g,z) <--ERROR <<P3.22>>: ILLEGAL USE OF
QUALIFIER-->Reflexive
~| In the absence of a Select prefix, option Reflexive may be used
only if the generic index tuple of the current genus has exactly 2
indices and these are aliases of one another. ~.

V1A (Gg, Gz, Gy) /ce/ Where (g,z) <--ERROR <<P3.22>>:
ILLEGAL USE OF QUALIFIER-->Symmetric
~| In the absence of a Select prefix, option Symmetric may be used
only if the generic index tuple of the current genus has exactly 2
indices and these are aliases of one another. ~.

V1B (Gg, Gz, Gy) /ce/ Where (g,z) <--ERROR <<P3.22>>:
ILLEGAL USE OF QUALIFIER-->Transitive
~| In the absence of a Select prefix, option Transitive may be used
only if the generic index tuple of the current genus has
exactly 2 indices and these are aliases of one another. ~.

U2 (Gg) /ce/ Where (g,g) <--ERROR <<P3.16>>: INDICES NOT DISTINCT
ALIASES--><--ERROR <<P3.23>>: ILLEGAL USE OF QUALIFIER OPTION-->
Irreflexive
~| In the absence of a Select prefix, option Irreflexive may be used
only if the generic index tuple of the current genus has at
least 2 indices. ~.

W (Gg) /ce/ Where (g,g) Asymmetric <--ERROR <<P3.24>>:
ILLEGAL USE OF QUALIFIER OPTION-->~| Options 'Asymmetric',
'Antisymmetric', and 'Covers' need a Select prefix. ~.

X (Gg, Gz) /ce/ Where Reflexive, Irreflexive <--ERROR <<P3.25>>:
ILLEGAL USE OF QUALIFIER OPTION--> ~|
'Extension' should not be used with 'Restriction'. ~.

Y1 (Gg, Gz) /ce/ Select Where Reflexive, Irreflexive <--ERROR
<<P3.26>>: ILLEGAL USE OF QUALIFIER OPTION-->~|

In the presence of a Select prefix, the two options here are not allowed to appear at the same time. ~.

Y2 (Gg, Gz) /ce/ Select Where Symmetric, Asymmetric <--ERROR
<<P3.26>>: ILLEGAL USE OF QUALIFIER OPTION-->~| In

the presence of a Select prefix, the two options here are not allowed to appear at the same time. ~.

Y3 (Gg, Gz) /ce/ Select Where Reflexive, Asymmetric <--ERROR
<<P3.26>>: ILLEGAL USE OF QUALIFIER OPTION-->~| In

the presence of a Select prefix, the two options here are not allowed to appear at the same time. ~.

Y4 (Gg, Gz) /ce/ Select Where Symmetric, Transitive, Irreflexive
<--ERROR <<P3.26>>: ILLEGAL USE OF QUALIFIER OPTION-->

~| In the presence of a Select prefix, the options here are not allowed to appear at the same time. ~.

Y5 (Gg, Gz) /ce/ Select Where Symmetric, Irreflexive,
Antisymmetric <--ERROR <<P3.26>>: ILLEGAL USE OF QUALIFIER OPTION-->

~| In the presence of a Select prefix, the options here are not allowed to appear at the same time. ~.
~.~.

6.2.4 Example of Semantic Restrictions in Generic Rules

The following model contains violations of semantic restrictions <<P4.1>>-
<<P4.34>>. The error messages are shown in uppercase.

&PREP_SRC ~| ~/PREPARATORY SOURCE~/ ~.

Pi,j /pe/ ~| ~.

Rk /pe/ ~| ~.

S (Pi) /a/ : Real+ ~| ~.

T (Pi,Pj) /a/ : Real+ ~| ~.

\$1(T) /f/ ; @SUMi SUMj (Tij) ~| ~.

&TESTS ~| ~.

TEST2 (Si) /f/ ; %A, Where %A Is Logical <--ERROR <<P4.2>>:
GENERIC RULE MUST BE NUMERIC VALUED--> ~| Rule must be a
numeric-valued expression. ~.

OK3 (Si) /f/ ; i ~| ~.

TEST3 (Si) /f/ ; j<--ERROR <<P4.3>>: ILLEGAL USE OF LOCAL
INDEX--> ~| The ordinate function instance may not be
renamed and it must be among the indices the current genus'
generic index tuple. ~.

TEST3_ALT (Si) /f/ ; i'<--ERROR <<P4.3>>: ILLEGAL USE OF
LOCAL INDEX--> ~| The ordinate function instance may not
be renamed and it must be among the indices the current genus'
generic index tuple. ~.

OK4 (Si) /f/ ; @SUMi' (i') ~| The index i' must be a local
index or a not renamed index among the indices of the current
genus' generic index tuple. ~.

OK4_ALT (Si) /f/ ; @SUMi' (i) ~| The index i must be a local
index or a not renamed index among the indices of the current
genus' generic index tuple. ~.

TEST4 (Si) /f/ ; @SUMi' (j'<--ERROR <<P4.4>>: ILLEGAL
USE OF LOCAL INDEX-->) ~| The index j' must be a local index or a
not renamed index among the indices of the current genus'
generic index tuple. ~.

OK5 (Si1(j)) /f/ ; @IPRODi (i1(i)) ~| The independent
index (i) is identical to that associated with the definition
of the functional dependency. ~.

TEST5 (Si2(j)) /f/ ; @IPRODi (i2(i,i) <--ERROR <<P4.5>>,
<<P4.10A>>: ILLEGAL USE OF INDEPENDENT INDEX-->) ~| The
independent indices (i,i) are different from those
associated with the definition of the functional dependency. ~.

TEST6 (Si3(j)) /f/ ; @IPRODi (i1(i) <--ERROR <<P4.6>>, <<P4.10B>>: FUNCTIONAL DEPENDENCY IS UNREACHABLE-->) ~| The functional dependency i1 is not defined in the calling sequence of the current genus or one that reaches it. ~.

TEST7 (\$1,T<1>j) /f/ ; T<1>j + \$1j<--ERROR <<P4.7>>: ILLEGAL USE OF INDEPENDENT INDEX--> + j ~| The genus name \$1 is unindexed; thus it must have no indices. ~.

TEST8 (Sj,Pj,Rk,Tij) /f/ ; Tii + Sk<--ERROR <<P4.8>>: ILLEGAL USE OF INDEPENDENT INDEX--> + j ~| The simple variable Sk must have correctly replaced indices. ~.

TEST9 (Pj) /f/ ; Sj<--ERROR <<P4.9>>: INVALID GENUS CALL--> ~| The named genus is unreachable. ~.

TEST9A (Pj) /f/ ; Pj<--ERROR <<P4.9A>>: INVALID GENUS CALL--> ~| The named genus is not of type /a/ or /f/ or /t/. ~.

OK2_OLD_10 (S<i+1>) /f/ ; i + Si ~| ~.

TEST10A (Si8(j)) /f/ ; Si8(j,j) <--ERROR <<P4.5>>,<<P4.10A>>: ILLEGAL USE OF INDEPENDENT INDEX--> ~| The independent indices (j,j) are different from those associated with the definition of the functional dependency. ~.

TEST10B (Si5(j)) /f/ ; Si4(j) <--ERROR <<P4.6>>,<<P4.10B>>: FUNCTIONAL DEPENDENCY IS UNREACHABLE--> ~| The functional dependency i4 is not defined in the calling sequence of the current genus or one that reaches it. ~.

TEST10B_OK (OK5,Pj) /f/ ; Si1(j) ~| ~.

A (Si6(i)) /f/ ; Si6(i) ~| The functional dependency i6 is defined in the calling sequence of the current genus. ~.

B (Pj) /a/ ~| ~.

TEST10C (Bj,Ai) /f/ ; Bi6(i) <--ERROR <<P4.10C>>: ILLEGAL USE OF DEPENDENT INDEX--> ~| The correct index replacement for B is j, not i. ~.

TEST11 (Sj) /f/ ; Si<--ERROR <<P4.11>>: ILLEGAL USE OF LOCAL INDEX--> ~| The free index i is not in the current generic index tuple. ~.

TEST11_ALT (Sj) /f/ ; Sj'<--ERROR <<P4.11>>: ILLEGAL USE OF LOCAL INDEX--> ~| The free index j' should not be renamed. ~.

TEST12 (Sj) /f/ ; @SUMj'<1:2> (Si<--ERROR <<P4.12>>: ILLEGAL USE OF LOCAL INDEX-->) ~| The free index i is not controlled either by the current generic index tuple or by a local index. ~.

TEST12_ALT (Si,Sj) /f/ ; @SUMj'<1:2> (Si'<--ERROR <<P4.12>>: ILLEGAL USE OF LOCAL INDEX-->) ~| The free index i' is not controlled either by the current generic index tuple or by a local index. ~.

TEST13 (Sj) /f/ ; @SUMj'<1:2> (Sj') + %Ai<--ERROR <<P4.13>>: INDEX NOT IN GENERIC INDEX TUPLE--> + Sj ~| The free index i is not in the current generic index tuple. ~.

TEST14 (Sj,Si) /f/ ; @SUMj'<1:2> (Sj') + %Ai + %Aj<--ERROR <<P4.14>>: SYMBOLIC PARAMETER INDEX MISMATCH--> ~| The symbolic parameter uses different free indices. ~.

TEST15 (S) /f/ ; -"#TRUE"<--ERROR <<P4.15>>: OPERAND MUST BE NUMERIC VALUED--><--ERROR <<P4.2>>: GENERIC RULE MUST BE NUMERIC VALUED--> ~| The operand of the unary minus must be numeric valued. ~.

TEST16 (Si) /f/ ; Si - "STRING"<--ERROR <<P4.16>>: OPERAND MUST BE NUMERIC VALUED--><--ERROR <<P4.2>>: GENERIC RULE MUST BE NUMERIC VALUED--> ~| Both operands of the binary minus must be numeric valued. ~.

TEST17 (Si) /f/ ; @ABS("STRING"<--ERROR <<P4.17>>: ARGUMENT MUST BE NUMERIC VALUED-->) ~| The arguments of built-in functions must be numeric valued. ~.

TEST18 (Si) /f/ ; @ABS(-3, -5) <--ERROR <<P4.18>>: INCORRECT NUMBER OF ARGUMENTS--> ~| The number of arguments of built-in functions must be respected. ~.

TEST20 (Tij) /f/ ; @SUMi'IPRODj'<--ERROR <<P4.20>>: INCORRECT INDEX SUPPORTING FUNCTION NAME--> (Ti'j') + Tij ~| Iterated index supporting function names must match. ~.

TEST21 (Si) /f/ ; @SUMi' ("STRING"<--ERROR <<P4.21>>: ARGUMENT MUST BE NUMERIC VALUED-->) + Si ~| The argument of the index supporting function must be numeric-valued. ~.

TEST22 (Si) /f/ ; @SUMi<--ERROR <<P4.22>>: SYMBOLIC INDEX MUST BE RENAMED--> (5) + Si ~| The local index 'i' is in the generic index tuple; thus it must be renamed. ~.

TEST23 (Si) /f/ ; @SUMk<--ERROR <<P4.23>>: ILLEGAL USE OF LOCAL INDEX--> (5) + Si ~| The local index 'k' is not among the indices of a reaching genus. ~.

TEST24 (Si,Rk) /f/ ; @SUMi' <k<--ERROR <<P4.24>>: ILLEGAL USE OF SYMBOLIC INDEX-->-1:4> (5) + Si ~| The limit index 'k' is not in the same domain as the local index i'. ~.

OK_24 (Si,Rk) /f/ ; @SUMi' <i-1:4> (5) + Si ~| ~.

TEST25 (Si) /f/ ; @SUMi' <i'<--ERROR <<P4.25>>: ILLEGAL USE OF SYMBOLIC INDEX--><--ERROR <<P4.26>>: ILLEGAL USE OF SYMBOLIC INDEX-->:4> (5) + Si ~| The limit index "i'" should not be identical to the local index i'. ~.

TEST26 (Si) /f/ ; @SUMi' <j<--ERROR <<P4.26>>: ILLEGAL USE OF SYMBOLIC INDEX-->-1:4> (5) + Si ~| The limit index 'j' should not be free. ~.

TEST27 (Si) /f/ ; @SUMi' <5:4> <--ERROR <<P4.27>>: ILLEGAL LOWER/UPPER LIMITS-->(5) + Si ~| The lower limit should be less than or equal the upper limit. ~.

TEST28 (Si) /f/ ; @SUMi' <i+4:4> <--ERROR <<P4.28>>: ILLEGAL LOWER/UPPER LIMITS-->(5) + Si ~| (1+4) should be less than or equal the upper limit. ~.

TEST29 (Si) /f/ ; @SUMi' <i+5:i+4> <--ERROR <<P4.29>>: ILLEGAL LOWER/UPPER LIMITS-->(5) + Si ~| The lower limit (5) should be less than or equal the upper limit (4). ~.

TEST30 (Si) /f/ ; @SUMi' <i-4:i-5> <--ERROR <<P4.30>>:
ILLEGAL LOWER/UPPER LIMITS-->(5) + Si ~| The lower limit
(-4) should be less than or equal the upper limit (-5). ~.

TEST31 (Si) /t/ ; Si = "#TRUE"<--ERROR <<P4.31>>: OPERAND
TYPE MISMATCH--> ~| The operands of the relational
operator must be of the same type. ~.

TEST31A (Pi) /f/ ; @EXIST(Pi, "#TRUE", 0) <--ERROR
<<P4.31A>>: BOTH ARGUMENTS MUST BE NUMERIC OR STRING VALUED-->
<--ERROR <<P4.2>>: GENERIC RULE MUST BE NUMERIC VALUED--> ~| The
second and third operands of the @EXIST must both be
numeric-valued expressions here (since it is a /f/ genus). ~.

TEST32 (Si) /f/ ; @IF(Si > 0 , "CAT" , "HOUSE")
<--ERROR <<P4.32>>: BOTH ARGUMENTS MUST BE NUMERIC VALUED-->
<--ERROR <<P4.2>>: GENERIC RULE MUST BE NUMERIC VALUED--> ~| The
second and third operands of the @IF must be numeric-valued
expressions here (since it is a /f/ genus). ~.

OK_32 (Si) /t/ ; @IF(Si > 0 , #TRUE, #FALSE) ~| ~.

TEST33 (Si) /f/ ; Si , Where %A Is Real <--ERROR
<<P4.33>>: SYMBOLIC PARAMETER NOT IN GENERIC RULE--> ~| Symbolic
parameter is not used in rule. ~.

TEST34 (Si) /f/ ; Si + %Ai, Where %A Is 1<= Real <= -1
<--ERROR <<P4.34>>: INVALID SUBRANGE LIMITS--> ~| Subranges
require that lo <= hi. ~.

~.~.

CHAPTER 7

Conclusions

This dissertation proposed adopting an attribute grammar formalism to rigorously formalize mathematical modeling languages used by the Operations Research/Management Science (OR/MS) community, and to automatically generate computer-based modeling environment tools. This proposition was formulated in terms of a main claim, stated at the outset of Chapter 1.

Via the studies that we undertook of five of the hypotheses designed to test the main claim, we have shown its feasibility. Thus, we can conclude that a semantic formalization based on the attribute grammar framework can and should be considered by those who design modeling languages and major components of OR/MS modeling environments. The adoption of the semantic formalization paradigm suggested here would help to make the modeling activity in OR/MS less error-prone and more effective, thereby lessening the time required to develop correct new models and maintain existing ones.

Although our evidence and experience does not demonstrate that the attribute grammar formalism can be used to implement all aspects of a modeling environment, it suggests that it can be used to implement tools beyond those of syntax-directed editors, inferencing tools, and code generators. A new and very important

application to the area of mathematical optimization has been already described in Section 3.7. The area of information retrieval is also explored in Section 7.3.1. In general, activities of model-analysis (like expression evaluation, query facilities, and optimization) appear the most suitable to this formalization approach.

Finally, a word is appropriate with regard to the material presented in Chapter 2. On its own, that chapter stands as a source of motivation for the rest of this work. However, we now suggest (based on experience and without further proof) that it should be possible to employ the methodologies and techniques shown in Chapter 3 through 6 to formalize the semantics of AMPL, GAMS, and LINGO, and to generate tools for those languages.

The balance of this chapter is structured as follows. Section 7.1 provides a summary of the most important contributions made by this dissertation. Section 7.2 then lists some of the limitations of this work. Finally, Section 7.3 concludes with a description of areas for future research activity.

7.1 Summary of Contributions

Recognized and documented context-sensitive semantics in modeling languages for OR/MS. Several important languages for mathematical programming, namely AMPL, GAMS, and LINGO, are described from the new perspective of explicitly identifying static semantics. Chapter 2 exposes the prevalent lack of formalization of context-sensitive semantics among such languages. It provides evidence that this absence of formalization makes the modeling activity more error-

prone; thus models written in these languages are more difficult to develop and maintain.

Pioneered the application of an attribute grammar methodology to describe the full static semantics of modeling languages for OR/MS. Chapter 3 shows that it is feasible to provide the complete formal description of the static semantics of even a complex modeling language like SML via attribute grammars. We developed an attribute grammar to include the full semantics of data and schema in SML, and a similar methodology may be adopted for other modeling languages like AMPL, GAMS, or LINGO. The attribute grammar description of SML helped to clarify the semantics of difficult areas, increasing the understanding of the language while it was being designed.

Adapted the attribute grammar methodology to perform inferencing in modeling languages for OR/MS. Chapter 4 shows that it is feasible to perform type checking in mathematical modeling languages by inferencing missing information via attribute grammar equations. Apart from providing a useful check, our approach to inferencing missing schema information in SML provides economy to the schema description. A variation of our type checking approach can be used to automatically deduce other constructs, thereby allowing other mathematical modeling languages to be simplified. These languages should have certain structures such that those conclusions which are reached about a model from information that is explicitly declared might also be implicitly reached from an analysis of references.

Initiated research into how to apply the attribute grammar methodology for code generation in modeling languages for OR/MS. Chapter 5 builds a tool that generates code which can be later executed to evaluate generic rules in SML. This generic approach is a first step necessary to support the evaluation of expressions in immediate mode in any mathematical modeling language where evaluation is a well defined task.

Built the first syntax-directed editor in a modeling environment using the attribute grammar approach. A modeling environment tool has been built that prevents both syntactic and semantic errors in SML models. Chapter 6 demonstrates how our syntax-directed editor for SML models can be generated automatically. Special attention was paid to presenting error messages in a format that is clear and useful, and to testing faulty models.

Debugged a large collection of models using the methodology. The syntactic and semantic correctness of over 100 different models described in the Library of Structured Models [Geof89e] and elsewhere has been asserted using the syntax-directed editor in the environment developed for this dissertation. Feedback on each model's syntactic and semantic errors is made immediately available, thus making our system a valuable tool for model formulation and testing.

7.2 Limitations

The attribute grammar methodology has proven to be quite powerful for representing static semantics of SML Schemas (the Schema Properties). We have shown

that data properties (the Table Content Rules) can also be represented in the attribute grammar framework, but not without difficulties. The primary difficulty lies in the fact that, in the attribute grammar representation, elements are kept as values of attributes, and several of them need to be attached to the nonterminal(s) generating the element. This representation is not a very efficient one since any (minor) update to a datum requires updating all the attributes of all the affected attribute equations. An abundance of attributes which ultimately will not change their values, together with other non-local subtree constructs, need to be considered before the equations can be updated properly. Hence the attribute evaluator will perform unnecessary work. It follows that the attribute grammar approach is more appropriate for maintaining correct schemas with schema-directed editors, than for maintaining valid element data with data-directed editors.

In our prototype implementation of a semantics-driven modeling environment, we worked with the full SML language as given in [Geof88]. The only (rather minor) difference is in the area of keywords. Whereas SML recognizes both upper/lower and fully lower case spellings for keywords, our implementation only admits the upper/lower case combination.

In our prototype, syntax errors need to be fixed immediately before the model can be processed further. A required repair action cannot be delayed by the user, since the system will continuously point to the syntax error and wait for a fix. This behavior is rather annoying for some users who prefer to have their model parsed all at once, with erroneous input discarded temporarily if necessary. Our prototype

would require a different approach to parsing and error recovery to satisfy these users.

Another limitation of our implementation approach is its performance. The main problem with attribute grammars is that they consume large amounts of storage, as has been noted by others [Reps83]. Computers with large amounts of memory are required to handle large models (i.e., those with hundreds of paragraphs). Alternatively, storage improvements can be achieved at the expense of speed. The attribute evaluators can be modified to consume less storage, but then they will necessarily run slower because they have to perform more algorithmic steps [Deme81].

7.3 Further Research

This dissertation has uncovered various opportunities for further research on the design and application of attribute grammars to modeling environments. Three of the most important areas are now described.

7.3.1 Answering Queries about Model and Data

A powerful modeling environment should be able to provide some reasoning capabilities on the objects which it supports. In a modeling environment for SML models, it would be desirable to be able to reason about the structured models themselves.

For models written in SML, it is natural to pose queries such as these about a

model Schema:

- Which genus calls, or is called by, a given genus?
- Which genus introduces a particular symbolic index?
- Where is a certain symbolic index used?
- Where is a particular genus introduced?

These queries involve the retrieval of information about the Schema. Similar queries could be considered about the data contained in Elemental Detail Tables. The approaches for answering the queries about the Schema and the queries about the element-data would need to be integrated in a uniform way in the modeling environment.

Because a language-based editing environment does maintain (implicitly if not explicitly) all this information about the models being created, it should be possible to provide capabilities for answering these queries on the model. Provision for this kind of capability would greatly enhance the modeling environment.

To support the kinds of Schema queries described above, the pure attribute-grammar formalism on which the Synthesizer Generator is based needs to be expanded to include tables, or relations (which are distinct from Elemental Detail Tables). Tables would serve to aggregate information about the Schema, and the answer to queries would also be tables. The model Schema would then be represented as an attributed abstract syntax tree with an associated database.

The merging of a relational database model and attributes in the Synthesizer system has been partially explored in [Horw85] [Horw86]. In that scheme, attributes participate as tuples in several relations. The tuples are declaratively defined on a per-production basis. The tuples can depend on values of attributes and, conversely, attributes can depend on information contained in relations. Computations which cannot be carried out on the attributes can be carried in the relations. The relations are kept consistent by the normal attribution scheme. Queries can be made against the relations using an SQL-like language and a specially designed editor.

The symbiosis of attributes and tables in the context of an attribute-grammar based modeling environment should be further investigated. The product of this aspect of the research should be a design that extends the modeling environment with attributed tables, and answers queries like those mentioned above. The interface of attributes and tables could be at an internal level, as in Horwitz, or perhaps at an external level, with some commercial database system. Some query language would be provided to answer the queries.

The following are some of the topics that need to be covered in this design:

- The editing style. One of the goals of a modeling environment is that it be built using language-independent features. A “generic” editing style is needed for querying the Schema.
- The data in tables. It is necessary to decide which kind of data will be placed

in the tables: either static or dynamic data. A close look should be given to the data in the Schema, and to typical modeling operations.

- The environment definition language. To customize the extended environment, some language needs to be provided. This language should be declarative in nature, not procedural.
- The kinds of query evaluation mechanisms that should be provided. Is a database kept in Prolog more adequate?
- Would operations on views (tables derived from other tables) be allowed? If so, how should the currency of these views be maintained?
- The database operators. If the relational model is chosen, queries would be expressed in a relational algebra. In that case, the basic relational operators need to be provided.

7.3.2 Support for a Complete Modeling Environment

The integration of various kinds of “modeling in the large” tools, including configuration managers and version control systems, with the existing “modeling in the small” tools, like semantics-driven editors and immediate evaluators, would produce a complete environment for modeling. An example of integration of tools in the programming language context can be found in UNIX, which provides a number of tools to assist the design, coding, verification, and maintenance of individual programs.

The development of a complete and consistent model development system would make for a very advanced modeling environment. The role that attribute grammars may offer as a formal framework for tool integration is an open question, and more research is warranted in this area. The first step required to extend the system in this direction is to evaluate the peculiar demands of model management, which may be different from those of programming.

7.3.3 Full Support for Immediate Evaluation

We developed a tool to support the code generation phase of expression evaluation. More research is needed in the area of the actual evaluation of the generated code. To support this task, improved store and manipulation facilities for model data are required. Sophisticated data handling facilities can permit the efficient performance that is required to obtain truly “immediate” evaluation. A possible approach is to use a commercial relational database management system, such as ORACLE, to store model data, and to use our attribute grammar approach to generate special function calls to the database system. The code structure shown in Chapter 5 would be maintained since it is generic in nature, but it would need to be specialized to the details of the particular database management system utilized.

Also, special attention should be paid to providing interfaces that are as user-friendly as those found in current systems that perform immediate evaluation. Good interfaces are provided, for example, in spreadsheets like 1-2-3, and in sci-

entific software packages like Mathematica, Gauss, and MathCAD.

APPENDIX A

An Abstract Syntax for SML

```
root schemafile;
schemafile:
    SchemaFileList(schema_par EOF opt_element_detail)
;
optional list schema_par;
schema_par:
    SchemaNil()
  | SchemaPair(paragraph schema_par)
;
optional paragraph ;
paragraph:
    SubschemaNil()
  | SubschemaPrompt()
  | SubschemaModPar(indent_or_null modparagraph)
  | SubschemaGenPar(indent_or_null genparagraph)
;
optional indent_or_null;
indent_or_null:
    IndentOrNullEmpty()
  | IndentOrNullPrompt()
  | IndentOrNullNonEmpty(BLANKSPACE)
;
modparagraph:
    ModParNull()
  | ModParNonNull(modname TBAR interp TPERIOD)
;
modname:
    ModNameNull()
  | ModNameNonNull(MNAME)
;
list interp;
interp:
    InterpEmpty()
  | InterpPair(interp_line interp)
```

```

;
interp_line:
    InterpNilKeyPhrase()
    | InterpNonKeyPhrase(NON_KEY_PHRASE)
    | InterpDefKeyPhrase(DEF_KEY_PHRASE)
    | InterpRefKeyPhrase(REF_KEY_PHRASE)
;
genparagraph:
    GenParNull()
    | PePar(peparagraph)
    | CePar(ceparagraph)
    | AVaPar(avapagraph)
    | FPar(fparagraph)
    | TPar(tparagraph)
;
peparagraph:
    PeParNull()
    | PeNode(gname opindices PE_TYPE_DECL
             op_iss op_dom_stat TBAR interp TPERIOD)
;
ceparagraph:
    CeParNull()
    | CeNode(gname opindices LPAREN calls RPAREN CE_TYPE_DECL
             op_iss op_dom_stat TBAR interp TPERIOD)
;
avapagraph:
    AVaParNull()
    | AVaNode(gname opindices LPAREN calls RPAREN A_VA_TYPE_DECL
              op_iss op_dom_stat op_range_stat TBAR interp TPERIOD)
;
fparagraph:
    FParNull()
    | FNode(gname opindices LPAREN calls RPAREN F_TYPE_DECL
            op_iss op_dom_stat SEMICOLON modfunexpr TBAR interp TPERIOD)
;
tparagraph:
    TParNull()
    | TNode(gname opindices LPAREN calls RPAREN T_TYPE_DECL
            op_iss op_dom_stat SEMICOLON modtstexpr TBAR interp TPERIOD)
;
gname:
    GenNameNull()
    | GenNameNonNull(GNAME)

```

```

;
optional opindices;
opindices:
    OpIndNull()
    | OpIndPrompt()
    | OpIndNonNull(delimindices)
;
delimindices:
    DelimNull()
    | IndexLetter(index)
    | OpIndices(delimindices COMMA index)
;
calls:
    CallComps(components)
;
components:
    CompSingle(component)
    | CompPair(component COMMA components)
;
component:
    CompSimple(simplecomp)
    | CompGeneral(generalcomp)
;
simplecomp:
    SimpleCompNull()
    | SimpleComp(GNAME)
;
generalcomp:
    GeneralCompNull()
    | GeneralComp(GNAME indexcells)
;
indexcells:
    IndexCellsUnique(cell)
    | IndexCellsPair(cell indexcells)
;
cell:
    CellOptionA(optiona)
    | CellOptionB(optionb)
    | CellOptionC(optionc)
    | CellOptionD(optiond)
    | CellOptionE(optione)
    | CellIndex(index)
;

```

```

optiona:
    OptionNonNullA(PERIOD)
;
optionb:
    OptionNonNullB1(ANG_BR_PINT optsign pinteger GT)
    | OptionNonNullB2(LT optsign pinteger COLON optsign pinteger GT)
;
optionc:
    OptionNonNullC1_1(ANG_BR_INDEX_PINT index PLUS_SIGN pinteger GT)
    | OptionNonNullC1_2(ANG_BR_INDEX_PINT index MINUS_SIGN pinteger GT)
    | OptionNonNullC2(LT optsign pinteger COLON index GT)
    | OptionNonNullC3(LT optsign pinteger COLON index sign pinteger GT)
    | OptionNonNullC4(LT index COLON optsign pinteger GT)
    | OptionNonNullC5_1(LT index PLUS_SIGN pinteger COLON optsign
        pinteger GT)
    | OptionNonNullC5_2(LT index MINUS_SIGN pinteger COLON optsign
        pinteger GT)
    | OptionNonNullC6(LT index COLON index PLUS_SIGN pinteger GT)
    | OptionNonNullC7(LT index PLUS_SIGN pinteger COLON index PLUS_SIGN
        pinteger GT)
    | OptionNonNullC8(LT index MINUS_SIGN pinteger COLON index GT)
    | OptionNonNullC9(LT index MINUS_SIGN pinteger COLON index sign
        pinteger GT)
;
optiond:
    OptionNonNullD(fundepname LPAREN delimindices RPAREN)
;
optione:
    OptionNonNullE(mvdname LPAREN delimindices RPAREN)
;
fundepname:
    FuncDepName(index pinteger)
;
mvdname:
    MvDepNameNull()
    | MvDepName(index pinteger MULT_SIGN)
;
optsign:
    OptSignNull()
    | OptSignNonNull(sign)
;
sign:
    SignPlus(PLUS_SIGN)

```

```

    | SignMinus(MINUS_SIGN)
;
pinteger:
    One(STR)
    | Pinteger(P_INTEGER)
;
nzinteger:
    NzInteger1(pinteger)
    | NzInteger2(sign pinteger)
;
nninteger:
    NnInteger1(pinteger)
    | NnInteger2(STR)
;
integer:
    Integer1(nninteger)
    | Integer2(sign nninteger)
;
optional op_iss;
op_iss:
    OpIssNull()
    | OpIssPrompt()
    | OpUnindexedIss(unindexed_iss)
    | OpSelfIss(self_iss)
    | OpExternalIss(external_iss)
;
unindexed_iss:
    UnindexedIssNull()
    | UnindexedIssNonNull(STR)
;
self_iss:
    SelfIssNull()
    | SelfIss1(SIZE GENUS_INDEX_SET EQ pinteger)
    | SelfIss2(pinteger LE SIZE GENUS_INDEX_SET)
    | SelfIss3(SIZE GENUS_INDEX_SET LE pinteger)
    | SelfIss4(pinteger LE SIZE GENUS_INDEX_SET LE pinteger)
;
external_iss:
    ExternalIssNull()
    | ExternalIss1(SELECT)
    | ExternalIss2(rel_alg_expr)
    | ExternalIss3(qualifiers)
    | ExternalIss4(SELECT rel_alg_expr)

```

```

    | ExternalIss5(SELECT qualifiers)
    | ExternalIss6(SELECT rel_alg_expr qualifiers)
    | ExternalIss7(rel_alg_expr qualifiers)
;
rel_alg_expr:
    RelAlg1(unioning)
    | RelAlg2(difference)
    | RelAlg3(cartesian_product)
    | RelAlg4(intersection)
    | RelAlg5(natural_join)
    | RelAlg6(projection)
    | RelAlg7(filtering)
    | RelAlg8(index_set)
;
unioning:
    UnionPair(index_set UNION index_set)
;
difference:
    DifferencePair1(index_set MINUS index_set)
    | DifferencePair2(index_set MINUS_SIGN index_set)
;
cartesian_product:
    CartesianProductPair1(index_set TIMES index_set)
    | CartesianProductPair2(index_set C_P_TOKEN index_set)
;
intersection:
    IntersectionPair(index_set INTERSECT index_set)
;
natural_join:
    NaturalJoinPair1(index_set NJOIN index_set)
    | NaturalJoinPair2(index_set MULT_SIGN index_set)
;
projection:
    ProjectionPair(PROJECT LPAREN delimindices RPAREN index_set)
;
filtering:
    FilteringPair(FILTER LPAREN filter_formula RPAREN index_set)
;
filter_formula:
    FilterFormula1(proposition)
    | FilterFormula2(LPAREN filter_formula RPAREN)
    | FilterFormula3(AT_NOT LPAREN filter_formula RPAREN)
    | FilterFormula4(AT_AND LPAREN filter_formula COMMA

```

```

        filter_formula_list RPAREN)
    | FilterFormula5(AT_OR LPAREN filter_formula COMMA
        filter_formula_list RPAREN)
    | FilterFormula6(AT_IF LPAREN filter_formula COMMA
        filter_formula COMMA filter_formula RPAREN)
;
filter_formula_list:
    FilterFormulaList1(filter_formula)
    | FilterFormulaList2(filter_formula_list COMMA filter_formula)
;
proposition:
    PropositionPair(predicate_term relop predicate_term)
;
relop:
    RelopNull()
    | RelOpLt(LT)
    | RelOpLe(LE)
    | RelOpEq(EQ)
    | RelOpGt(GT)
    | RelOpGe(GE)
    | RelOpNe(NE)
;
predicate_term:
    PredicateTerm1(nzinteger)
    | PredicateTerm2(index)
    | PredicateTerm3(unren_func_dependency)
    | PredicateTerm4(LPAREN predicate_term RPAREN)
    | PredicateTerm5(LPAREN predicate_term sign predicate_term RPAREN)
;
index_set:
    IndexSet1(GENUS_INDEX_SET)
    | IndexSet2(LBRACE fundepname RBRACE)
    | IndexSet3(LBRACE mvdname RBRACE)
    | IndexSet4(LPAREN rel_alg_expr RPAREN)
;
unren_func_dependency:
    FunctionSymbolName(fundepname LPAREN delimindices RPAREN)
;
qualifiers:
    QualifierNull()
    | Qualifier1(WHERE qualifier_string)
    | Qualifier2(COMMA WHERE qualifier_string)
;

```

```

qualifier_string:
    QualifierStringNull()
    | QualifierString1(qualifier_phrase)
    | QualifierString2(qualifier_phrase COMMA qualifier_string)
;
qualifier_phrase:
    QualifierPhraseNull()
    | QualifierPhrase1(REFLEXIVE)
    | QualifierPhrase2(LPAREN delimindices RPAREN REFLEXIVE)
    | QualifierPhrase3(SYMMETRIC)
    | QualifierPhrase4(LPAREN delimindices RPAREN SYMMETRIC)
    | QualifierPhrase5(TRANSITIVE)
    | QualifierPhrase6(LPAREN delimindices RPAREN TRANSITIVE)
    | QualifierPhrase7(IRREFLEXIVE)
    | QualifierPhrase8(LPAREN delimindices RPAREN IRREFLEXIVE)
    | QualifierPhrase9(ASYMMETRIC)
    | QualifierPhrase10(LPAREN delimindices RPAREN ASYMMETRIC)
    | QualifierPhrase11(ANTISYMMETRIC)
    | QualifierPhrase12(LPAREN delimindices RPAREN ANTISYMMETRIC)
    | QualifierPhrase17(index COVERS GENUS_INDEX_SET)
    | QualifierPhrase18(LPAREN delimindices RPAREN COVERS GENUS_INDEX_SET)
;
optional op_dom_stat;
op_dom_stat:
    OptionalDomStatNull()
    | OptionalDomStatPrompt()
    | OptionalDomStatNonNull(domain_stat)
;
domain_stat:
    DomainStatNull()
    | DomainStatInteger(DOUBLE_COLON integer_options)
    | DomainStatString(DOUBLE_COLON string_options)
;
optional op_range_stat;
op_range_stat:
    OptionalRangeStatNull()
    | OptionalRangeStatPrompt()
    | OptionalRangeStatNonNull(range_stat)
;
range_stat:
    RangeStatNull()
    | RangeStatUnqual(COLON unqual_range_body)
    | RangeStatQual(COLON qual_range_body)

```



```

;
qual_range_body:
    QualRangeBodyNull()
    | QualRangeBodyString(string_type_range UNIQUE)
    | QualRangeBodyInteger(integer_type_range UNIQUE)
    | QualRangeBodyReal(real_type_range UNIQUE)
    | QualRangeBodyLogical(logical_type_range UNIQUE)
    | QualRangeBodyList(list_type_range UNIQUE)
    | QualRangeBodyUnique(UNIQUE)
;
unqual_range_body:
    UnqualRangeBodyNull()
    | UnqualRangeBodyString(string_type_range)
    | UnqualRangeBodyInteger(integer_type_range)
    | UnqualRangeBodyReal(real_type_range)
    | UnqualRangeBodyLogical(logical_type_range)
    | UnqualRangeBodyList(list_type_range)
;
string_type_range:
    StringTypeRangeNull()
    | StringTypeRangeSingle(single_string_range)
    | StringTypeRangeMultiple(multiple_string_range)
;
multiple_string_range:
    MultipleStringRangeNull()
    | MultipleStringRangeOption1(paren1_str_range OR paren1_str_range)
    | MultipleStringRangeOption2(multiple_string_range OR paren1_str_range)
;
paren1_str_range:
    Paren1StrRangeNull()
    | Paren1StrRangeNonNull(LPAREN single_string_range RPAREN)
;
single_string_range:
    SingleStringRangeNull()
    | SingleStringRangeOption1(string_options)
    | SingleStringRangeOption2(QUOTED_STRING ltle string_options)
    | SingleStringRangeOption3(string_options ltle QUOTED_STRING)
    | SingleStringRangeOption4(QUOTED_STRING ltle string_options ltle
        QUOTED_STRING)
;
string_options:
    StringOptionsNull()
    | StringOptions1(String)

```

```

    | StringOptions2(CHARACTER)
    | StringOptions3(STRING pinteger)
    | StringOptions4(CHARACTER pinteger)
;
integer_type_range:
    IntegerTypeRangeNull()
    | IntegerTypeRangeSingle(single_integer_range)
    | IntegerTypeRangeMultiple(multiple_integer_range)
;
multiple_integer_range:
    MultipleIntegerRangeNull()
    | MultipleIntegerRangeOption1(paren1_int_range OR paren1_int_range)
    | MultipleIntegerRangeOption2(multiple_integer_range OR paren1_int_range)
;
paren1_int_range:
    Paren1IntRangeNull()
    | Paren1IntRangeNonNull(LPAREN single_integer_range RPAREN)
;
single_integer_range:
    SingleIntegerRangeNull()
    | SingleIntegerRangeOption1(integer_options)
    | SingleIntegerRangeOption2(integer LT_LE_INTEGER optsign)
    | SingleIntegerRangeOption3(integer_options ltle integer)
    | SingleIntegerRangeOption4(integer LT_LE_INTEGER optsign ltle integer)
;
integer_options:
    IntegerOptionsNull()
    | IntegerOptions1(INTEGER)
    | IntegerOptions2(INTEGER sign)
;
ltle:
    LtLeNull()
    | LtLeNonNullLT(LT)
    | LtLeNonNullLE(LE)
;
real_type_range:
    RealTypeRangeNull()
    | RealTypeRangeSingle(single_real_range)
    | RealTypeRangeMultiple(multiple_real_range)
;
multiple_real_range:
    MultipleRealRangeNull()
    | MultipleRealRangeOption1(paren1_real_range OR paren1_real_range)

```

```

    | MultipleRealRangeOption2(multiple_real_range OR paren1_real_range)
;
paren1_real_range:
    Paren1RealRangeNull()
    | Paren1RealRangeNonNull(LPAREN single_real_range RPAREN)
;
single_real_range:
    SingleRealRangeNull()
    | SingleRealRangeOption1(real_options)
    | SingleRealRangeOption2(real_int ltle REALLY optsign)
    | SingleRealRangeOption3(real_options ltle real_int)
    | SingleRealRangeOption4(real_int ltle REALLY optsign ltle real_int)
;
real_options:
    RealOptionsNull()
    | RealOptions1(REALLY)
    | RealOptions2(REALLY sign)
;
real_int:
    RealInt1(integer)
    | RealInt2(NN_REAL)
    | RealInt3(sign NN_REAL)
;
logical_type_range:
    LogicalTypeRangeNull()
    | LogicalTypeRangeSimple(LOGICAL)
;
list_type_range:
    ListTypeRangeNull()
    | ListTypeRangeOption1(QUOTED_STRING)
    | ListTypeRangeOption2(QUOTED_STRING COMMA list_type_range)
;
modfunexpr:
    ModFunNull()
    | MFunExpr(funexpr opt_sp_type_decl)
;
modtstexpr:
    ModTstNull()
    | MTstExpr(testexpr opt_sp_type_decl)
;
testexpr:
    TExpNull()
    | LitConst(LITERAL)

```

```

| TestPair(expr relop expr)
| TestTriple(expr relop expr relop expr)
| LogIFc(log_index_sup_fun)
| AndPair(AT_AND LPAREN testexpr COMMA testexpr_list RPAREN)
| OrPair(AT_OR LPAREN testexpr COMMA testexpr_list RPAREN)
| NotPair(AT_NOT LPAREN testexpr RPAREN)
| IfTPair(AT_IF LPAREN testexpr COMMA testexpr COMMA testexpr RPAREN)
| TParen(LPAREN testexpr RPAREN)
| TExist(AT_EXIST LPAREN exist_arg COMMA testexpr COMMA testexpr RPAREN)
;
testexpr_list:
    TExpList1(testexpr)
    | TExpList2(testexpr_list COMMA testexpr)
;
funexpr:
    FuncExprNonNull(expr)
;
expr:
    ExprNull()
    | TermSingle(term)
    | TermPair(MINUS_SIGN term)
    | PlusPair(expr PLUS_SIGN term)
    | MinusPair(expr MINUS_SIGN term)
;
term:
    PowerSingle(power)
    | ProdPair(term MULT_SIGN power)
    | QuotPair(term DIVIDE_SIGN power)
;
power:
    FactorSingle(factor)
    | ExpPair(factor EXP_SIGN power)
;
factor:
    Const(constant)
    | Var(variable)
    | ParenExpr(LPAREN expr RPAREN)
    | IfFTPair(AT_IF LPAREN testexpr COMMA expr COMMA expr RPAREN)
    | FExist(AT_EXIST LPAREN exist_arg COMMA expr COMMA expr RPAREN)
;
exist_arg:
    ExistArg1(simplevar)
    | ExistArg2(functional_dependency)

```

```

;
constant:
    ConstNum1(nninteger)
    | ConstNum2(NN_REAL)
    | ConstNum3(QUOTED_STRING)
;
variable:
    VarNull()
    | Variable1(sympar)
    | Variable2(simplevar)
    | Variable3(builtin_function)
    | Variable4(pp_index)
    | Variable5(functional_dependency)
    | Variable6(arith_index_sup_fun)
;
functional_dependency:
    FuncDep(index pinteger LPAREN delim_pp_indices RPAREN)
;
sympar:
    SymParNull()
    | SymPar1(S_P_STEM)
    | SymPar2(S_P_STEM sym_par_indices)
;
sym_par_indices:
    SymParIndices1(index)
    | SymParIndices2(sym_par_indices index)
;
simplevar:
    SimpVarConst(GNAME)
    | SimpVarName(GNAME grindices)
;
grindices:
    GrIndices1(grule_index)
    | GrIndices2(grule_index grindices)
;
grule_index:
    GruleIndexNull()
    | GrInd1(pp_index)
    | GrInd2(replaced_index)
    | GrInd3(offset_index)
    | GrInd4(functional_dependency)
;
replaced_index:

```

```

    ReplacedIndex1(STR optsign pinteger GT)
;
offset_index:
    OffsetIndexNonNull(STR pp_index sign pinteger GT)
;
builtin_function:
    BuiltinFunctionNonNull(AT_SIGN GNAME expr_pack)
;
expr_pack:
    ExprPackNonNull(expr_head expr RPAREN)
;
expr_head:
    ExprHeadNull(LPAREN)
    | ExprHeadNonNull(expr_head expr COMMA)
;
optional opt_sp_type_decl;
opt_sp_type_decl:
    OptSpTypeDeclNull()
    | OptSpTypeDeclPrompt()
    | OptSpTypeDeclNonNull(COMMA WHERE sp_type_decl_clause_list)
;
sp_type_decl_clause_list:
    SpTypeDeclClauseListNull()
    | SpTypeDeclClauseListNonNull1(sp_type_decl_clause)
    | SpTypeDeclClauseListNonNull2(sp_type_decl_clause_list COMMA
        sp_type_decl_clause)
;
sp_type_decl_clause:
    SpTypeDeclClauseNonNull(S_P_STEM IS sp_type)
;
sp_type:
    SymParTypeUnqual(unqual_sp_type)
    | SymParTypeQual(qual_sp_type)
;
qual_sp_type:
    QualSymParTypeInteger(integer_type_range UNIQUE)
    | QualSymParTypeReal(real_type_range UNIQUE)
    | QualSymParTypeLogical(logical_type_range UNIQUE)
    | QualSymParTypeUnique(UNIQUE)
;
unqual_sp_type:
    UnqualSymParTypeInteger(integer_type_range)
    | UnqualSymParTypeReal(real_type_range)

```

```

    | UnqualSymParTypeLogical(logical_type_range)
;
index:
    Index1(INDEX)
    | Index2(C_P_TOKEN)
;
pp_index:
    PpIndex1(index)
    | PpIndex2(S_P_INDEX)
    | PpIndex3(D_P_INDEX)
;
delim_pp_indices:
    IndexPpLetter(pp_index)
    | OpPpIndices(delim_pp_indices COMMA pp_index)
;
arith_index_sup_fun:
    ArithIndexSupFunctionNonNull(it_arith_fun_unit LPAREN expr RPAREN)
;
log_index_sup_fun:
    LogIndexSupFunctionNonNull(it_log_fun_unit LPAREN testexpr RPAREN)
;
it_arith_fun_unit:
    IteratedArithFunUnitNonNull1(AT_ARITH_FC_NAME pp_index index_range)
    | IteratedArithFunUnitNonNull2(AT_ARITH_FC_NAME pp_index index_range
        arith_fun_units)
;
it_log_fun_unit:
    IteratedLogFunUnitNonNull1(AT_LOG_FC_NAME pp_index index_range)
    | IteratedLogFunUnitNonNull2(AT_LOG_FC_NAME pp_index index_range
        log_fun_units)
;
arith_fun_units:
    ArithFunUnitsNonNull1(ARITH_FC_NAME pp_index index_range)
    | ArithFunUnitsNonNull2(ARITH_FC_NAME pp_index index_range arith_fun_units)
;
log_fun_units:
    LogFunUnitsNonNull1(LOG_FC_NAME pp_index index_range)
    | LogFunUnitsNonNull2(LOG_FC_NAME pp_index index_range log_fun_units)
;
index_range:
    IndexRangeNull()
    | IndexRangeOption1(index_range1)
    | IndexRangeOption2(index_range2)

```

```

    | IndexRangeOption3(index_range3)
    | IndexRangeOption4(index_range4)
;
index_range1:
    IndexRange1Option1(LT nzinteger COLON nzinteger GT)
    | IndexRange1Option2(LT nzinteger COLON pp_index STR)
    | IndexRange1Option3(LT nzinteger COLON pp_index sign pinteger STR)
;
index_range2:
    IndexRange2Option1(LT pp_index COLON nzinteger GT)
    | IndexRange2Option2(LT pp_index COLON pp_index PLUS_SIGN
        pinteger GT)
;
index_range3:
    IndexRange3Option1(LT pp_index MINUS_SIGN pinteger COLON
        nzinteger GT)
    | IndexRange3Option2(LT pp_index MINUS_SIGN pinteger COLON
        pp_index GT)
    | IndexRange3Option3(LT pp_index MINUS_SIGN pinteger COLON
        pp_index sign pinteger GT)
;
index_range4:
    IndexRange4Option1(LT pp_index PLUS_SIGN pinteger COLON
        nzinteger GT)
    | IndexRange4Option2(LT pp_index PLUS_SIGN pinteger COLON
        pp_index PLUS_SIGN pinteger GT)
;
optional opt_element_detail;
opt_element_detail:
    OptElementDetailNil()
    | OptElementDetailPrompt()
    | OptElementDetailTable(ed_table_list)
;
ed_table_list:
    EDTableListSingle(ed_table TPERIOD)
    | EDTableListCompound(ed_table TPERIOD ed_table_list)
;
ed_table:
    EDTableEmpty(table_struct)
    | EDTableFull(table_struct line_list)
;
table_struct:
    TableStructOption(QUOTED_STRING COMMA

```



```

        QUOTED_STRING COMMA AN_INTEGER)
;
line_list:
    LineListOption1(data_line)
    | LineListOption2(data_line line_list)
;
data_line:
    DataLineOption1(QUOTE_BAR_BAR_QUOTE COMMA data_list)
    | DataLineOption2(data_list COMMA QUOTE_BAR_BAR_QUOTE)
    | DataLineOption3(data_list COMMA QUOTE_BAR_BAR_QUOTE COMMA data_list)
;
data_list:
    DataListOption1(data)
    | DataListOption2(data_list COMMA data)
;
optional data;
data:
    DataOptionNull()
    | DataOptionPrompt()
    | DataOption1(QUOTED_STRING)
    | DataOption2(A_REAL)
    | DataOption3(AN_INTEGER)
;

```

Bibliography

- [Aho86] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).
- [Arba86] B. Arbab, Compiling Circular Attribute Grammars into Prolog, *IBM Journal of Research and Development*, 30(3):294–309 (May 1986).
- [Asht88] Ashton–Tate, *Framework III*, 20101 Hamilton Ave., Torrance, CA 90502 (1988).
- [Bahl86] R. Bahlke and G. Snelting, The PSG System: From Formal Language Definitions to Interactive Programming Environments, *ACM Trans. on Programming Languages and Systems*, 8(4) (October 1986).
- [Bake86] T. Baker, Hierarchical/Relational Approach to Modeling, in *paper presented at the conference on Integrated Modeling Systems*, University of Texas at Austin (1986).
- [Biss87] J. Bisschop, Language Requirements for A Priori Error Checking and Model Reduction in Large–Scale Programming, in *Proceedings of the NATO Advanced Study Institute on Mathematical Models for Decision Support*, Val d’Isere, France (July 27 – August 6 1987).
- [Boeh87] B.W. Boehm, Improving Software Productivity, *Computer* (September 1987).
- [Brad87] G.H Bradley and R.D. Clemence Jr., A type calculus for executable modeling languages, *IMA Journal of Mathematics in Management*, 1(4):277–291 (1987).
- [Broo88] A. Brooke, D. Kendrick, and A. Meeraus, *GAMS: A User’s Guide*, The Scientific Press, Redwood City, California (1988).
- [Chil85] C. Childs and C.R. Meacham, ANALYTICOL—An Analytical Computing Environment, *AT&T Technical Journal*, 64(9):1995–2007 (November 1985).
- [Chir79] L.M. Chirica and D.F. Martin, An Order–Algebraic Formulation of Knuthian Semantics, *Math. Syst. Theory*, 13:1–27 (1979).

- [Clem86] G. Clemm and L. Osterweil, *A Mechanism for Environment Integration*, CU-CS-323-86, Department of Computer Science, University of Colorado (April 1986).
- [Cuni87a] R.A. Cuninghame-Green and R.S. Stainton, Special Issue on Mathematical Programming Modelling Systems, *IMA Journal of Mathematics in Management*, 1(3):147–236 (1986/87).
- [Cuni87b] R.A. Cuninghame-Green and R.S. Stainton, Special Issue on Mathematical Programming Modelling Systems, *IMA Journal of Mathematics in Management*, 1(4):237–300 (1986/87).
- [Cunn89] K. Cunningham and L. Schrage, *The LINGO Modeling Language*, University of Chicago (1989), 89 pages.
- [Dart87] S.A. Dart, R.J. Ellison, P.H. Feiler, and A.N. Habermann, Software Development Environments, *Computer*, 18–28 (November 1987).
- [Deme81] A. Demers, A. Reps, and T. Teitelbaum, Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors, pp. 105–116, in *Conference Record of the 8th ACM Symposium on Principles of Programming Languages*, Williamsburg, VA (26-28 January 1981).
- [Dera86] P. Deransart, M. Jourdan, and B. Lorho, *A Survey on Attribute Grammars, Part II, Review of Existing Systems*, Technical Report 510, INRIA-Rocquencourt, B.P.105, 78153 Le Chesnay Cedex, France (March 1986).
- [DeRe76] F. DeRemer and H. Kron, Programming-in-the-Large Versus Programming-in-the-Small, *IEEE Transactions on Software Engineering*, 2(2):80–86 (June 1976).
- [Donz80] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, *Programming environments based on structured editors: The MENTOR experience*, Tech. Rep. 26, INRIA, Rocquencourt, France (July 1980).
- [Eijk83] P. Van Der Eijk and H.A. Patton, *Description of the GAMS Machine* (April 1983), First Draft.
- [Elli85] R.J. Ellison and B.J. Staudt, The evolution of the GANDALF system, *J. Syst. Softw.*, 5(2):107–119 (May 1985).
- [Farr82] R. Farrow, Experience with an Attribute Grammar-based Compiler, in *Ninth Annual ACM Symposium on Principles of Programming Languages* (25-27 January 1982).

- [Four89] R. Fourer, D.M. Gay, and B.W. Kernighan, *AMPL: A Mathematical Programming Language*, Computing Science Technical Report 133, AT&T Bell Laboratories, Murray Hill, N.J. (Revised June 1989), forthcoming in *Management Science*.
- [Gay89a] D.M. Gay, *AMPL Extensions and Changes (DRAFT)*, AT&T Bell Laboratories, Murray Hill, N.J. (July 14 1989).
- [Gay89b] D.M. Gay, *AMPL Syntax Summary (DRAFT)*, AT&T Bell Laboratories, Murray Hill, N.J. (July 14 1989).
- [Geof87] A.M. Geoffrion, An Introduction to Structured Modeling, *Management Science*, 33(5):547–588 (May 1987).
- [Geof88] A.M. Geoffrion, *SML: A Model Definition Language for Structured Modeling*, Working Paper 360, Western Management Science Institute, UCLA (May 1988), revised November, 1989.
- [Geof89a] A.M. Geoffrion, Computer-Based Modeling Environments, *European Journal of Operational Research*, 41(1):33–43 (July 1989).
- [Geof89b] A.M. Geoffrion, The Formal Aspects of Structured Modeling, *Operations Research*, 37(1):30–51 (January–February 1989).
- [Geof89c] A.M. Geoffrion, *Indexing in Modeling Languages for Mathematical Programming*, Working Paper 371, AGSM, UCLA (November 1989).
- [Geof89d] A.M. Geoffrion, Integrated Modeling Systems, *Computer Science in Economics and Management*, 2:3–15 (1989).
- [Geof89e] A.M. Geoffrion, *A Library of Structured Models*, Informal Note, AGSM, UCLA (8 November 1989).
- [Geof90] A.M. Geoffrion, *FW/SM: A Prototype Structured Modeling Environment*, draft Working Paper 377, AGSM, UCLA (13 April 1990).
- [Gogu87] J. Goguen and M. Moriconi, Formalization in Programming Environments, *Computer*, 55–64 (November 1987).
- [Gord78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, A metalanguage for interactive proof in LCF, in *Proc. 5th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Tucson, Arizona (1978).
- [Gord79] M.J.C. Gordon, *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, New York (1979).

- [Gree83] H.J. Greenberg, A Functional Description of ANALYZE: A Computer-Assisted Analysis for Linear Programming Models, *ACM Transactions on Mathematical Software*, 9:18–56 (1983).
- [Hoar69] C.A.R. Hoare, An Axiomatic Basis for Computer Programming, *Communications ACM*, 576–583 (October 1969).
- [Horw85] S. Horwitz, *Generating Language-Based Editors: A Relationally-Attributed Approach*, PhD dissertation, Computer Science Department, Cornell University (1985).
- [Horw86] S. Horwitz and T. Teitelbaum, Generating editing environments based on relations and attributes, *ACM Trans. on Programming Languages and Systems*, 8(4) (October 1986).
- [Hü87] T. Hürlimann, *LPL: A Structured Language for Modeling Linear Programs*, Peter Lang, Series V, Volume 865, Bern, Switzerland (1987).
- [John78] S.C. Johnson, *YACC: Yet Another Compiler-Compiler*, UNIX Programmer's Manual Vol. 2B, Bell Laboratories, Murray Hill, N.J. (July 1978).
- [Jone84] N.D. Jones and A. Mycroft, Stepwise Development of Operational and Denotational Semantics for Prolog, pp. 281–288, in *Proc. 1984 Intl. Symp. on Logic Programming*, IEEE Computer Society, Atlantic City (6–9 February 1984).
- [Kais89] G.E. Kaiser, Incremental Dynamic Semantics for Language-Based Programming Environments, *ACM Trans. on Programming Languages and Systems*, 11(2) (April 1989).
- [Kini82] V. Kini, D.F. Martin, and A. Stoughton, Testing the INRIA Ada Formal Definition: The USC-ISI Formal Semantics Project, pp. 120–128, in *Proc. of the AdaTEC Conference on ADA*, Arlington, Virginia (6–8 October 1982).
- [Knut68] D.E. Knuth, Semantics of context-free languages, *Math. Syst. Theory*, 2(2):127–145 (June 1968).
- [Knut71] D.E. Knuth, Semantics of context-free languages: Correction, *Math. Syst. Theory*, 5(1):95–96 (March 1971).
- [Lesk75] M.E. Lesk, *Lex – A Lexical Analyzer Generator*, Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J. (1975).
- [Matu87] S.V. Maturana, *Comparative Analysis of Mathematical Programming Systems*, Working Paper 347, AGSM, UCLA (May 1987).

- [Meye88] B. Meyer, Genericity Versus Inheritance, *Journal of Pascal, Ada, & Modula-2*, 7(2):13–30 (1988).
- [Much80] S.S. Muchnick and U.F. Pleban, A Semantic Comparison of LISP and SCHEME, in *Conf. Record of the 1980 LISP Conference*, Stanford Univ. (August 25–27 1980).
- [Murt87] B.A. Murtagh and M.A. Saunders, *MINOS 5.1 User's Guide*, Technical Report SOL 83–20R, Systems Laboratory, Department of Operations Research, Stanford University (1987).
- [Reps83] T.W. Reps, *Generating Language-Based Environments*, MIT Press (1983).
- [Reps89a] T.W. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York (1989).
- [Reps89b] T.W. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual*, Springer-Verlag, New York, third edition (1989).
- [Ritc74] D.M. Ritchie and K.L. Thompson, The UNIX time sharing system, *CACM*, 17(7) (July 1974).
- [Robi79] J.A. Robinson, *Logic: Form and Function*, North Holland, New York (1979).
- [Rose86] R. Rosenthal, Review of the GAMS/MINOS Modeling Language and Optimization Program, *OR/MS Today*, 13(3):24–32 (1986).
- [Roy86] A. Roy, L. Lasdon, and J. Lordeman, Extending Planning Languages to Include Optimization Capabilities, *Management Science*, 32(3):360–373 (March 1986).
- [Schr86] L. Schrage, *Linear, Integer and Quadratic Programming with LINDO*, The Scientific Press, 540 University Avenue, Palo Alto, CA 94301 (1986), Third Edition, 284 pages.
- [Soft] Maximal Software, *Reykjavik, Iceland*.
- [Ster86] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA (1986).
- [Tenn76] R.D. Tennent, The Denotational Semantics of Programming Languages, *Communications ACM*, 437–453 (August 1976).
- [Ullm82] J.D. Ullman, *Principles of Database Systems*, Second Edition, Computer Science Press, Rockville, MD (1982).

- [Vicu90] F. Vicuña, *Specification of a Syntax-Directed Editor for SML*, draft Technical Report, Computer Science Department, University of California, Los Angeles, CA (May 1990).
- [Wand84] M. Wand, A Semantic Prototyping System, *SIGPLAN Notices*, 19(6):213–221 (June 1984).
- [Welc87] J.S. Welch, PAM – A Practitioner’s Approach to Modeling, *Management Science*, 33(5):610–625 (May 1987).