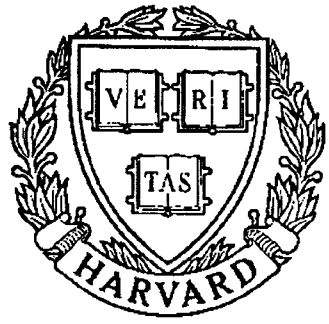
TECHNICAL RESEARCH REPORT



SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

Grammars and Relations

by L. Mark and R. Cochrane

TR 89-47

Grammars and Relations

Leo Mark and Roberta Cochrane
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

Programming languages and databases have evolved on separate foundations and with separate goals for many years. Buzzwords such as persistent data objects, object–oriented databases and software engineering databases reflect increased activity aimed at integrating the two areas at their current state of evolution.

This paper suggests that we return to basics and consider the foundation of the two areas, grammars and relations, as a basis for their integration.

We present an algorithm, *GeneRel*, which given a grammar automatically generates a set of normalized relations in which objects derived from the grammar can be stored.

We demonstrate how the algorithm applied to a meta–grammar generates relations in which grammars derived from the meta–grammar can be stored.

We show how an extended relational algebra can be used to retrieve information about stored grammars and stored objects derived from the grammars.

We outline an algorithm, *GeneView*, which given a set of non–terminals from a grammar generates a set of view definitions on the created relations.

1. Introduction

Inspired by databases, programming languages have adopted the notion of persistent data structures; and inspired by programming languages, databases have adopted the notion of object orientation. In addition, substantial interest in using databases to support software engineering processes has resulted in a number of software engineering databases. Having followed this development, we find that an integration of the two areas at their current state of evolution has not been achieved.

A programming language with persistent objects does not achieve the goals proposed by relational databases: it does not promote information sharing; it does not support data structure inheritance; it does not guarantee consistency; it does not support concurrency; it does not support set oriented retrieval languages like relational algebra; it does not give the same performance as a database.

A database with operations or methods specified in its schema cannot be object–oriented in the sense of programming languages without losing its identity as a database: it cannot support information hiding or be extensible (by introducing new data types through a definition of operations) while, at the same time, supporting a set–oriented retrieval language which allows access to all data; it does not support method inheritance.

The idea of using databases to support software engineering is good, but the systems we have seen so far are not adequate. The database schemata of these systems support specific software engineering process models. Therefore, the systems can only be used by software engineers subscribing to these specific models. The schemas cannot change dynamically, they cannot be extended dynamically, and information cannot be shared between software engineers using

different process models.

Instead of trying to integrate the many important concepts from programming languages and databases at their current state of evolution, we provide a foundation on which an integrated evolution can start. Our work is based on the fundamentals of programming languages and databases: context free grammars [Hopcroft & Ullman] and database relations [Codd].

We propose an algorithm, *GeneRel*, which automatically generates a set of normalized relation schemes R from a given grammar G. Objects derived from grammar G can be stored in the set of relations R.

$$GeneRel: G \rightarrow R$$

We propose an *extended relational algebra* with operators especially designed to retrieve information from the automatically generated set of relations R. Finally, we propose a view generation mechanism, *GeneView*. Let grammar G' be a version of grammar G with the subset of the non-terminals representing objects of interest to a user marked. *GeneView* generates a set of view relations V from the set of relations R and the marked grammar G' using the extended relational algebra.

$$GeneView: G' \rightarrow V$$

Grammars are used extensively in a number of areas that would benefit from a tight integration with databases or from database support:

Software engineering

Software engineering processes can be defined in terms of grammars; we can therefore automatically generate relations in which information about instances of products and processes in a software engineering process can be stored.

Grammars and programs can be defined in terms of grammars; we can therefore automatically generate relations in which they can be stored. This implies that we have relational database support and access to program libraries.

Expert systems

Horn clauses, deductive rules, production rules can all be defined by grammars; we can therefore automatically generate relations in which they can be stored.

Engineering systems

Engineering process plans and complex objects can be defined in terms of grammars; we can therefore automatically generate relations in which we can store information about instances of engineering processes and complex objects.

Extensible databases

New data types that we want a database to support can be defined by grammars; we can therefore automatically generate relations in which instances of these data types can be stored. Furthermore, instances of these new data types can be readily accessed using the extended relational algebra, thereby avoiding the hiding of internal structure of new data types that normally comes with methods.

Persistent objects

If the definition of types in a program is given in terms of a grammar, then relations that provide persistent support for data used in the program can be automatically generated.

One of the past problems of integrating and using databases in the above areas has been that the databases must be designed by hand. The algorithms presented in this paper remove that obstacle. This is important, not only because database design takes time, but also because the people

that need the databases may be experts in using grammars but novices in database design. Another important aspect is standardization. If two database designers design databases for the same grammar, the databases are bound to come out different. This obstacle is also removed by using the algorithms.

There has been a considerable number of efforts merging databases and programming languages specific to each of the above application areas:

- Database support for *Software engineering* has been developed in systems like DAMASCUS [Dadam *et al.*], GENESIS [Ramamoorthy, Usuda, Tsai & Prakash], and CACTIS [Hudson & King].
- Deductive rule systems, like PROBE [Dayal & Smith] and POSTGRES [Stonebraker, Hanson, & Hong] indicate the effort to merge databases and *Expert systems.*
- The use of Relational DBMSs in the support of Computer Aided Design (CAD) [Batory & Kim], [Dadam *et al.*] indicate efforts in extending existing database management systems to accommodate *Engineering* applications.
- *Extensible databases* are being studied in the context of object–oriented database management systems such as EXODUS [Carey *et al.*]
- The representation of databases in programming languages, i. e. *Persistent objects* is the theme of the collection of papers [Atkinson, Buneman, & Morrison].

However, we are familiar with only a few closely related efforts that consider a combination of the fundamentals – grammars and relations:

[Gyssens, Paredaens, & Van Gucht 88], [Gyssens, Paredaens, & Van Gucht 89]

define a new and powerful data model, the grammatical model, with data structures based on grammars and with an algebra for manipulating grammars. The grammatical model would be an ideal basis for a database supporting the new application areas described above.

Our approach agrees with theirs in recognizing the importance of grammars as a tool for describing information in many new application areas. The major difference is that we provide an algorithm for generating relations from a grammar definition and use an extended relational algebra for manipulating the contents of the relations.

The two approaches can be thought of as alternative solutions to the same problems. It is also possible to think of our approach as an automatic implementation of the grammatical model on top of a relational model.

[Horwitz & Teitelbaum]

form a symbiosis of the relational model and attribute grammars by defining a model for language–based editing environments that includes a relational database. Programs are represented as attributed abstract syntax trees with an associated relational database which aggregates information that would otherwise be scattered throughout the program tree.

We also consider the combination of relations and grammars. However, we look at how grammars can be used as a specification of a database; they consider how a relational database can be used to support specifications and operations in attribute grammars.

They define the concept of *implied relations*, which are defined by the structure of the abstract syntax tree, and their use in queries, but they never discuss how such relations would be made persistent.

Inspiration for this work comes from our previous efforts in *Self-Describing Databases, Engineering Information Systems, Operative Expert Systems,* and *Software Engineering Databases.* In

[Mark] we introduced the notion of *Self-Describing Databases* with multiple levels of intensions and extensions of data explicitly represented. This notion has inspired the framework used in the present work (see Section 5). In [Mark & Roussopoulos] we used *Update Dependencies*, a powerful operational database specification language, to specify the intension–extension dimension of a relational database system. In [Cochrane] we described Update Dependencies through the use of attribute grammars. The semantic actions were specified in terms of Update Dependencies against a set of relations storing Update Dependencies. These relations were manually generated from a context free grammar specification of Update Dependencies. This process was a key inspiration for the *GeneRel* algorithm presented in this paper. Finally, in [Mark & Rombach] we investigated the automatic generation of customized *Software Engineering Databases* from grammatical specifications of software processes.

The paper is structured as follows. In the following section, the class of grammars chosen as input to the algorithms is given. Section 3 consists of a brief overview of the Relational Model. The algorithm, *GeneRel*, is presented in section 4. Section 5 consists of applications of *GeneRel* showing how a meta–grammar for the class of grammars described in section 2 can be used by *GeneRel* to store grammar definitions. In section 6, we address the issue of retrieving information. The relational algebra described in section 3 is extended to include the traversal operators. Using the extended relational algebra, we describe the *GeneView* algorithm. Future research topics are presented in section 7.

2. Grammars

Context free grammars have played an important role in every aspect of computer science. We have experimented with a number of formalisms for expressing context free grammars to find one for which an algorithm can produce a nice set of normalized relations. However, this augmentation does not increase the power of the languages expressible by context free grammars. It simply structures the definition of the context free grammars into a more usable format.

In this section, we first give a formal definition of context free grammars [Hopcroft & Ullman] and then present our grammar formalism, *tagged context free grammars*, on which *GeneRel* operates.

DEFINITION

A *context free grammar* is a 4 tuple $\mathbf{G=(V,T,P,S)}$ where:

- $\mathbf{V}$ is a finite set of *nonterminals*;
- $\mathbf{T}$ is a finite set of *terminals*;
- $\mathbf{V}$ and $\mathbf{T}$ are disjoint;
- $\mathbf{S}$ is the special symbol called the start symbol;
- $\mathbf{P}$ is a finite set of *productions* of the form $\mathbf{A} \rightarrow \alpha$ where $\mathbf{A} \in \mathbf{V}$ and α is a string generated from the *regular expression* $(\mathbf{V} \cup \mathbf{T})^*$.

□

The grammar formalism used by *GeneRel* is a variant of context free grammars. It incorporates the concept of *tokens*, the adaptation of *closure* and *positive closure* notations, and the inclusion of *tag–names*.

The concept of *tokens* is included because, as in other applications of context free grammars, it is often convenient to group together terminal characters into single entities. These single entities are referred to as *tokens*. However, we would like to further distinguish between those tokens that have associated data and those that do not. We define *delimiters* as those tokens that do not have associated data values and *lexicons* as those tokens that have associated data values. This distinction is made because the data values for lexicons must be explicitly represented for any string in the defined language. We will assume the existence of a lexical analyzer that returns tokens and values.

The *closure* (*) and *positive closure* ($^+$) notations used in the representations of regular expressions are convenient ways of indicating repeating structures. Although we can develop an algorithm to generate relations without these notations, a grammar expressed using these notations allows the generation of relations that utilizes set retrieval aspects of the relational query languages. This will be discussed in further detail in section 3.

The idea to include *tag-names* into our tagged context free grammar was inspired by [Madsen & Nϕrgaard]. This notation allows the user to specify meaningful names for the generated relations and attributes. Note that we currently force all nonterminal and lexicon occurrences to be *tagged*; however, this restriction can be lifted by introducing default naming conventions.

DEFINITION

A *tagged context free grammar* is a 6 tuple **E=(S, V, L, D, T, P)** where:

- **S** is the special symbol called the start symbol;
- **V** is a finite set of *nonterminals*;
- **L** is a finite set of *lexicons*;
- **D** is a finite set of *delimiters*;
- **V**, **L**, and **D** are disjoint;
- **T** is a set of *tag-names*;
- **P** is a set of *productions* of the form $\mathbf{<t{:}A>} \rightarrow \alpha$ where $\mathbf{A} \in \mathbf{V}$, $\mathbf{t} \in \mathbf{T}$, **t** uniquely specifies a production, and α has one of the following forms:
  - a string generated by the *regular expression* $\mathbf{(<T{:}V> \cup <T{:}L> \cup D)^*}$ where each element from **T** is unique within the string,
  - $\mathbf{K^*}$ where **K** is a symbol in $\mathbf{(<T{:}V> \cup <T{:}L> \cup D)}$
  - $\mathbf{K^+}$ where **K** is a symbol in $\mathbf{(<T{:}V> \cup <T{:}L> \cup D)}$

□

A production consists of a *left-side* and a *right-side*. The *left-side* is the string (i.e. <t:A> from the definition) that precedes the "→", and the *right-side* is the string that follows the "→".

The *nonterminals*, *lexicons*, and *delimiters* make up the symbols of the grammar.

A *tagged symbol* is of the form <t:A> where t is a *tag-name*, and A is either a *nonterminal* or a *lexicon* (i.e. A is a symbol requiring storage).

There are two applications of tag names that correspond directly to the restrictions of the use of tag-names listed in the definition:

- to uniquely name productions within a grammar
- to uniquely name non–delimiter symbols within the right–side of a production.

The first use allows the algorithm to distinguish between produtions defined for the same nonterminal. A given nonterminal can exist on the left–side of more than one production. We therefore tag the nonterminal with a tag–name that must be unique among all tag–names used to tag left–side nonterminals. The second use allows the algorithm to distinguish between the non–delimiter symbols within the right–side of a single production. Since a given symbol can occur several times on the right–side of a single production, each non–delimiter symbol is given a tag–name which must be unique among all tag–names used within the right–side of the given production.

Productions are classified as either *constructor* rules or *list* rules. Productions that have a right–side of the first form given in the definition are *constructor* rules. Productions that have a right–side of the second and third forms given in the definition are *list* rules.

List rules indicate that strings derivable from the production consist of a concatenation of occurrences of the single specified symbol. The closure notation (*) means that there can be zero or more such occurrences; the positive closure notation ($^+$) indicates that there is at least one such occurrence.

EXAMPLE

A block in a programming language is constructed from a list of statments enclosed by the strings "begin" and "end".

<BLOCK:BLOCK> → begin <body:STMTLIST> end

where "BLOCK" and "STMTLIST" are nonterminals, "begin" and "end" are delimiters, and "BLOCK" and "body" are tag–names.

A list of statments in a programming language is a possibly empty list of statments. This is expressed by the list rule:

<STMTLIST:STMTLIST> → <stmt:STMT>*

where "STMTLIST" and "STMT" are nonterminals and "STMTLIST" and "stmt" are tag–names.

□

3. Relations

The Relational Model, [Codd], is a widely accepted database model based on the set–theoretic *relation*. This section briefly overviews the structures and operations of the model, emphasizing those aspects of the model referenced in the subsequent sections of the paper.

It is often convenient to think of a relation as a table in which there is no duplication of rows (tuples), row order is insignificant, column (attribute) order is insignificant, and all table entries are atomic values. The relational scheme defines the table name, column headings and domains.

We now give the formal definitions of these concepts.

DEFINITIONS

A *relational scheme* for a relation has the form $R(A_1{:}D_1, A_2{:}D_2, \ldots A_n{:}D_n)$ where

- **R** is the *relation name*
- $\mathbf{A_i}$'s are *attribute names* denoting attributes of the relation; each attribute name is unique within a given relation scheme
- $\mathbf{D_i}$'s are *domain names* denoting domains over which the attributes are defined; domains need not be distinct.

A *relation* R consists of a set of tuples where a *tuple* is a mapping from the set of attributes to a set of values in the associated domains.

A *key* of a relation is a combination of attributes for which the corresponding tuple values are unique.

The *relational algebra* is a procedural language consisting of operators which yield relations when applied to relations. An extensional definition of these operators is summarized in Figure 1.

□

union	$R \cup S$	set of all tuples in R or in S
intersection	$R \cap S$	set of all tuples in R and in S
set difference	R / S	set of all tuples in R that are not also in S
cartesian product	$R \times S$	relation obtained by concatenating every tuple in R with every tuple in S
projection	$\pi_{A_1,A_2,\ldots,A_n} R$	relation obtained by dropping all attributes not listed and removing duplicates
selection	$\sigma_{expression} R$	set of all tuples in R satisfying the *expression*
theta join	$R \underset{R.A\theta S.B}{\bowtie} S$	all tuples in the cartesian product of R and S in which R.A stands in relation θ with S.B.
natural join	$R \bowtie S$	relation obtained by dropping one of the identical attributes of $R \underset{R.A=S.A}{\bowtie} S$

Figure 1: Relational Algebra Operators

EXAMPLE

The relation scheme BLOCK(occur:BLOCK, body:STMTLIST) defines a relation with the two attributes "occur" and "body" defined over the domains BLOCK and STMTLIST respectively. The relation BLOCK consists of a set of tuples $<$occur:B_i, body:$S_i$$>$ where $B_i \in$ BLOCK and $S_i \in$ STMTLIST.

□

The extended relational model used as a basis for the algorithms is the relational model as described above extended with:

- domains of *surrogates* [Hall, Owlett & Todd] for representing non–lexical object types. Surrogates are systems generated internal identifiers that are ideal for representing unnamed objects such as uses of production rules in a grammar.
- a set of *traversal operators* which simplify the expression of queries about the stored sentences and grammars. These are operators are further described in section 6.

4. The GeneRel Algorithm

The *GeneRel* algorithm generates a set of relation schemes from a tagged context free grammar. *GeneRel* generates one relation scheme for each production in the grammar. Relation name, attributes names and domains are determined from the tag names and symbols in the

production.

The form of the generated relation scheme depends on the structure of the production as summarized in Figure 2. Relation names and attribute names are derived from the tag names, with the exception that position attributes are introduced for list rules to indicate the order of symbols in a given sentential form. The type of a symbol in the production determines the domain. For each nonterminal N in the grammar, a domain N of surrogates uniquely represent derivations of the nonterminal N. For each lexicon L in the grammar, a domain L of lexical objects represents the syntactic category defined by the lexicon.

Note that productions are stored in relations generated from a meta-grammar to allow reconstruction of stored sentences. This will be discussed in the next section.

As mentioned in section 2, productions are classified as either *constructor* rules or *list* rules depending on the structure of the right-side. We further classify the list rules: rules whose right-side symbol is a delimiter are *list-of-delimiter* rules; rules whose right-side symbol is a non-terminal or a lexicon are *list-of-structure* rules. Accordingly, relations generated for constructor rules are *constructor relations*, relations generated for list-of-structure rules are *list-of-structure relations*, and relations generated for list-of-delimiter rules are *list-of-delimiter relations*.

A *constructor* rule has the form:

$$<t_0{:}A> \rightarrow w_1 <t_1{:}A_1> w_2 <t_2{:}A_2> \ldots w_n <t_n{:}A_n> w_{n+1}$$

where w_i is either a delimiter or the empty string, A_i is a nonterminal or lexicon, A is a nonterminal, and t_i is a tag-name.

The relation name for a constructor relation is the tag-name of the left-side nonterminal of the corresponding constructor rule. Each constructor relation has one attribute named "occur" that is defined over the domain A. This attribute constitutes a key attribute for the relation. Additionally, there is an attribute corresponding to every nonterminal or lexicon symbol on the right-side. The attribute name is the tag-name, t_i, and the domain is A_i. If A_i is a lexicon, the

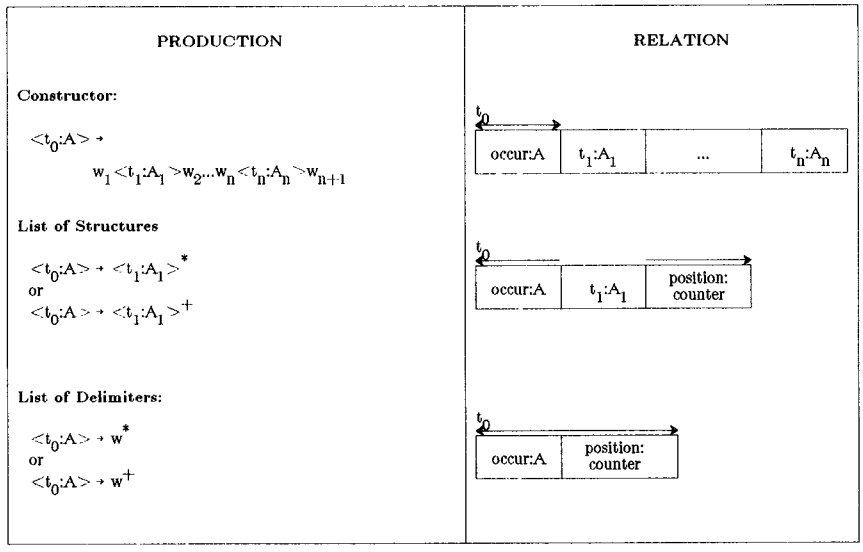


Figure 2: *GeneRel* Algorithm

domain A_i is a lexical domain. If A_i is a nonterminal, the domain A_i is a domain of surrogates.

A *list–of–structure* rule has the form:

$$<t_0{:}A> \rightarrow <t_1{:}A_1>^*$$
or
$$<t_0{:}A> \rightarrow <t_1{:}A_1>^+$$

where A_i is a nonterminal or lexicon, A is a nonterminal, and t_i is a tag–name.

The relation name for a list–of–structure relation is the tag–name of the left–side nonterminal of the corresponding list–of–structure rule. There is one attribute named "occur" that is defined over the domain A, and there is one attribute corresponding to the nonterminal or lexicon symbol A_i with attribute name t_i and domain A_i. In addition, there is an attribute named "position" defined over the domain of positive integers. This "position" attribute records the position of the symbol (represented by the attribute t_i) in the sentence (represented by the attribute "occur"). The attributes "occur" and "position" constitute a key for the relation.

A *list–of–delimiter* rule has the form

$$<t_0{:}A> \rightarrow w^*$$
or
$$<t_0{:}A> \rightarrow w^+$$

where w is a delimiter, A is a nonterminal, and t_0 is a tag–name.

The relation name for a list–of–delimiter relation is the tag–name of the left–side nonterminal of the corresponding rule. There is one attribute named "occur" that is defined over the domain A, and there is a "position" attribute defined over the domain of positive integers. As with the list–of–structure relation, this "position" attribute records the position of the delimiter in the sentence. The attributes "occur" and "position" constitute a key for the relation. It is unnecessary to store delimiter values since they are the same for each occurrence and can be retrieved from the stored underlying grammar.

EXAMPLE

This example shows the production set of a tagged grammar that defines a simplified structured programming language (Figure 3). A block is defined as a possibly empty list of statements. The lexicons "id" and "int" are assumed to be defined in the lexical analyzer.

```
<BLOCK:BLOCK>          →  begin <body:STMTLIST> end
<STMTLIST:STMTLIST>    →  <stmt:STMT>*
<IFSTMT:STMT>          →  if <cond COND>
                          then <trueact:STMTLIST>
                          else <falseact:STMTLIST> endif
<ASSIGNSTMT:STMT>      →  <var:id> := <value:int>
<EQUAL:COND>           →  <var:id> == <value:int>
```

Figure 3: Example Tagged Grammar

Figure 4 shows the relation schemes that are generated by applying *GeneRel* to the above grammar. We introduce a new notation here for expressing relation schemes which allows us to highlight shared domains (which indicate joinable attributes). The domains are represented by circles: a solid circle represents a domain of surrogates (non lexical objects), a dashed circle represents a domain of lexical objects. The rectangles denote the relation schema with the

attribute names occurring within the rectangle and the relation name occurring somewhere outside but near the rectangle. The arrowed lines indicate attributes that constitute a key for the relation.

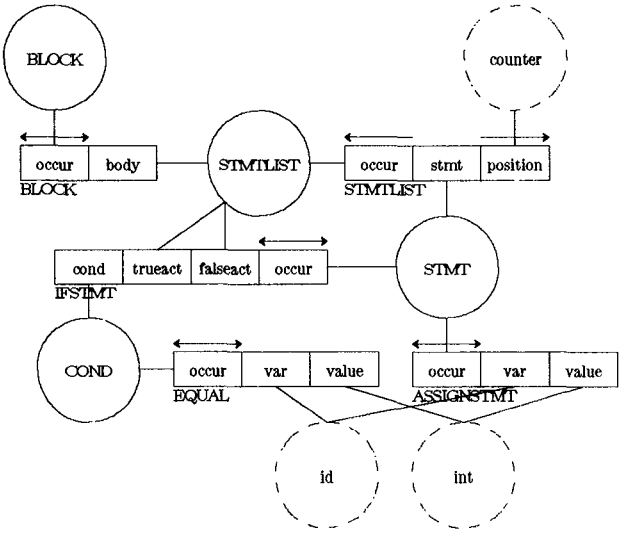


Figure 4: Relations Generated

□

5. Meta–grammars, Grammars, Programs

Given a tagged context free grammar, the *GeneRel* algorithm generates a set of relation schemes under which sentences derived from the grammar can be stored. It is also necessary to store the grammar in order to automate the reconstruction of stored sentences. To generate relations under which grammars can be stored we define a meta–grammar for the class of tagged context free grammars. This meta–grammar is itself defined using a tagged context free grammar. By applying the *GeneRel* algorithm on the meta–grammar, a set of relation schemes is generated under which any tagged grammar – including the meta–grammar itself – can be stored.

A parser is needed in order to store sentences derived from a grammar under the relation schemes generated by *GeneRel*. This parser must be capable of not only parsing sentences, but also storing them under the relation schemes. Such a parser can be generated by a compiler–compiler. We need one such parser to be generated from the meta–grammar, and we need one parser to be generated from each new grammar we define. The parser generated from the meta–grammar must be hardwired with the relation generating rules of the *GeneRel* algorithm and insert statement generating rules. The resulting intension–extension framework is illustrated in Figure 5. It is quite similar to the intension–extension framework for DBMSs presented in [Mark] and in [Mark & Roussopoulos]

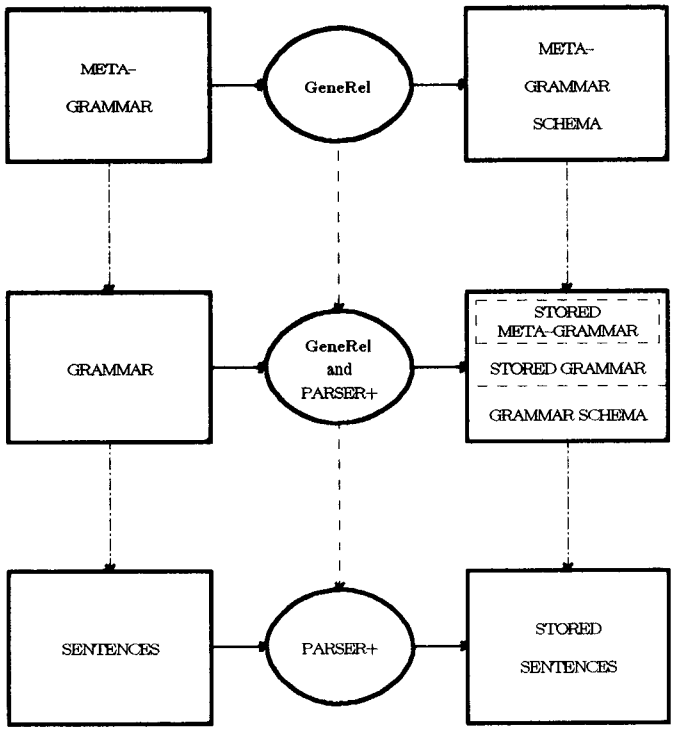


Figure 5: Intension–Extension Framework

We shall now show examples of the three levels of the intension–extension framework. We first apply *GeneRel* to a meta–grammar for the class of grammars described in section 2, generating the meta–level relations for storing grammar definitions. We then give an example of a grammar, show how it is stored in the meta–level relations, and show how *GeneRel* applied to it generates a new set of relations. We finally give an example of a program and show how it is stored in the new set of relations.

In Figure 6, we define a tagged meta–grammar which defines the class of tagged context free grammars.

```
{ GRAMMAR,
  { GRAMMAR START NONTERMS LEXICONS DELIMITERS TAG_NMS PRODS PRODUCTION
    NONTERM_TAG RHS SYMBOL
    LEXICON_TAG }
  { nonterminal delimiter lexicon tag_name }
  { { } → *+ ' <
    > : }
  { GRAMMAR, start, nonterms, lexicons, delimiters, tag_nms, prods, START, value,
    NONTERMS, LEXICONS, DELIMITERS, TAG_NMS, PRODS, PRODUCTIONS, left-side, right-side,
    CONSTRUCTOR, LIST, POSITIVE_LIST, DELIMITER_SYM, NONTERM_SYM, LEXICON_SYM,
    NONTERM_TAG, name, domain, LEXICON_TAG }
  {

    <GRAMMAR:GRAMMAR>          →  '{'<start:START>
                                  '{'<nonterms:NONTERMS>'}'
                                  '{'<lexicons:LEXICONS>'}'
                                  '{'<delimiters:DELIMITERS>'}'
                                  '{'<tag_nms:TAG_NMS>'}'
                                  '{'<prods:PRODS>'}' '}'

    <START:START>              →  <value:nonterminal>

    <NONTERMS:NONTERMS>        →  <value:nonterminal>+
    <LEXICONS:LEXICONS>        →  <value:lexicon>*
    <DELIMITERS:DELIMITERS>    →  <value:delimiter>*
    <TAG_NMS:TAG_NMS>          →  <value:tag_name>*

    <PRODS:PRODS>              →  <value:PRODUCTION>*
    <PRODUCTIONS:PRODUCTION>→  <left-side:NONTERM_TAG> '→'
                                  <right-side:RHS>

    <CONSTRUCTOR:RHS>          →  <value:SYMBOL>*
    <LIST:RHS>                 →  <value:SYMBOL>'*'
    <POSITIVE_LIST:RHS>        →  <value:SYMBOL>'+'
    <DELIMITER_SYM:SYMBOL>     →  '''<value:delimiter>'''
    <NONTERM_SYM:SYMBOL>       →  <value:NONTERM_TAG>
    <LEXICON_SYM:SYMBOL>       →  <value:LEXICON_TAG>
    <NONTERM_TAG:NONTERM_TAG>  →  '<'<name:tag_name>':'<domain:nonterminal>'>'
    <LEXICON_TAG:LEXICON_TAG>  →  '<'<name:tag_name>':'<domain:lexicon>'>'
  }
}
```

Figure 6: Meta–Grammar

Figure 7 depicts the relation schemes generated by applying *GeneRel* to the highlighted meta–grammar productions from Figure 6. The appendix contains the full meta–grammar schema, followed by an illustration of the meta–grammar stored under the meta–grammar schema.

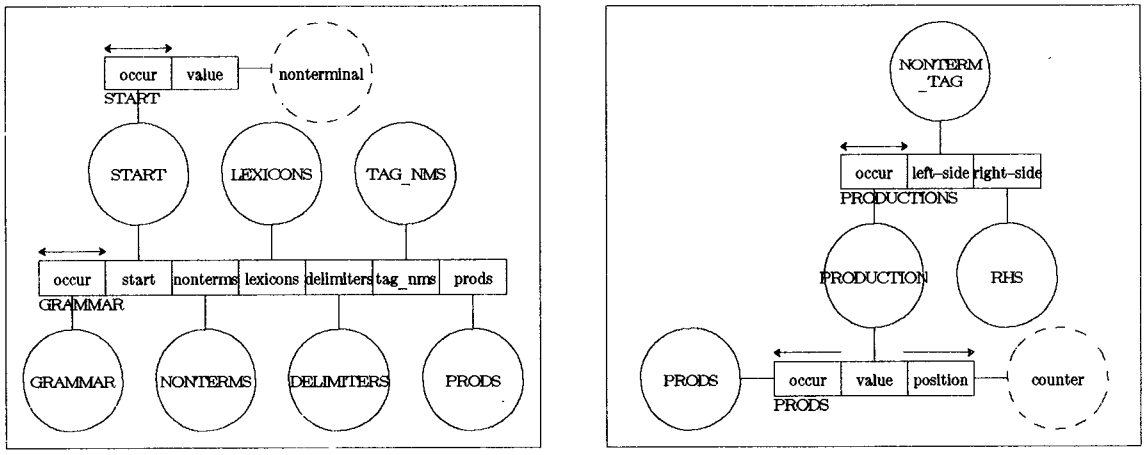


Figure 7: Meta–Grammar Schema

In Figure 3 we gave an example of a tagged context free grammar and Figure 4 showed the relations generated for this grammar. Figure 8a and 8b shows this grammar stored under the meta–grammar schema.

GRAMMAR

occur : GRAMMAR	start : START	nonterms NONTERMS	lexicons: LEXICONS	delimiters DELIMITERS	tag_nms: TAG_NMS	prods : PRODS
G_2	S_2	NT_2	TKS_2	TS_2	TNS_2	PS_2

START

occur : START	value : nonterminal
S_2	"BLOCK"

NONTERMS

occur : NONTERMS	occur : nonterminal	position : counter
NT_2	"BLOCK"	1
NT_2	"STMTLIST"	2
NT_2	"STMT"	3
NT_2	"COND"	4

LEXICONS

occur : LEXICONS	value : lexicon	position : counter
TKS_2	"id"	1
TKS_2	"int"	2

TAG_NMS

occur: TAG_NMS	value: tag_name	position: counter
TNS_2	"BLOCK"	1
TNS_2	"body"	2
TNS_2	"STMTLIST"	3
TNS_2	"stmt"	4
TNS_2	"IFSTMT"	5
TNS_2	"cond"	6
TNS_2	"trueact"	7
TNS_2	"falseact"	8
TNS_2	"ASSIGNSTMT"	9
TNS_2	"var"	10
TNS_2	"value"	11
TNS_2	"EQUAL"	12

DELIMITERS

occur : DELIMITERS	value : delimiter	position : counter
TS_2	"begin"	1
TS_2	"end"	2
TS_2	"if"	3
TS_2	"then"	4
TS_2	"else"	5
TS_2	"endif"	6
TS_2	":="	7
TS_2	"=="	8

PRODS

occur : PRODS	value : PRODUCTION	position : counter
PS_2	P_51	1
PS_2	P_52	2
PS_2	P_53	3
PS_2	P_54	4
PS_2	P_55	5

Figure 8a: Stored Grammar

PRODUCTIONS

occur : PRODUCTION	left-side : NONTERM_TAG	right-side : RHS
P_51	TG_51	R_51
P_52	TG_53	R_52
P_53	TG_55	R_53
P_54	TG_59	R_54
P_55	TG_62	R_55

LIST

occur : RHS	value : SYMBOL
R_52	SYM_54

POSITIVE_LIST

occur : RHS	value : SYMBOL

DELIMITER_SYM

occur : SYMBOL	value : delimite:
SYM_51	"begin"
SYM_53	"end"
SYM_55	"if"
SYM_57	"then"
SYM_59	"else"
SYM_61	"endif"
SYM_63	":="
SYM_65	"=="

CONSTRUCTOR

occur : RHS	value : SYMBOL	position : counter
R_51	SYM_51	1
R_51	SYM_52	2
R_51	SYM_53	3
R_53	SYM_55	1
R_53	SYM_56	2
R_53	SYM_57	3
R_53	SYM_58	4
R_53	SYM_59	5
R_53	SYM_60	6
R_53	SYM_61	7
R_54	SYM_62	1
R_54	SYM_63	2
R_54	SYM_64	3
R_55	SYM_62	1
R_55	SYM_65	2
R_55	SYM_64	3

LEXICON_SYM

occur : SYMBOL	value : TOK_TAG
SYM_62	TG_10
SYM_64	TG_11

NONTERM_TAG

occur: TAG	name: tag_name	domain: nonterminal
TG_51	"BLOCK"	"BLOCK"
TG_52	"body"	"STMTLIST"
TG_53	"STMTLIST"	"STMTLIST"
TG_54	"stmt"	"STMT"
TG_55	"IFSTMT"	"STMT"
TG_56	"cond"	"COND"
TG_57	"trueact"	"STMTLIST"
TG_58	"falseact"	"STMTLIST"
TG_59	"ASSIGNSTMT"	"STMT"
TG_12	"EQUAL"	"COND"

LEXICON_TAG

occur: TAG	name: tag_name	domain: DOMAIN
TG_10	"var"	"id"
TG_11	"value"	"int"

NONTERM_SYM

occur : SYMBOL	value : NONTERM_TAG
SYM_52	TG_52
SYM_54	TG_4
SYM_56	TG_6
SYM_58	TG_7
SYM_60	TG_8

Figure 8b: Stored Grammar

Figure 9 shows a program written in the example grammar and its storage under the relations generated from the example grammar.

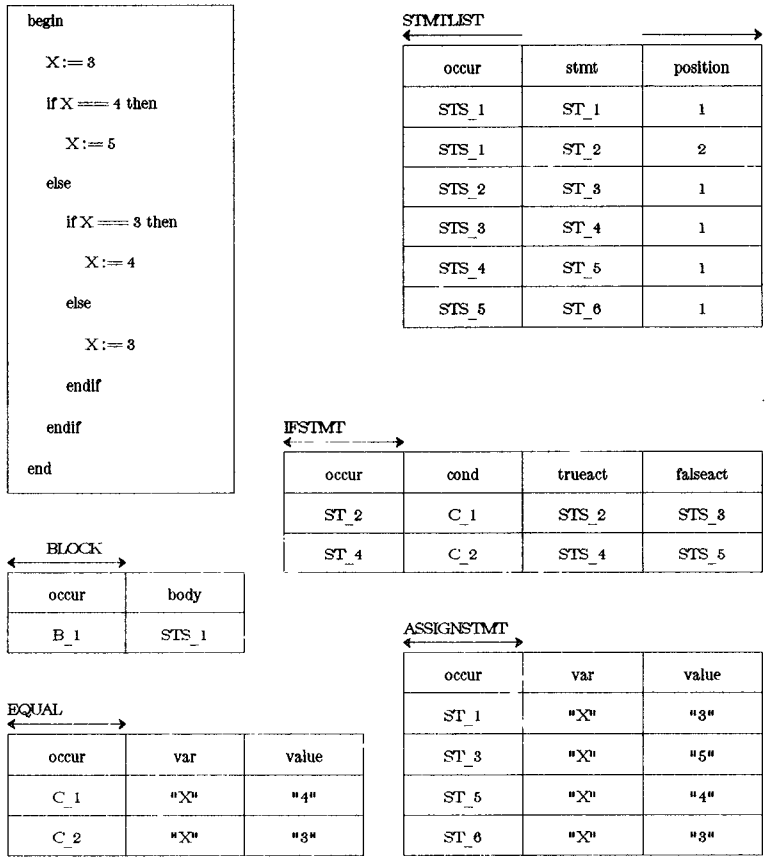


Figure 9: Program

This section has demonstrated how the sentences of a language and the grammars defining the language can be stored under the same framework. This property allows us to use the same retrieval language for asking questions about the grammars and sentences.

6. The Extended Algebra

Thus far we have shown how relation schemes are generated from tagged grammar specifications and how sentences derived from these grammars are stored under the generated relation schemes. In this section the issue of expressing queries about the stored sentences and the stored grammars is considered.

There are many different queries that one might ask about stored sentences and stored grammars:

What are all the components of an *if-statement*?

What are all sentences derivable from a given *nonterminal*?

Is *nonterminal N* used in the definition of *nonterminal S*?

Which *if-statement* has *C1* as its *condition*?

What are all the *if-statements* which have one of the members of the set *C* as its *condition*?

Sentences are stored as derivation trees. Queries about these sentences can span multiple levels of the derivation trees and therefore involve multiple relations. Queries can also be made about

the structure of grammars. However, only queries about sentences need to be considered since grammars are stored sentences under the meta–grammar.

Many queries can be expressed in relational algebra. However, queries that involve traversing the derivation tree become quite complicated. Still other queries are recursive in nature and thus cannot be described by the relational algebra.

We introduce a set of operators as an extension to the relational algebra; they simplify the tree traversal queries while also solving the problem of recursive queries. They can be combined with relational algebra for selecting nodes in the tree and finding properties of these nodes. The tree traversal performed by the operators is directed by the stored grammars.

Before defining our operators, some terminology is introduced. Suppose G is a directed graph with nodes n_1, n_2, ... n_k. The REACH of node n_i is the set of all nodes n_j such that there is a path from n_i to n_j in G. The set of all nodes n_j such that n_i is in the REACH of n_j (i.e. there is a path from n_j to n_i) is referred to as the INVERSE REACH of the node n_i.

The set of operators is defined below using the above terminology.

DERIVES

Let N be a unary relation with one attribute of surrogate values denoting nodes. $\Longrightarrow^* N$ is a binary relation with attributes *node* and *sentence* representing the mapping between nodes in N and their derived sentences.

REACH

Let D be a nonterminal representing a *descendent node type*; let AN be a unary relation with one attribute of surrogate values denoting the *ancestor nodes*. $\rho_D AN$ is all nodes of the *descendent node type* that are in the REACH of at least one of the *ancestor nodes*. An explanation of a computation of this operator is given in the examples.

INVERSE REACH

Let A be a nonterminal representing a *ancestor node type*; let DN be a unary relation with one attribute of surrogate values denoting *descendent nodes*. $\rho^{-1}{}_A DN$ is all nodes of the *ancestor node type* that are in the INVERSE REACH of one of the *descendent nodes*. An explanation of a computation of this operator is given in the examples.

The traversal operators translate into queries on the database, some of which may be recursive and expensive. However, techniques for optimizing recursive queries have been developed [Jagadish, Agrawal & Ness], [Ullman 89] which may considerably reduce the cost.

EXAMPLES

The question "Which variables participate in an *assignment-statement*?" is formed by a projection of the variable attribute of the ASSIGNSTMT relation.

$$\pi_{var} \text{ ASSIGNSTMT}$$

Similarly the question "Which variables participate in an *equality-condition*?" is answered by the query:

$$\pi_{var} \text{ EQUAL}$$

By performing selections the retrieved tuples can be restricted by some common property. The question "Which variables have a "4" assigned to them?" is answered by the query:

$$\pi_{var} \underset{value=4}{\sigma} \text{ASSIGNSTMT}$$

Retrieving all surrogates of a nonterminal is accomplished by simply taking the union of all the occur attributes of the relations associated with the productions for the nonterminal. Note that the surrogates cannot be printed. Such a query is useful in a programmatic interface where some property or operation needs to be computed on the entire set of such surrogates. The following query retrieves all occurrences of nonterminal STMT.

$$\pi_{occur} \text{IFSTMT} \bigcup \pi_{occur} \text{ASSIGNSTMT}$$

To retrieve all the sentences derived from the nonterminal STMT the DERIVES operator is applied to the result of the previous query.

$$\Longrightarrow^{*} (\pi_{\text{occur}} \text{IFSTMT} \bigcup \pi_{occur} \text{ASSIGNSTMT})$$

Now, suppose it is necessary to retrieve the surrogates of all statements which contain an assignment of the value "3" to a variable. The simplest such statments are *assignment-statements.* However, *if-statements* contain *assignment-statements.* The leaves of the derivation tree satisfying our condition must be selected and all nodes representing the nonterminals of type STMT that are on the path from these leaves to the root must be collected. Such a tree traversal query can be expressed using the INVERSE REACH operator ρ^{-1}.

$$\rho^{-1}{}_{STMT}\, \pi_{occur} \underset{value=3}{\sigma} \text{ASSIGNSTMT}$$

As promised, a computation of ρ^{-1} is now described. Given a query as above plus the stored grammar a set of equations can be generated. The fix-point of these equations contains the desired result. That is, the evaluation of the equations is repeated until no new results are obtained.

By examining the generated relations for the example, we see that the above query can be computed by repeatedly executing the following set of joins until nothing new is obtained. These joins can be generated automatically from the stored meta-grammar. The algorithm for this generation is not trivial and is left for future research. In the following equations, I_<relation> indicates an intermediate relation used for computing a result.

$$\text{I_STMTLIST} \equiv \text{STMTLIST} \underset{stmt=occur}{\rhd\!\lhd} \text{I_STMT}$$

$$\begin{aligned} \text{I_STMT} \equiv\ & \pi_{IFSTMT.occur} \text{IFSTMT} \underset{trueact=occur}{\rhd\!\lhd} \text{I_STMTLIST} \bigcup \\ & \pi_{IFSTMT.occur} \text{IFSTMT} \underset{falseact=occur}{\rhd\!\lhd} \text{I_STMTLIST} \bigcup \\ & \pi_{ASSIGNSTMT.occur} \underset{value=3}{\sigma} \text{ASSIGNSTMT} \end{aligned}$$

Evaluation begins with the intermediate relations initialized to the empty set. An assignment of the right-side of the equations to the left-side is repeatedly computed until no more tuples can be added to any of the intermediated relations.

As mentioned above, the result of the query is contained in the fix-point of this system of equations. This query is concerned with occurrences of the nonterminal STMT, and thus the answer is the value of the intermediated relation I_STMT.

This computational model is very similar to the computation of fix–points of datalog equations, [Ullman 88] where our intermediate relations correspond to the IDB predicates in datalog.

The results of a ρ^{-1} operation can be combined with other algebraic operators to further enhance the query. For example, let the result of the above query be the view P. Now the query "retrieve all conditions of if statements that contain the assignment of the value "3" to a variable" can be computed as:

$$\pi_{cond}\,(\,\mathrm{IFSTMT} \underset{occur=\,occur}{\rhd\!\lhd} \mathrm{P}\,)$$

To demonstrate the use of the REACH operator ρ, consider the query: "Retrieve the surrogates of all assignment statements which are embedded in if–statements that have a condition on the variable X". This query can be formulated in the extended algebra as:

$$\rho_{ASSIGNSTMT}\,\rho^{-1}{}_{IFSTMTS}\,\pi_{occur}\,\underset{var=X}{\sigma}\,\mathrm{EQUAL}$$

Similar to the example of ρ^{-1}, a system of equations for the above query can be generated from the query and the stored grammar. The desired result is one of the final values of one of the intermediate relations that results from the computation of the fix–point of this system of equations.

The answer of the query is the intermediate relation I_ASSIGNSTMT resulting from the computation of the fix–point of the following system of equations.

$$\begin{aligned}
\mathrm{I_IFSTMT} \equiv\ & \pi_{IFSTMT.occur}\,\mathrm{IFSTMT} \underset{cond=\,occur}{\rhd\!\lhd} \pi_{occur}\,\underset{var=X}{\sigma}\,\mathrm{EQUAL} \cup \\
& \pi_{I_IFSTMT.occur}\,\mathrm{I_STMTLIST} \underset{occur=\,trueact}{\rhd\!\lhd} \mathrm{I_IFSTMT} \cup \\
& \pi_{I_IFSTMT.occur}\,\mathrm{I_STMTLIST} \underset{occur=\,falseact}{\rhd\!\lhd} \mathrm{I_IFSTMT}
\end{aligned}$$

$$\mathrm{I_STMTLIST} \equiv \pi_{STMTLIST.occur}\,\mathrm{STMTLIST} \underset{stmt=\,occur}{\rhd\!\lhd} \mathrm{I_IFSTMT}$$

$$\mathrm{I_ASSIGNSTMT} \equiv \pi_{ASSIGNSTMT.occur}\,\mathrm{I_STMTLIST} \underset{stmt=\,occur}{\rhd\!\lhd} \mathrm{ASSIGNSTMT}$$

The algorithm for evaluating a query and determining the system of equations is a topic for future research. It seems that such an algorithm will be quite similar to those presented in [Ullman 88] for generating datalog equations from a datalog program. The difference is that our algorithm will use the query plus a stored grammar to generate the datalog equations.

□

7. The GeneView Algorithm

The notions of normalization for grammars and relations are quite similar.

When a database is normalized, some of the relations do not directly correspond to the real world objects that are of interest to the user of the database; however, they are needed to retrieve information about these objects and their relationships to other objects. The objects which are of interest to the user are usually made easily accessible through the definition of views. These views are derived from the normalized database using the relational algebra.

A grammar has a number of productions where the left–side non–terminal does not directly correspond to a class of objects for which the user has interest; again, these productions are needed to tie together the objects that are of interest. Using the extended relational algebra we can define views on the set of relations generated by *GeneRel*. This makes the objects of interest readily available to the user. This view generation process can be automated by an algorithm, *GeneView*, which given a marked grammar G' generates a set of view definitions V on the relations. Each view contains a set of objects derived from the corresponding non–terminal.

$$GeneView: G' \rightarrow V$$

The existence of *GeneView* is important. Even with the addition of the algebra operators from the previous section, the retrieval language may be of little interest to a user who views the structure of his data through a grammar. The type of query language that would be useful to such a user is one in which the user could mark certain interesting aspects of the grammar. This concept is feasible since the grammars are stored. *GeneView* not only automates the view generation process, but it also reduces the number of relations that users need to know to access the database. Users only need to be aware of those relations that correspond to non–terminals which represent the objects of interest.

In the simplest case, the user could indicate the set of nonterminals in which the stored sentences are of interest. These views are defined by the DERIVES operator applied to all occurrences of the nonterminal.

EXAMPLE

Suppose the user indicates an interest in the nonterminal <STMT>. *GeneView* would generate the view:

$$\text{STMT_V} \equiv \text{DERIVES } (\pi_{occur} \text{ IFSTMT} \bigcup \pi_{occur} \text{ ASSIGNSTMT})$$

□

This is a very simple example of a generated view which alleviates the need to know about the underlying relational schema to access desired information. We are also considering the possibilities of generating views from a marking of the grammar through a pictorial interface.

8. Future Research

The following tasks will be the core of our future research efforts:

(1) Definition of an algorithm for determining the system of equations for a query which uses the traversal operators.

(2) Extension of the tagged grammar and *GeneRel* to handle attribute grammars.

(3) Investigation of the relationship between grammar transformation and database reorganization. Some of the interesting issues under this category of research are:

- The design of algorithms for automatic database reorganization from grammar transformations such as reductions, normal form transformations, syntax directed translation schemes, etc. Thus the semantics of database reorganization can be expressed through grammar transformations.
- The possibility of defining mappings between two databases in terms of syntax directed translation schemes. Although the same concepts used in database reorganization apply here, such mappings are intended to provide an interface between users of separate

databases that model the same concepts.

(4) Application of the database reorganization algorithms to extendible databases, categorized by the following questions:

- What happens when a set of lexicons in a grammar is replaced by a more detailed grammatical definition? The corresponding database problem is to replace a primitive domain of character strings representing a sentence by a set of relations storing the sentence in parsed form. Can the database reorganization algorithms automatically expand the database definition to support the new data type? Is our approach a good solution to the problem of extensible databases?
- Similarly, what happens when a detailed grammatical definition is simplified? Can the database automatically be reorganized to support that?

(5) Investigation of the applicability of our algorithms to object–oriented systems: Can the view concept supported by the *GeneView* algorithm approximate the concept of information hiding in object–oriented systems?

(6) A continued study of the extension of the relational algebra to determine what kind of operations are useful for grammars?

(7) Investigation of the possibility of extending *GeneRel* to apply to non–context free grammars. Are there grammars for which such an extension is useful? One such example is the grammar containing one production: AB → CDE Such a production might produce the relation R(A,B,C,D,E) with (A,B) as key.

(8) Application of existing techniques for optimizing recursive queries.

References

[Atkinson, Buneman, & Morrison]
Atkinson, M. P., P. Buneman, and R. Morrison, eds., *Data Types and Persistence,* Springer–Verlag, 1988.

[Batory & Kim]
Batory, D. S. and W. Kim, Modeling Concepts for VLSI CAD Objects, in *Supplement to ACM SIGMOD International Conference on Management of Data,* 1985, pages 18–32.

[Carey *et al.*]
Carey, M., The EXODUS Extensible DBMS Project: An Overview, in *Readings in Object–Oriented Databases,* edited by D. Maier, Morgan–Kaufman Publ. Co., 1989.

[Cochrane] Cochrane, Roberta, *Operational Data Model – Implementation Through Specification,* Department of Computer Science and Systems Research Center, College Park, Maryland, SRC-TR–89–46, 1989.

[Codd] Codd, E. F., Extending the Database Relational Model to Capture More Meaning, *ACM Transactions on Database Systems* **Vol. 4, No. 4** (December 1979), pages 397–434.

[Dadam *et al.*]
Dadam, P., A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies, *Proc. ACM-SIGMOD International Conference on the Management of Data*, Washington DC, May 1986.

[Dayal & Smith]
Dayal, U. and J. M. Smith, PROBE: A Knowledge-Oriented Database Management System, *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Gyssens, Paredaens, & Van Gucht 88]
Gyssens, M., J. Paredaens, and D. Van Gucht, *A Grammar-Based Approach towards Unifying Hierarchical Data Models*, University of Antwerp, Antwerpen, Belgium, Technical Report 88-33, June 1988.

[Gyssens, Paredaens, & Van Gucht 89]
Gyssens, M., J. Paredaens, and D. Van Gucht, A Grammar-Based Approach towards Unifying Hierarchical Data Models (extended abstract), in *Proc. ACM SIGMOD International Conference on Management of Data*, Portland, OR, June 1989, pages 263-272.

[Hall, Owlett & Todd]
Hall, P., J. Owlett, and S. Todd, Relations and Entities, in *Modelling in Database Management Systems*, edited by G. M. Nijssen, North-Holland, 1976.

[Hopcroft & Ullman]
Hopcroft, John E. and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, 1979.

[Horwitz & Teitelbaum]
Horwitz, Susan and Tim Teitelbaum, Generating Editing Environments Based on Relations and Attributes, *ACM Transactions on Programming Languages and Systems* **Vol. 8, No. 4** (October 1986), pages 577-608.

[Hudson & King]
Hudson, S. and R. King, CACTIS: A Database System for Specifying Functionally-Defined Data, in *Proc. Workshop on Object-Oriented Databases*, Pacific Grove, CA, 1986.

[Jagadish, Agrawal & Ness]
Jagadish, H. V., Rakesh Agrawal, and Linda Ness, A Study of Transitive Closure As a Recursion Mechanism, in *Proc. ACM SIGMOD International Conference on Management of Data*, San Francisco, CA, May 1987, pages 331-344.

[Madsen & Nϕrgaard]
Madsen, O. L. and C. Nϕgaard, An Object-Oriented Metaprogramming System, in *Hawaii International Conference on System Sciences*, January 1988.

[Mark] Mark, Leo, *Self-Describing Database Systems - Formalization and Realization,* Department of Computer Science, University of Maryland, College Park, MD., TR-1264, 1985.

[Mark & Rombach]
Mark, Leo and H. D. Rombach, Generating Customized Software Engineering Information Bases from Software Process and Product Specifications, in *Proc. Twenty-Second Annual Hawaii International Conference on System Sciences,* edited by Bruce D. Shriver, IEEE Computer Society Press, Washington, D. C., January 1989, pages 587 - 595.

[Mark & Roussopoulos]
Mark, Leo and Nick Roussopoulos, Metadata Management, *IEEE Computer* **Vol. 19, No. 6** (December 1986).

[Ramamoorthy, Usuda, Tsai & Prakash]
Ramamoorthy, C. V., Y. Usuda, W. Tsai, and A. Prakash, GENESIS: An Integrated Environment for Supporting Development and Evolution of Software, in *Proc. COMPSAC,* 1985.

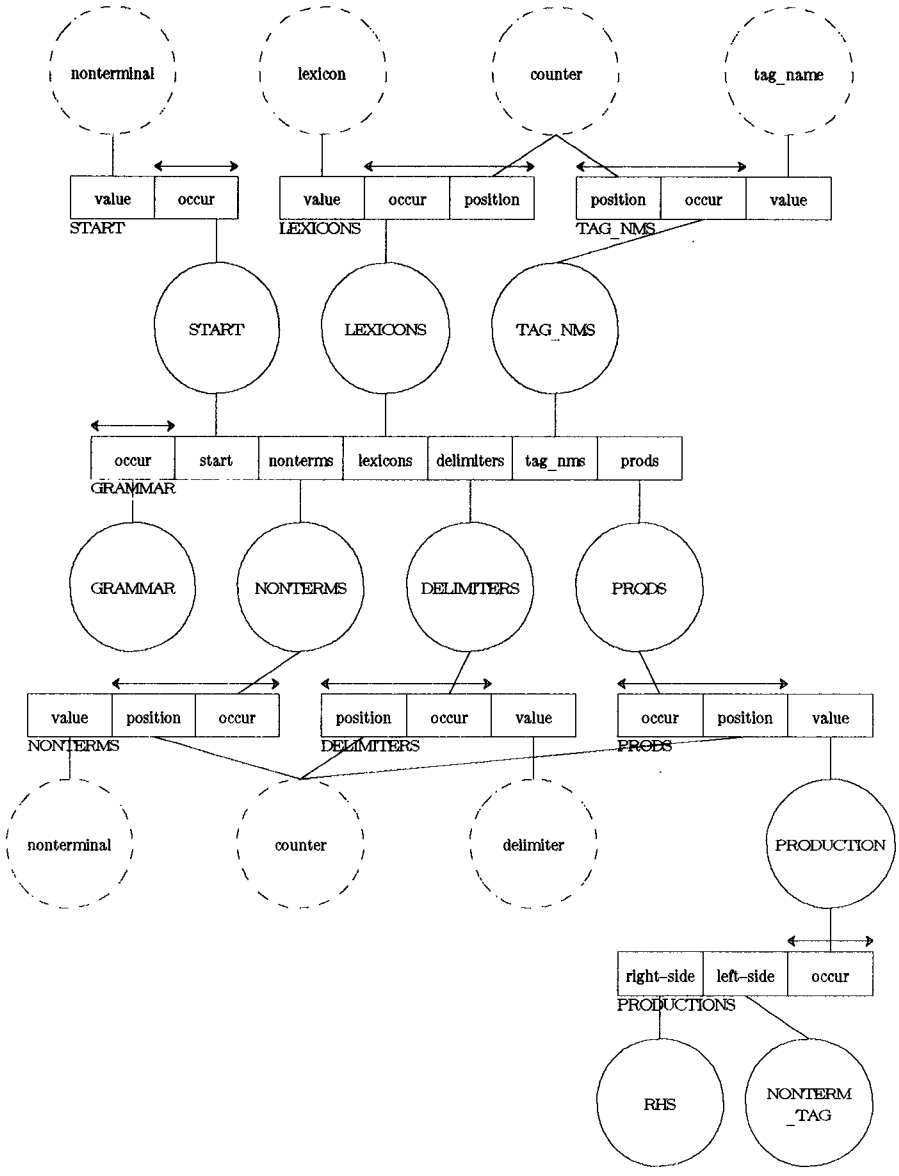
[Stonebraker, Hanson, & Hong]
Stonebraker, M., E. Hanson, and C. Hong, The Design of the POSTGRES Rules System, in *Proc. Third International Conference on Data Engineering,* Los Angeles, CA, February 1987.

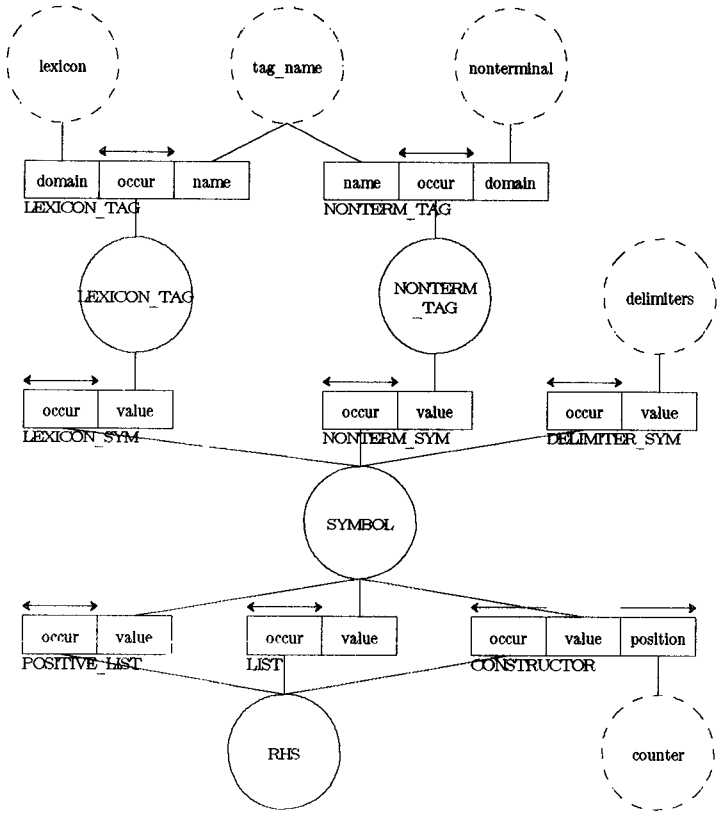
[Ullman 88] Ullman, Jeffrey D., *Principles of Database and Knowledge-Base Systems, Vol. I,* Computer Science Press, 1988.

[Ullman 89] Ullman, Jeffrey D., *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies,* Computer Science Press, 1989.

Appendix

Relations Generated
From the Meta-grammar




lexicon
tag_name
nonterminal
domain
occur
name
LEXICON_TAG
name
occur
domain
NONTERM_TAG
LEXICON_TAG
NONTERM_TAG
delimiters
occur
value
LEXICON_SYM
occur
value
NONTERM_SYM
occur
value
DELIMITER_SYM
SYMBOL
occur
value
POSITIVE_LIST
occur
value
LIST
occur
value
position
CONSTRUCTOR
RHS
counter

The Meta–Grammar Stored Under Meta–Level Relations

GRAMMAR

occur : GRAMMAR	start : START	nonterms NONTERMS	lexicons: LEXICONS	delimiters DELIMITERS	tag_nms: TAG_NMS	prods : PRODS
G_1	S_1	NT_1	TKS_1	TS_1	TNS_1	PS_1

START

occur : START	value : nonterminal
S_1	"GRAMMAR"

NONTERMS

occur : NONTERMS	occur : nonterminal	position : counter
NT_1	"GRAMMAR"	1
NT_1	"START"	2
NT_1	"NONTERMS"	3
NT_1	"LEXICONS"	4
NT_1	"DELIMITERS"	5
NT_1	"TAG_NMS"	6
NT_1	"PRODS"	7
NT_1	"PRODUCTION"	8
NT_1	"NONTERM_TAG"	9
NT_1	"RHS"	10
NT_1	"SYMBOL"	11
NT_1	"LEXICON_TAG"	12

LEXICONS

occur : LEXICONS	value : lexicon	position : counter
TKS_1	"nonterminal"	1
TKS_1	"delimiter"	2
TKS_1	"lexicon"	3
TKS_1	"tag_name"	4

DELIMITERS

occur : DELIMITERS	value : delimiter	position : counter
TS_1	"{"	1
TS_1	"}"	2
TS_1	"→"	3
TS_1	"*"	4
TS_1	"+"	5
TS_1	"'"	6
TS_1	"<"	7
TS_1	">"	8
TS_1	":"	9

TAG_NMS

occur: TAG_NMS	value: tag_name	position: counter
TNS_1	"GRAMMAR"	1
TNS_1	"start"	2
TNS_1	"nonterms"	3
TNS_1	"lexicons"	4
TNS_1	"delimiters"	5
TNS_1	"tag_nms"	6
TNS_1	"prods"	7
TNS_1	"START"	8

TAG_NMS(cont)

occur: TAG_NMS	value: tag_name	position: counter
TNS_1	"value"	9
TNS_1	"NONTERMS"	10
TNS_1	"LEXICONS"	11
TNS_1	"DELIMITERS"	12
TNS_1	"TAG_NMS"	13
TNS_1	"PRODS"	14
TNS_1	"PRODUCTIONS"	15
TNS_1	"left–side"	16
TNS_1	"right–side"	17
TNS_1	"CONSTRUCTOR"	18
TNS_1	"LIST"	19
TNS_1	"POSITIVE_LIST"	20
TNS_1	"DELIMITER_SYM"	21
TNS_1	"NONTERM_SYM"	22
TNS_1	"LEXICON_SYM"	23
TNS_1	"NONTERM_TAG"	24
TNS_1	"name"	25
TNS_1	"domain"	26
TNS_1	"LEXICON_TAG"	27

PRODS

occur : PRODS	value : PRODUCTION	position : counter
PS_1	P_1	1
PS_1	P_2	2
PS_1	P_3	3
PS_1	P_4	4
PS_1	P_5	5
PS_1	P_6	6
PS_1	P_7	7
PS_1	P_8	8
PS_1	P_9	9
PS_1	P_10	10
PS_1	P_11	11
PS_1	P_12	12
PS_1	P_13	13
PS_1	P_14	14
PS_1	P_15	15
PS_1	P_16	16

PRODUCTIONS

occur : PRODUCTION	left-side : NONTERM_TAG	right-side : RHS
P_1	TG_1	R_1
P_2	TG_2	R_2
P_3	TG_3	R_3
P_4	TG_4	R_4
P_5	TG_5	R_5
P_6	TG_6	R_6
P_7	TG_7	R_7
P_8	TG_8	R_8
P_9	TG_9	R_9
P_10	TG_10	R_10
P_11	TG_11	R_11
P_12	TG_12	R_12
P_13	TG_13	R_13
P_14	TG_14	R_14
P_15	TG_15	R_15
P_16	TG_16	R_16

LIST

occur : RHS	value : SYMBOL
R_4	SYM_10
R_5	SYM_11
R_6	SYM_12
R_7	SYM_13
R_9	SYM_17

POSITIVE_LIST

occur : RHS	value : SYMBOL
R_3	SYM_9

CONSTRUCTOR

occur : RHS	value : SYMBOL	position : counter
R_1	SYM_1	1
R_1	SYM_2	2
R_1	SYM_1	3
R_1	SYM_3	4
R_1	SYM_4	5
R_1	SYM_1	6
R_1	SYM_5	7
R_1	SYM_4	8
R_1	SYM_1	9
R_1	SYM_6	10
R_1	SYM_4	11
R_1	SYM_1	12
R_1	SYM_7	13
R_1	SYM_4	14

CONSTRUCTOR(cont)

occur : RHS	value : SYMBOL	position : counter
R_1	SYM_1	15
R_1	SYM_8	16
R_1	SYM_4	17
R_1	SYM_4	18
R_2	SYM_9	1
R_8	SYM_14	1
R_8	SYM_15	2
R_8	SYM_16	3
R_10	SYM_17	1
R_10	SYM_18	2
R_11	SYM_17	1
R_11	SYM_19	2
R_12	SYM_20	1
R_12	SYM_11	2

CONSTRUCTOR(cont)

occur : RHS	value : SYMBOL	position : counter
R_12	SYM_20	3
R_13	SYM_21	1
R_14	SYM_22	1
R_15	SYM_23	1
R_15	SYM_24	2
R_15	SYM_25	3
R_15	SYM_26	4
R_15	SYM_27	5
R_16	SYM_23	1
R_16	SYM_24	2
R_16	SYM_25	3
R_16	SYM_26	4
R_16	SYM_27	5

DELIMITER_SYM

occur : SYMBOL	value : delimiter
SYM_1	"{"
SYM_4	"}"
SYM_15	"→"
SYM_18	"*"
SYM_19	"+"
SYM_20	"'"
SYM_23	"<"
SYM_25	":"
SYM_27	">"

NONTERM_SYM

occur : SYMBOL	value : NONTERM_TAG
SYM_2	TG_17
SYM_3	TG_18
SYM_5	TG_19
SYM_6	TG_20
SYM_7	TG_21
SYM_8	TG_22
SYM_13	TG_27
SYM_14	TG_28
SYM_16	TG_29
SYM_17	TG_30
SYM_21	TG_31
SYM_22	TG_35

LEXICON_SYM

occur : SYMBOL	value : TOK_TAG
SYM_9	TG_23
SYM_10	TG_24
SYM_11	TG_25
SYM_12	TG_26
SYM_24	TG_32
SYM_26	TG_33
SYM_28	TG_34

NONTERM_TAG

occur: TAG	name: tag_name	domain: nonterminal
TG_1	"GRAMMAR"	"GRAMMAR"
TG_2	"START"	"START"
TG_3	"NONTERMS"	"NONTERMS"
TG_4	"LEXICONS"	"LEXICONS"
TG_5	"DELIMITERS"	"DELIMITERS"
TG_6	"TAG_NMS"	"TAG_NMS"
TG_7	"PRODS"	"PRODS"
TG_8	"PRODUCTIONS"	"PRODUCTION"
TG_9	"CONSTRUCTOR"	"RHS"
TG_10	"LIST"	"RHS"
TG_11	"POSITIVE_LIST"	"RHS"
TG_12	"DELIMITER_SYM"	"SYMBOL"
TG_13	"NONTERM_SYM"	"SYMBOL"
TG_14	"LEXICON_SYM"	"SYMBOL"
TG_15	"NONTERM_TAG"	"NONTERM_TAG"
TG_16	"LEXICON_TAG"	"LEXICON_TAG"
TG_17	"start"	"START"

NONTERM_TAG(cont)

occur: TAG	name: tag_name	domain: nonterminal
TG_18	"nonterms"	"NONTERMS"
TG_19	"lexicons"	"LEXICONS"
TG_20	"delimiters"	"DELIMITERS"
TG_21	"tag_nms"	"TAG_NMS"
TG_22	"prods"	"PRODS"
TG_27	"value"	"PRODUCTION"
TG_28	"left-side"	"NONTERM_TAG"
TG_29	"right-side"	"RHS"
TG_30	"value"	"SYMBOL"
TG_31	"value"	"NONTERM_TAG"
TG_35	"value"	"LEXICON_TAG"

LEXICON_TAG

occur: TAG	name: tag_name	domain: DOMAIN
TG_23	"value"	"nonterminal"
TG_24	"value"	"lexicon"
TG_25	"value"	"delimiter"
TG_26	"value"	"tag_name"
TG_32	"name"	"tag_name"
TG_33	"domain"	"nonterminal"
TG_34	"domain"	"lexicon"