GENERATING RELATIONS USING FORMAL GRAMMARS

S.H. von Solms
Rand Afrikaans University
P.O. Box 524, Johannesburg

Financial assistance from the Council for Scientific and Industrial Research and the Ernest Oppenheimer Memorial Trust in South Africa is acknowledged.

SUMMARY

Grammars generating 2-dimensional arrays have been studied by many people [1, 2, 3, 4]. One effort was Random Context Array Grammars [4], where different types of context conditions placed on the production, were used to control the generating process.

Relations, as used in relational data bases, can be considered as rectangular arrays, and therefore it should be possible to generate and manipulate relations using 2-dimensional Grammars.

Simple Relation Grammars generate relations, and can simulate some unary operations like select and project on these relations.

Extended Relation Grammars also generate relations, but these relations can communicate with each other within a "Extended Relation Schema (ERS)". Within such an ERS binary operations like union and join can be simulated.

This paper is a summary of a research project of which certain parts have already been submitted for publication.

1. Simple Relation Grammars

In this section we will only give an informal description of the definition and operation of a Simple Relation Grammar (SRG). A detailed discussion of SRG's can be found in [5].

An SRG is a 4-tuple $G = (V_N, V_T, P, S)$.

i) where $V_N = V_{N_1} \cup V_{N_2} \cup V_{N_3} \cup S$.

- V_{N_1} is a finite set of "entities". Entities are the basic symbols of an SRG. An entity is a string consisting of a finite number of characters from, for example, the ASCII set. "JONES" is an entity. In an SRG, entities are generated, replaced, etc., and can be seen as the "data" going into a relational data base.
- V_{N_2} is a finite set of "column headings", representing the attribute columns of a relation. An element of V_{N_2} is a 4-tuple $c = (e_1, x_1, x_2, x_3)$, where

  $e_1 \in V_{N_1}$, i.e. an entity. e_1 is the "absolute name" of the column heading.

  $x_1 \in \{Y,N\}$, indicating whether this column represents a primary key or not.

  $x_2 \in \{Y,N\}$, indicating whether duplicates are allowed in this column, or not.

  $x_3 \in \{Y,N\}$, indicating whether an entry in this column is compulsory, or not.

  (DEPT, Y, N, Y) is a valid element of V_{N_2} if 'DEPT' $\in V_{N_1}$. In this case, DEPT is a key attribute. For every $c_i \in V_{N_2}$, $\exists D_i \subseteq V_{N_1}$ called the domain of c_i.

  All entities appearing in the column with heading c_i must come from D_i.
- V_{N_3} is a set of utility (auxiliary) symbols.

ii) $S = \{B,N\}$, is the set of start symbols. We assume that S is surrounded by "background" symbols. As a structure is generated from S, these "background" symbols are replaced by symbols from G.

A derivation in G consists of 2 steps: (The productions of G are discussed below).

Step 1: Generation of the column headings (attributes). During this step, symbols from V_{N_2} will be introduced checking, for example, for duplication of "absolute names". A finite number of symbols from V_{N_2} will be generated, representing the attributes and their specifications.

quæstiones informaticæ, 5, 3, pp 51-55, issn 0254-2757

After Step 1 we may, for example, have

(ID-NR,Y,N,Y)	(NAME,N,Y,Y)	(STREET,N,Y,Y)	(AGE,N,Y,Y)
N			

where {"ID-NR", "NAME", "STREET", "AGE"} $\subseteq V_{N_1}$, and N is the start symbol for generating tuples.

Step 2: Generation of the actual rows (tuples) of the relation.

After Step 2, we may have

(ID-NR,Y,N,Y)	(NAME,N,Y,Y)	(STREET,N,Y,Y)	(AGE,N,Y,Y)
234	JONES	BLUE	30
784	BLACK	ACORN	50
N			

where {'234', '784'} $\subseteq$ D (ID-NR,Y,N,Y) $\subseteq V_{N_1}$, etc.

During Step 2, the restrictions as specified by the column headings, for example, no duplications or compulsory entry, are checked and enforced by the productions of the Grammar.

iii) $V_T \subseteq V_{N_1} \cup V_{N_2} \cup \{N\}$, $N \in S$.

iv) P consists of a set of productions of the form:

a) $A \rightarrow \alpha$ $((P_1)_h, (P_2)_v, (P_3)_g; (F_1)_h, (F_2)_v, (F_3)_g)$,P_i , $F_i \subseteq V_N$,
where $1 <= \text{length}(\alpha) <= 2$ and $1 <= i <= 3$.
The production above means A can be replaced by α if -

i) all elements of P_1 and no elements of F_1 appear in the same horizontal row as A,
ii) all elements of P_2 and no elements of F_2 appear in the same vertical column as A.
iii) all elements of P_3 and no elements of F_3 appear (somewhere/anywhere) in the picture (structure) in which A resides presently.

P_1/F_1 is called the horizontal permitting/forbidding context.
P_2/F_2 is called the vertical permitting/forbidding context.
P_3/F_3 is called the global permitting/forbidding context.

If length (α) = 2, i.e. $\alpha = X_1X_2$, X_1 , $X_2 \in V_N$, then a background symbol must appear immediately to the right of A. A is replaced by X_1 and the background symbol by X_2.

If length (α) = 1, i.e. $\alpha = X_1$, $X_1 \in V_N$, then A can be replaced by X_1, even if none of the other symbols bordering X_1 is a background symbol.

This means in effect that a string can only "grow" at the right most end of the string if a production of type (a) is used.

Replacing single symbols within a string by another single symbol can happen anywhere in the string.

b) $A \downarrow \alpha$ $((P_1)_h, (P_2)_v, (P_3)_g; (F_1)_h, (F_2)_v, (F_3)_g)$, where length (α)= 2.The definition and meaning of P_i, F_i, $1 <= i <= 3$, are precisely as in (a).

The production means:

Suppose $\alpha = X_1X_2$, and a background symbol appears immediately "below" A. A can be replaced by X_1, and the background symbol by X_2, if the context conditions as described in (a) holds.

Note that this class of productions gives an SRG its "2-dimensional" character, because a structure can now "grow" both horizontally and vertically.

Any structure consisting of symbols of V_T will be in the language generated by G.

v) Informally we can define the language generated by an SRG $G = (V_N, V_T, P, S)$ as the set of all rectangular arrays consisting of elements of V_T which can be generated from S using productions from P.

2. Generation vs Controlling power

In an SRG we are more concerned with the controlling power of the grammar, as opposed to the generating power. We therefore investigate the power of productions to control and monitor the generative process, and will not be so interested in the actual formal language generated by the grammar. For this reason, the relations generated by SRG's will, in a certain sense, be artificial, because we have little control over the specific "value" of an entity generated in a specific position.

Productions are therefore defined in terms of their controlling power, and the specific values on the left and righthand side of the productions will seldom be explicitly specified.

3. Selection and Projection.

In an SRG $G = (V_N, V_T, P, S)$ the selection operation, i.e. selecting tuples satisfying some set criteria, can be simulated by "marking", with a distinctive mark, those tuples (or part of tuples) satisfying the set criteria.

Given a generated relation, SRG productions can for example do the following:

i) Mark all entities, i.e. all attributes, of the tuple with primary key equal to e_i.
ii) Mark all entities of all tuples having a value e_i for attribute e_j.

Using the different permitting and forbidding context conditions of the productions, this is done very easily and clearly. Just as easily, the result can be projected over specified attributes by "de-marking" the non required column entities.

4. Summary

We introduced a formal mechanism, SRG, to generate and manipulate relations. The selection and projection operations are possible, but in a "primitive way" — "marking" the relevant tuples.

Extended Relation Grammars [6] allow binary operations like union and join, and generate the result of the operation. Therefore, instead of "marking" the result, a new relation is generated containing precisely the result of the specific binary operation.

5. Extended Relation Grammars [6]

An SRG can generate a single rectangular array representing a relation, and can simulate simple unary operations on the resulting relation. No communication with other relations is possible.

To investigate binary operations we need more than one relation. We need a way of distinguishing between these different relations, as well as a way to communicate between these different relations.

We extend the definition of an SRG to provide these facilities, and call the resulting version an Extended Relation Grammar.

5.1 Suppose $G_i = (V_N^i, V_T^i, P^i, S^i)$, $1 \leq i \leq n$, are n SRG's. We give every one a unique index by replacing every

$c = (e, x_1, x_2, x_3) \in V_{N_2} \subseteq V_N^i$ by
$c = (e, x_1, x_2, x_3, i)$, $1 \leq i \leq n$.

This provided every potential header element c from G_i with an index i, uniquely relating it to G_i and distinguishing it from the header element of G_j, $i \neq j$. We can refer to this index by c(5).

5.2 Let $G = \bigcup_{i=1}^{n} V_N^i$.

Suppose
(1) $A \otimes \alpha ((P_1)_h, (P_2)_v, (P_3)_l ; (F_1)_h, (F_2)_v, (F_3)_l)$,
is a production from P^i. $\otimes$ can be $\rightarrow$ indicating a "horizontal" production of $\downarrow$ indicating a "vertical" production.

Change production (1) to
(1) $A \otimes \alpha ((P_1)_h, (P_2)_v, (P_3)_l, (P_4)_g ; (F_1)_h, (F_2)_v, (F_3)_l, (F_4)_g)$,
where P4, F4 $\subseteq G$.
P4 is called the global permitting context, and F4 the global forbidding context.
Because P4, F4 $\subseteq G = \bigcup_{i=1}^{n} V_N^i$, information can be communicated between different structures (relations).

5.3 Apply the changes described in 5.1 and 5.2 to all relevant elements and productions from all G_i. Each G_i has now became an ERG.
We can therefore imagine some "active environment" in which a number of ERG's are operating. Each ERG can still only generate a single structure (relation), but this generation can be influenced by other structures already existing in the active environment.
In discussing ERG's, we will assume such an "active environment" as defined in the next section.

5.4 Note that if for any ERG G, $P_4 = F_4 = 0$ for all productions in G, G cannot communicate with any other structure, and acts precisely like an SRG.

6. Extended Relation Schemas (ERS)

An Extended Relation Grammar can generate a single relation, but can communicate with other relations using its global context. Of course, to communicate with other relations, more than one relation has to exist.

An Extended Relation Schema (ERS) is a set of relations together with the ERG's generating them. The relations in an ERS can communicate with each other via their global contexts, and can influence the operation of each other.

For example, a certain variable X, appearing somewhere in relation R_1, may appear in the global forbidding context of all other relations in the schema. This will prevent any other relation from doing anything, until X is rewritten in R_1.

In the same way, R_1 can "block" itself by introducing a variable, say X. This X appears in the forbidding global context of all but one of the productions, for example production i, of the grammar generating R_1.

This production i has a Y in its permitting global context.

Y can only be generated by the ERG generating another relation, for example R_2.

R_1 is therefore "blocked", waiting for R_2's grammar to introduce a Y. Only then can the generation of R_1 continue.

ERG's in an ERS can therefore synchronize and influence their mutual generation processes.

An Extended Relation Schema is therefore the "active" environment referred to in the previous chapter.

7. Binary Operations - Union and Join

Using this global context, relations in an ERS can now communicate with each other.
Using these facilities, binary operations can be simulated.
We will not describe the detail of these simulations. A thorough discussion is given in [6].

8. Uses of ERS's

Presently, ERS's are used to describe processes within the computer environment. The 2-phase protocol used in transaction management in distributed databases, had been successfully simulated.

References

1 Milgram D.L. and Rosenfeld A., [1971], *Array Automata and Array Grammar,* IFIP 71, North Holland.

2 Siromoney R., Subraman K.G. and Rangarajan K., [1079], Rectangular Array with Tables, *International Journal of Computer Mathematics,* Sec. A, **6**.

3 Siromoney R. and Siromoney G., [1977], Extended Controlled Table L-Arrays, *Information and Control*, **35**, 2.

4 Von Solms S.H., [1980], Random Context Array Grammars, *IFIP 80*, North Holland.

5 Von Solms, S.H., *Simple Relation Grammars* (Submitted)

6 Von Solms, S.H., *Extended Relation Grammars* (In Preparation).

BOOK REVIEW:

***A Model Implementation of Standard Pascal*, Jim Welsh and Atholl Hay,** Prentice-Hall 1986, ISBN 0-13-586454-2, 483 pages.

Reviewed by: Willem van Biljon, *ITR, University of Stellenbosch*

This book consists of three Pascal programs: a Standard Pascal compiler, a P-code interpreter and a post-mortem generator. The compiler accepts the full set of Standard Pascal as described in the British and International standards BS 6192 and ISO 7185, and performs all the enforced syntactic and semantic checks on a source program. It produces an equivalent program in P-code in a CodeFile, together with a NameFile listing all identifiers used, a DataMap file describing the representation and memory map of the identifiers, and a CodeMap file providing information on the source line number to P-code mapping.

The interpreter accepts the P-code produced by the compiler and executes it, performing all the enforced run-time checks on the executing program. In the event of a program failure the interpreter produces a CorpseFile which is a memory image of the program at the time of the failure.

The post-mortem generator accepts the CorpseFile of a program as well as its NameFile and DataMap files, and produces a source-level procedure call trace-back and symbolic variable dump for each called procedure.

The book has two levels. Firstly it presents a full operational definition of Standard Pascal and secondly, it provides an example of a large program written in Pascal.

On the first level it supercedes the well-known Pascal P4 compiler from ETH Zurich written for bootstrapping purposes by Urs Ammann and well documented by Pemberton and Daniels in [1] and [2]. It is interesting to see how some of the more interesting features of Standard Pascal that were either not defined when the P4 compiler was written, or were just left out of that compiler, are implemented. This includes features such as file management, procedures and functions as parameters, and conformant arrays.

On the second level, the book provides a well-documented example of a non-trivial task solved in Pascal. In this sense it highlights both the elegance of Pascal, especially its data structures, and the major failing of Pascal — its lack of a module construct. The program was modularly written throughout, but this programmer-enforced structure could have been far more elegant had some module construct been present in the language.

The book is not perfect. Although generally well-documented, some extra comments about the various data structures would have been welcomed, relieving the reader of time-consuming cross-examination of the declarations and code. There are also some minor inconsistencies in the use of the of the error procedure. C.A.R. Hoare (series editor of the Prentice-Hall Computer Science series) does, however, invite reader's comments and corrections to be sent to Software - Practice and Experience.

A model Implementation of Standard Pascal has a number of applications. It provides students with an example of a bootstrapped recursive-descent compiler (complementing the examples of YACC-generated LALR parsers that have become available). It also provides, due to its modularity, a front-end for more sophisticated code generation techniques. Such code generators could immediately be tested for correctness using the interpreter as execution model. Finally, the post-mortem generator could provide some useful insights into the construction of symbolic debuggers.

I recommend this book to the student of compiler theory who is interested to see an example of a well-written Pascal compiler, and to anyone looking towards the implementation of a Standard Pascal compiler.

References

1. Pemberton S. and Daniels M. C.,[1982], *Pascal implementation: The P4 Compiler,* Ellis Horwood Limited.
2. Pemberton S. and Daniels M. C., [1982], *Pascal Implementation: Compiler and Assembler/Interpreter*, Ellis Horwood Limited .