

A GRAMMATICAL APPROACH TO DATA BASES

A.L. FURTADO and P.A.S. VELOSO
Pontificia Universidade Catolica do R.J.
Rua Marques de S. Vicente, 225, 22453, Rio de Janeiro, RJ, Brasil

Marco A. CASANOVA
Centro Cientifico de Brasilia
IBM do Brasil
Caixa Postal 853, 70.000, Brasilia, DF, Brasil

A generative formalism for the characterization of fundamental data base concepts and for the methodical specification of data base applications is proposed. The main emphasis lies in the definition of a complete formal language reflecting the jargon of each application area. The formalism is based on W-grammars.

1. INTRODUCTION

In this paper we propose a formalism with the double purpose of precisely characterizing certain fundamental concepts in the area of data bases, and of developing a generative methodology [1] for the design of data base applications. The need to formalize concepts pertaining to the conceptual schema has been recognized and the subject has received considerable attention recently [6]. Generative data base specifications, using some form of grammar, have been proposed [3,4,11].

The choice of a generative formalism was influenced by the recognition of user-friendly interfaces as an important tool to improve the usefulness of data base systems, since they enlarge the class of potential users. Classes of user-friendly languages include English-like languages, graphic or bidimensional syntax languages and natural language interfaces. However a fourth class of user-friendly interfaces can still be devised. The basic idea is to define not just the conceptual schema of a data base application, but a complete formal language that reflects the jargon of the application area. This approach avoids the difficulties created by certain artificial language constructs; it also avoids the complexities of natural language interfaces and yet will look natural to users.

Besides this emphasis on the language aspect, generative methodologies are particularly appropriate to validate (test) specifications, since grammars can be used to concretely generate instances of the specified language or to parse what the specifier believes to be legitimate instances. In addition, translations between different representations can be done in a convenient way with grammars structured in two levels; roughly, one level is used to parse instances of the given representation and the other level to generate the corresponding instances in the new representation.

Our methodology is based on W-grammars [13], which are two-level grammars allowing the specification of both syntax (including the so-called 'context conditions') and semantics within a single formalism. Other definitional methods either concentrate on semantics or describe just the context-free aspects of a formal language [9]. W-grammars also provide mechanisms, to become apparent in later sec-

tions, facilitating modularization, which is a crucial feature since a data base is a complex object that should be specified in a stepwise way.

Our data base specifications will consist of several W-grammars incrementally defined, passing from declarative to procedural specifications. Validation and testing can be performed at each step. General aspects of data bases are distinguished from those belonging to the given application.

The paper is organized as follows. Section 2 gives some basic notions of W-grammars. Section 3 informally describes some data base concepts, which section 4 shows how to capture using W-grammars. Section 5 illustrates how W-grammars are used to define data base systems without referring to operations. Section 6 continues the process on the basis of fixed sets of application-oriented operations. Section 7 explains how W-grammars can be employed to investigate mappings between schemas. Finally section 8 contains conclusions and directions for future research.

2. BASIC NOTIONS OF W-GRAMMARS

Formally [8], a W-grammar is an 8-tuple $G = (M, S, T, z, MT, H, RM, RH)$, where:

- M is a finite set of metanotions (which are denoted by sequences of capital letters);
- S is a finite set of s-notions (which are denoted by sequences of lower-case letters);
- T is a finite set of terminals (which are denoted by sequences of lower-case letters between double quotes (following [10]) or by special indicated symbols);
(Sets M , S and T are pairwise disjoint)
- $z \in S^+$ is the start symbol;
- $MT \subseteq S$ is a finite set of metaterminals;
- $H \subseteq (M \cup S)^+$ is a finite set of hypernotions;
- $RM \subseteq M \times (M \cup MT)^*$ is a finite set of meta-rules (which are written as $X_0 :: X_1 X_2 \dots X_m$.);
- $RH \subseteq H \times (H \cup T)^*$ is a finite set of hyper-rules (which are written in the form $x_0 : x_1, x_2, \dots, x_n$.);
(Metarules with the same left-hand side may be combined using semicolons to separate the right-hand side alternatives, the same convention being used for hyperrules. The null sequence is denoted by 'e').

Financial help was provided by CNPq.

By applying metarules just like ordinary context-free rules, one can generate from a metanotion X sequences of metaterminals, called metaproductions of X . Now, consider a hyperrule $x_0 : x_1, x_2, \dots, x_k$.. Each x_i being an element of $(M \cup S)^+$ may contain occurrences of X . By taking a metaproduction y of X and consistently replacing each occurrence of X in the above hyperrule by y we obtain a new rule $x'_0 : x'_1, x'_2, \dots, x'_k$.. If we perform this process of uniform replacement on all metanotions occurring in the hyperrule we obtain a context-free production rule $x_0 : x_1, x_2, \dots, x_k$.., where each x_i is a sequence without metanotions. The language generated by a hypernotation h , $L(h)$, consists of all sequences of terminals derived from h by such production rules and the language generated by the W -grammar G is $L(z)$.

A predicate in a W -grammar is a hypernotation p with $L(p) = \{ \epsilon \}$. In other words, it either eventually produces the null sequence, after the application of one or more production rules, or else leads to a 'blind alley' (generating no terminal sequence).

W -grammars have been shown to be as powerful as type 0 grammars [12]. The correspondence between W -grammars and formal systems of logic has been investigated [8]. The language generated by a W -grammar coincides with the set of all objects of interest: that is, each finite sentence in the language will represent some object with the intended properties. Therefore we rule out by definition infinitary instances that seem to plague approaches based on first-order logic. For more details on W -grammars, see [2].

For data base specification, uniform replacement and the use of predicates have a particular importance, to express, respectively, parameterization [5] and integrity constraints involving different elements in the data base (non-local or 'context-sensitive' constraints). The two features together favor modular specifications, since they allow to relate separately specified elements.

3. INFORMAL DESCRIPTION OF DATA BASE CONCEPTS

The contents of a particular data base at some instant of time is called a state. More precisely, a state is a possibly empty finite set of facts. For specific data base applications only certain kinds of facts are admitted. For example, in an academic application, a fact may be that a course is being offered or that a student is taking a course.

It might occur that certain states do not adequately represent concrete situations, however. Hence, static constraints are imposed that rule out such states. In our running example we impose that:

static constraint : students can only take courses that are currently offered.

States may change as time flows. Moving from a state to another constitutes a transition. Valid transitions must involve pairs of states which are both valid (with respect to static

constraints) and, in addition, must obey the required transition constraints. In our example, the only transition constraint we impose is:

transition constraint : once a student starts taking some course, the number of courses that he takes cannot drop to zero (in the academic term being recorded).

One may assert or deny a fact. If a fact is denied it simply vanishes from the state, since only positive facts are kept. Assert, deny and create (an initial empty state) are primitive operations. Systems where primitive operations can be used directly, at the risk of violating constraints, are called open systems. As an alternative approach more apt to preserve constraints, we shall also consider systems with encapsulation, where only a fixed set of application-oriented operations is directly available. Such operations are associated with appropriate sequences of primitive operations, which are executed only if certain pre-conditions hold. In this way, the declarative specification of static and transition constraints is replaced by a procedural style, where constraints are 'embedded' in the operations.

The application operations in our example are: initiate academic term, offer course, cancel course, enroll student in course and transfer student from a course to another.

Thus far our discussion has been confined to the conceptual schema, describing the entire data base. Each user has an external schema, where only part of the information is available, perhaps re-organized according to the different needs of users. Different schemas are related by mappings (introduced with the ANSI/X3/SPARC architecture [6]). In our example, we define, for each student, an external schema that exhibits all courses offered and the set of courses he is currently taking, so that a student cannot see courses other students are taking through his view. Note that a whole family of schemas and their mappings is thus defined.

The kinds of facts belonging to a data base application, together with the application-oriented operations, convey part of the jargon of the application area. External schemas provide a further specialization of this jargon for each class of users.

4. STRATEGY

The four W -grammars in the next two sections have the property that all rules of each W -grammar are included in the next one. The last W -grammar, G_5 , will contain G_1 alone. Similar 'incremental' strategies have been used, for example, in [2,8].

The particular concept defined by each W -grammar corresponds to its start symbol (indicated by an arrow), which is, respectively, state, transition, execution (of sequences of primitive operations), single appexecution (execution of one application operation) and the several possible mappings.

Each of the concepts above involves previous concepts. Indeed, transitions involve pairs of states, executions achieve transitions, an application operation is executed as a sequence of primitive operations and mappings are obtained from (conceptual) states. This justifies our incremental design strategy: by incorporating the preceding rules, a W-grammar is led to check the properties of the concepts involved besides those of the new concept being introduced.

Note that, in each W-grammar, the concept is introduced by its first hyperrule with the form

$x : \dots X \dots$, predicates testing X .

We are separating the part contributing to the generation of terminal strings from the part performing the tests, which vanishes if the tests succeed. The principle of uniform replacement guarantees that the X tested is actually a copy of the other X .

At a later stage we may wish to reduce the objects generated by certain W-grammars or cease to check some properties. An especially important case corresponds to the application-oriented operations in G4. If they have been correctly designed, no integrity violation can result from their execution. Hence, after convincing ourselves of the correctness of their design, we may simply eliminate all validity tests from the first hyperrule of G4.

Another reduction, also involving an easy simplification of the first hyperrule of the respective W-grammar, can be done to compare the set of all transitions with those that can be brought about by one execution of primitive or application operations.

Yet another important design feature is the separation between general and specific metarules and hyperrules; allowing to distinguish aspects that are common to all data bases from those that characterize a given application. The former may be compared to a very basic idea of a 'data model', without any structural properties.

We shall resort to several widely-used 'auxiliary' constructs (see [10], for example for their precise definition) such as 'where', 'unless', 'contains' and 'is'. Additional auxiliary constructs will be introduced as needed. As usual, metarules of the form $X_1 :: X$, $X_2 :: X$, ..., $X_i :: X$, for every metanotation X , (like 'DB1' and 'DB2' in the next section) will be assumed. All predicates start with 'where' or 'unless'. We stress that sentences within double quotes are terminal and so will be, by convention, the special symbols ' \rightarrow ', '[', ']', '<', '>' and '/'.

5. DECLARATIVE SPECIFICATIONS

W-grammar G1 (below) defines the set of consistent data base states. An instance of $L(G1)$ is:

"offered(i50)" "takes(333|i24)" "offered(i24)"

The general metarules define the metanotation 'DB' as a possibly empty collection of facts. In turn, each fact consists of a name followed by a parenthesized sequence of values, separated by '|'. Values are combinations of letters and digits.

The specific metarules define the kinds of facts appearing in the data base application being specified. For each application fact, its name and the format of the values (domains) involved are given.

Application facts must conform to the syntax previously established for facts in general. This compliance is ensured by the combined effect of the two general hyperrules. The first hyperrule defines the start symbol, 'state', in terms of facts, whereas the second hyperrule converts into terminal notation (by enclosing between double quotes) only the facts allowed in the data base application. In addition, the hyperrules require that the entire state be valid and disallow duplicate occurrences of facts.

The static constraints of the data base application are imposed by the specific hyperrules. For the example application we adopted, for simplicity (and thus disregarding efficiency), the strategy of recursively scanning the given 'copy' of the state; there are no constraints on courses being offered, whilst, for courses that are taken, the fact that they are being offered must be present. By placing a fact between two possibly empty sub-collections of facts ('DB1' and 'DB2') in the left-hand side of the hyperrules, we provide the freedom to do the scanning in an appropriate order.

W-GRAMMAR G1

Metarules

general

DB :: DB FACT ; ϵ .
 FACT :: FACTNAME (VALUESEQ) .
 FACTNAME :: FACTNAME ALPHA ; ALPHA .
 VALUESEQ :: VALUESEQ | VALUE ; VALUE .
 VALUE :: VALUE SYMBOL ; SYMBOL .
 SYMBOL :: ALPHA ; DIGIT .
 ALPHA :: a ; b ; c ; d ; e ; f ; g ; h ;
 i ; j ; k ; l ; m ; n ; o ; p ;
 q ; r ; s ; t ; u ; v ; w ; x ;
 y ; z .
 DIGIT :: 0 ; 1 ; 2 ; 3 ; 4 ;
 5 ; 6 ; 7 ; 8 ; 9 .

specific

APPFACT :: offered (COURSEID) ;
 takes (SNUMB | COURSEID) .
 SNUMB :: DIGIT DIGIT DIGIT .
 COURSEID :: ALPHA DIGIT DIGIT .

Hyperrules

general

\rightarrow state : DB ,
 where valid DB .
 APPFACT DB : "APPFACT" , DB ,
 unless DB contains APPFACT .

specific

where valid
 DB1 offered(COURSEID) DB2 :
 where valid DB1 DB2 .
 where valid
 DB1 takes(SNUMB|COURSEID) DB2 :
 where DB1 DB2 contains
 offered(COURSEID) ,

where valid DB1 DB2 .
 where valid $\epsilon : \epsilon$.

W-grammar G2 defines transitions as pairs of states. An instance of L(G2) is:

```
"offered(i50)" "takes(333|i24)" "offered(i24)"
→
"offered(i50)" "offered(i24)" "takes(333|i50)"
```

The general hyperrule defines a valid transition as a pair of valid states (with respect to the static constraints inherited from G1) which, in addition, obeys the declared transition constraints. The latter requirement may have to be verified in two directions (forward and backward), which follows from the need to check certain facts appearing only either in the current or in the next state (recall that the negation of a fact is not represented explicitly).

The second specific hyperrule checks the single transition constraint of our academic data base. If a student is taking a course in the current state he must be taking some course (possibly the same) in the next state. For the academic data base example, forward checking is sufficient.

One may wonder whether a transition wherein the next state contains a student taking a course that is not offered is a valid transition, since there is no transition constraint excluding this possibility. Indeed, it is not a valid transition and it will not be generated by G2, since incorporating the static constraints from G1 in G2 ensures that valid transitions can only be defined between valid states.

W-GRAMMAR G2

Metarules

Those of G1

Hyperrules

general

Those of G1, and

→ transition :

DB1, →, DB2 ,
 where valid DB1 , where valid DB2 ,
 where forward valid DB1 to DB2 ,
 where backward valid DB2 from DB1 .

specific

Those of G1, and

where forward valid
 DB1 offered(COURSEID) DB2 to DB3 :
 where forward valid DB1 DB2 to DB3 .
 where forward valid
 DB1 takes(SNUMB|COURSEID1) DB2
 to DB3 :
 where DB3 contains
 takes(SNUMB|COURSEID2) ,
 where forward valid DB1 DB2 to DB3 .
 where forward valid ϵ to DB : ϵ .
 where backward valid DB2 from DB1 : ϵ .

6. PROCEDURAL SPECIFICATIONS

W-grammar G3 defines the application of sequences of primitive operations. An instance of L(G3) is:

```
[ "deny(takes(333|i24))"
  "assert(takes(333|i50))" ]
"offered(i50)" "takes(333|i24)"
```

```
"offered(i24)" →
"offered(i50)" "offered(i24)"
"takes(333|i50)"
```

The general metarules enumerate the primitive operations and introduce sequences of one or more of their occurrences.

The first general hyperrule defines an execution as a bracketed sequence of operations followed by a transition that can be achieved by the sequence. Static and transition constraints are checked; however they are not checked for intermediate states and transitions. The third hyperrule isolates each operation in the sequence so that we may verify its ability to perform an intermediate step.

The remaining hyperrules express the effect of the primitive operations. In order to avoid duplicates, 'assert' can be applied only if the fact in question is not already present. Assuming that duplicates cannot arise, 'deny' acts by removing (single) occurrences of facts. The present characterization of 'create' is really a simplification and does not convey the idea that it is a constant.

W-GRAMMAR G3

Metarules

general

Those of G2, and

OPSEQ :: OPSEQ OPERATION ; OPERATION .

OPERATION ::

create () ; assert (FACT) ;

deny (FACT) .

specific

Those of G2

Hyperrules

general

Those of G2, and

→ execution :

[, OPSEQ ,] , DB1 , → , DB2 ,
 where valid DB1 , where valid DB2 ,
 where forward valid DB1 to DB2 ,
 where backward valid DB2 from DB1 ,
 where achieved DB1 to DB2 by OPSEQ .
 OPERATION OPSEQ : "OPERATION" , OPSEQ .
 where achieved DB1 to DB3
 by OPERATION OPSEQ :
 where achieved DB1 to DB2
 by OPERATION ,
 where achieved DB2 to DB3 by OPSEQ .
 where achieved ϵ to ϵ by create() : ϵ .
 where achieved DB1 DB2 to DB1 FACT DB2
 by assert(FACT) :
 unless DB1 DB2 contains FACT .
 where achieved DB1 FACT DB2 to DB1 DB2
 by deny(FACT) : ϵ .

specific

Those of G2

W-grammar G4 defines the execution of a single application-oriented operation. An instance of L(G4) is:

```
<"transfer 333 from i24 to i50">
["deny(takes(333|i24))"
 "assert(takes(333|i50))"]
"offered(i50)" "takes(333|i24)"
"offered(i24)" →
"offered(i50)" "offered(i24)"
"takes(333|i50)"
```

The specific metarule enumerates the application operations of our example.

The first general hyperrule defines single executions of application operations by associating an application operation with the execution of a sequence of primitive operations. Besides checking the static and transition constraints and the effectiveness of the sequence to achieve the transition, the hyperrule verifies whether the sequence meets the pre-conditions required for its application to the first state in the transition.

The specific hyperrules give the sequences and corresponding pre-conditions for each application operation in our example data base. In more complex situations there may be more than one sequence for an application operation, corresponding to different pre-conditions.

Pre-conditions ensure that the execution of sequences will preserve all static and transition constraints. They also ensure that the execution will not be vacuous. As an example, consider the 'transfer' operation. The pre-conditions require that in the current state the original course really be taken but not the new one (which implicitly excludes the trivial situation where the two courses are the same), and that the new course be offered (in view of the static constraint). The transition constraint is preserved since 'transfer' will not alter the number of courses taken by the student.

The cancel operation invokes the locally defined 'void' predicate, which checks if the course to be cancelled is not taken by any student, since otherwise the static constraint would be violated. The universal quantification (on students) is emulated by an exhaustive search defined recursively.

W-GRAMMAR G4

Metarules

general

Those of G3

specific

Those of G3, and

APPOPERATION ::

- initiate academic term ;
- offer COURSEID ;
- cancel COURSEID ;
- enroll SNUMB in COURSEID ;
- transfer SNUMB from COURSEID to COURSEID .

Hyperrules

general

Those of G3, and

→ single appexecution :

- < , APPOPERATION , > , [, OPSEQ ,] ,
- DB1 , → , DB2 ,
- where valid DB1 , where valid DB2 ,
- where forward valid DB1 to DB2 ,
- where backward valid DB2 from DB1 ,
- where achieved DB1 to DB2 by OPSEQ ,
- where applicable APPOPERATION to DB1 as OPSEQ .

APPOPERATION : "APPOPERATION" .

specific

Those of G3, and

where applicable

initiate academic term to ε

```

as create() : ε .
where applicable
offer COURSEID to DB
as assert(offered(COURSEID)) :
unless DB contains offered(COURSEID) .
where applicable
cancel COURSEID to DB
as deny(offered(COURSEID)) :
where DB contains offered(COURSEID) ,
where void COURSEID in DB .
where applicable
enroll SNUMB in COURSEID to DB
as assert(takes(SNUMB|COURSEID)) :
unless DB contains
takes(SNUMB|COURSEID) ,
where DB contains offered(COURSEID) .
where applicable
transfer SNUMB from COURSEID1
to COURSEID2 to DB
as deny(takes(SNUMB|COURSEID1))
assert(takes(SNUMB|COURSEID2)) :
where DB contains
takes(SNUMB|COURSEID1) ,
unless DB contains
takes(SNUMB|COURSEID2) ,
where DB contains offered(COURSEID2) .
where void COURSEID1 in
takes(SNUMB|COURSEID2) DB :
unless COURSEID2 is COURSEID1 ,
where void COURSEID1 in DB .
where void COURSEID1 in
offered(COURSEID2) DB :
where void COURSEID1 in DB .
where void COURSEID in ε : ε .
    
```

7. MAPPINGS BETWEEN SCHEMAS

W-grammar G5 defines mappings between conceptual schema states and the corresponding external schema states of given users. Mappings are represented as triples (u,C,E) where u is a user identification, C is a conceptual state and E is the external state of user u corresponding to C. Students are the only class of users to be considered here. An instance of L(G5), is given below, noting that it refers to student 333:

```

"333" /
"offered(i50)" "takes(333|i24)"
"offered(i24)" "takes(333|i50)"
"takes(200|i24)" /
"offered(i50)" "offered(i24)"
"study-list({i24 i50})"
    
```

The specific metarules introduce sets of courses which will appear in the external fact 'study-list'. They also indicate that the external schemas to be specified are meant to be used by students; more precisely, a family of external schemas is specified, one for each student.

The first general hyperrule defines mappings as triples, as explained above, and requires that only valid conceptual states be considered.

The specific hyperrules evaluate the function mapping conceptual into external states of students. Courses being offered are included without change. Facts concerning other students taking courses are ignored. The last two hyperrules, to be applied after the others, group the courses taken by the student in question to

finally compose a single occurrence of 'study-list'; the set of courses may be empty.

W-GRAMMAR G5

Metarules

general

Those of G1

specific

Those of G1, and

COURSELIST :: COURSELIST COURSEID ; ε .
USER :: SNUMB .

Hyperrules

general

Those of G1, and

→ mapping :

USER , / , DB , / ,
external DB of USER ,
where valid DB .
USER : "USER" .

specific

Those of G1, and

external DB1 offered(COURSEID) DB2
of SNUMB :
"offered(COURSEID)" ,
external DB1 DB2 of SNUMB .
external DB1 takes(SNUMB2|COURSEID) DB2
of SNUMB1 :
external DB1 DB2 of SNUMB1 ,
unless SNUMB2 is SNUMB1 .
external COURSELIST
takes(SNUMB|COURSEID) DB of SNUMB :
external COURSELIST COURSEID DB
of SNUMB .
external COURSELIST of SNUMB :
"study-list({COURSELIST})" .

8. CONCLUSION

The main thrust of our approach is to provide a complete formal language whereby users can interact with the data base in a way that closely resembles how they talk about the application.

Generative specifications, like equational or axiomatic specifications, can be used for formally verifying data base properties. Yet their ability to parse or generate instances makes them particularly adequate for experimentation and testing. Among the aspects that can be checked are validity of states and transitions, ability of operations (primitive or application-oriented) to achieve transitions, adequacy of application operations to preserve static and transition constraints, and transference of the authorized information from conceptual to external states.

Intuition is favored by the incremental style used in introducing the W-grammars. By distinguishing specific rules from general rules, we can correspondingly reduce and discipline the task of the designer of particular data base applications, whose concern will be to supply appropriate specific rules. The natural characterization of mappings as translations is another contribution of the formalism.

A useful analogy can be established between metanotions in hyperrules and patterns, which suggests looking at the application of rules of W-grammars from a pattern-matching viewpoint, rather than the traditional generation of sets of production rules. The reader familiar with

SNOBOL may compare metarules to pattern definitions and hyperrules to pattern-matching statements. Also comparable are predicates in W-grammars and in SNOBOL.

Another point of practical interest deserves special mention, if we agree to use a different language for each data base application. In this case W-grammars (as also other comparable formalisms, such as attribute grammars and affix grammars) can provide the basis for compiler-generators [7, 9].

REFERENCES

- [1] R.L. Carvalho, B. A. Pereda, C. J. P. Lucena and T. S. E. Maibaum, Data specification methods, Proc. International Conference on Systems Methodology (1982).
- [2] J. C. Cleaveland and R. C. Uzgalis, Grammars for programming languages, Elsevier North-Holland (1977).
- [3] H. Ehrig and H. J. Kreowski, Applications of graph grammar theory to consistency, synchronization and scheduling in data base systems, Information Systems, vol. 5 (1980) 225-238.
- [4] A. L. Furtado, Transformations of data base structures, in Graph-grammars and their application to computer science and biology, V. Claus, H. Ehrig and G. Rozenberg (eds.), Springer (1979) 224-236.
- [5] G. H. Gonnet and F. W. Tompa, A constructive approach to the design of algorithms and their data structures, technical report CS-80-47, University of Waterloo (1980).
- [6] J. Griethuysen (ed.), Concepts and terminology for the conceptual schema and the information base, report from the ISO TC97/SC5/WG3 group (1982).
- [7] M. Griffiths, Introduction to compiler-compilers, in Compiler construction, F. L. Bauer and J. Eickel (eds.), Springer (1974) 356-365.
- [8] W. Hesse, A correspondence between W-grammars and formal systems of logic and its application to formal language description, technical report TUM-INFO-7727, Technische Universitaet Muenchen (1977).
- [9] F. G. Pagan, Formal Specification of programming languages, Prentice-Hall (1981).
- [10] J. E. L. Peck, Two-level grammars in action, in Information processing 74, J. L. Rosenfeld (ed.), North-Holland (1974) 317-321.
- [11] D. Ridjanovic and M. L. Brodie, Defining database dynamics with attribute grammars, Information Processing Letters, vol. 14, n. 3 (1982) 132-138.
- [12] M. Sintzoff, Existence of a van Wijngaarden syntax for every recursive enumerable set, Annales de la Soci t  Scientifique de Bruxelles (1967) 115-118.
- [13] A. van Wijngaarden et al (eds.), Revised report on the algorithmic language ALGOL 68, Acta Informatica, 5 (1975) 1-236.

ACKNOWLEDGEMENT

The authors are grateful to M. L. Brodie and W. Hesse for useful suggestions. Luiz Tucherman's help in preparing this version is appreciated.