DATA STRUCTURES AND GRAPH GRAMMARS

P.L. Della Vigna
C. Ghezzi
Istituto di Elettrotecnica ed Elettronica
Politecnico di Milano - Piazza L. da Vinci 32
20133 Milano - Italy

ABSTRACT

This paper is concerned with a formal model for data structure definition: data graph grammars (DGG's).

The model is claimed to give a rigorous documentation of data structures and to suit very properly program design via stepwise refinement.

Moreover it is possible to verify data structure correctness, with regard to their formal definition.

Last, attribute context-free data graph grammars (A-CF-DGG's) are introduced. A-CF-DGG's not only give a complete and clean description of data structures and algorithms running along data structures, but also can support an automatic synthesis of such algorithms.

KEY WORDS AND PHRASES

Data structure, abstraction, stepwise refinement, software reliability, correctness, program synthesis, context-free grammars, attribute grammars, parsing.

1. INTRODUCTION

Programming methodologies which can help in designing correct, easily modifiable, readable and portable software have become an important topic in computer science.

A widely accepted principle is that the quality of software can be considerably improved if the programmer can express his tasks in a free and natural way, without being concerned with details of the machine, which could force him to tailor his solution to some unnatural or unessential features.

Very high-level languages are an ambitious answer to these problems, but it has been argued they cannot exhaust all the needs of programmers. Moreover the serious problems of optimization which arise have not yet received a solution which allows to obtain a code of good quality.

Another attractive attack to this problem consists in successively decomposing a solution through "levels of abstraction". This means that the solution is initially specified by using an abstract machine whose operations and data tailor the problem to be solved. Whenever an abstraction is not directly supported by the language, it is recursively detailed until a level is reached which is directly supported by the system.

We feel that programming through levels of abstraction should not only be considered as a general philosophy to be divulged to non-believers, but should also inspire the design of computer-aided program development systems which allow to test, measure and modify programs at each stage of their stepwise refinement. Our research effort is presently in this area.

Quoting Liskov /1/, two kinds of abstraction are recognized to be useful in writing programs: "abstract operations and abstract data types . Abstract operations are naturally represented by subroutines or procedures, which permits them to be used abstractly (without knowledge of details of implementation). However, a program representation for abstract data types is not so obvious; the ordinary representation, a description of the way the objects of the type will occupy storage, forces the user of the type to be aware of implementation information".

These principles have inspired the definition and implementation of the CLU programming language/system, which is one of the most interesting research efforts towards the definition of a programming system supporting structured programming /2/ and modularity /3/.

We present here another model for data structures definition, abstraction and refinement which is based on graph grammars. In particular, we will show how the model can be used for clean documentation of the project and how it can support a computer assisted design of data structures,resulting in a considerable improvement of program reliability.

The reader who is interested in this topic is invited to read some related works which have appeared in the literature (/4/,/5/,/6/).

2. DATA GRAPH GRAMMARS

A data structure can be viewed abstractly as a set of objects connected by a network of access paths. Thus we can formally define a data graph over Σ (the node alphabet) and Δ (the link alphabet) as a triplet $D = (N, \phi, \psi)$, where N is the set of nodes, $\phi : N \to \Sigma$ and $\psi \subseteq N \times \Delta \times N$ are the node and link labelling functions respectively.

Let $\mathcal{G} = \{D | D$ is a data graph over $\Sigma, \Delta\}$; a data graph language $\mathfrak{D}$ over Σ, Δ is a subset of $\mathcal{G}$.

Two data graphs $D = (N_D, \phi_D, \psi_D)$ and $F = (N_F, \phi_F, \psi_F)$ are equivalent ($D \equiv F$) if a one-to-one equivalence function $e : N_D \to N_F$ can be found such that

1) $\phi_D(n) = \phi_F(e(n)), \forall n \in N_D$

2) $(n_1,a,n_2) \in \psi_D$ iff $(e(n_1), a, e(n_2)) \in \psi_F$

Languages of graphs, as an extention of the well-known languages of strings, have been studied by researchers in pattern-recognition in a number of papers (/7/,/8/,/9/,/10/,/11/). Applications to language definition and translation are explained in /12/ by Pratt, from whom we borrow some formalism.

Also if it appears that the theory of graph grammars may be developed along lines similar to the theory of string grammars, many problems remain yet to be studied; for example :

a) connections with graph-automata;
b) parsing;
c) definition of meaningful restricted classes of grammars.

We shall consider here mainly context-free graph grammars, without trying to give any answers to the questions above for which we refer to /10/, /11/. We shall rather restrict our attention to their use as a tool for data definition.

Let $D = (N, \phi, \psi)$ and $n_i, n_j \in N$; the interpretation of $\phi(n_i) = X_i$

and $\phi(n_j) = X_j$ is that object n_i and n_j are of type X_i and X_j respectively. $(n_i, Y, n_j) \in \psi$ means that object n_j can be accessed by n_i following the access link Y.

Links should not be considered as pointers, as well as nodes do not represent memory locations; rather they are abstract ways of referencing objects whose definition can be recursively given in terms of other links. In practice, for example, links could represent a simple reference or even a search algorithm.

A top-down design of a data structure should be considered as a set of operations which recursively detail the description of types and links. In particular, we shall concentrate here on data type refinements: link refinements could also be taken into account with minor changes to the model.

Node type refinements are represented here as production rules which describe the structure of a type in terms of lower level component data types.

Formally, a data graph grammar DGG is a 5-tuple $G = (\Sigma_n, \Sigma_t, \Delta, S, R)$, where the nonterminal node alphabet Σ_n, the terminal node alphabet Σ_t ($\Sigma = \Sigma_t \cup \Sigma_n$ is the total alphabet) and the link alphabet Δ are finite non-empty mutually disjoint sets, $S \in \Sigma_n$ is the starter (the axiom) and R is the set of production rules. Each element $r \in R$ is a 5-tuple $r = (A, D, I, O, W)$ such that

1) $A \in \Sigma_n$

2) $D = (N, \phi, \psi)$ is a connected graph over Σ and Δ (°)

3) $I \in N$ is the input node

4) $O \in N$ is the output node

5) $W \subseteq N$.

Before defining how productions are used to derive data graphs, we introduce the operation join which, applied to two graphs, gives a set of graphs as result. Let $D_1 = (N_1, \phi_1, \psi_1)$, $D_2 = (N_2, \phi_2, \psi_2)$, $D' = (N', \phi', \psi')$ be graphs over Σ, Δ and $\overline{N}_2$ a (possibly empty) subset of N_2. $D' \in$ join $(D_1, D_2, \overline{N}_2)$ if D' is equivalent to a graph $D = (N, \phi, \psi)$ such that :

1) $N = M_1 \cup M_2 \cup M_3$ where M_1, M_2, M_3 and M_4 are mutually disjoint set such that $N_1 = M_1 \cup M_2$, $N_2 = M_3 \cup M_4$, $\overline{N}_2 \subseteq M_3$, $\eta : N_2 \rightarrow M_2 \cup M_3$

(°) Let $D' = (N, \phi, \psi')$ be the undirected graph associated to D, such that $\psi' = \{(n_1, a, n_2) | (n_1, a, n_2) \in \psi \vee (n_2, a_1, n_1) \in \psi\}$: D is connected if D' is connected.

is a surjective application satisfying the conditions

a) $\eta(n)=n \ \forall n \in M_3$

b) $\eta(n) \in M_2 \ \forall n \in M_4$ such that $\phi_2(n) = \phi_1(\eta(n))$

2) a) $\phi(n) = \phi_1(n) \quad n \in N_1$

b) $\phi(n) = \phi_2(n) \quad n \in M_3$

3) $(n,a,m) \in \psi$ iff $n, m \in N_1$ and $(n,a,m) \in \psi_1$ or $n',m' \in N_2$ can be found such that $n = \eta(n')$, $m = \eta(m')$ and $(n', a, m') \in \psi_2$.

Intuitively, each resulting graph of join $(D_1, D_2, \overline{N}_2)$ can be viewed as a juxtaposition of D_1 and D_2 where nodes of D_2 not in $\overline{N}_2$ can be identified with nodes in D_1 with the same label.

If e is the equivalence function $e : N \to N'$, the joint function $j : N_1 \cup N_2 \to N'$ is defined as follows :

1) $j(n) = e(n) \ \forall n \in N_1$

2) $j(n) = e(\eta(n)) \ \forall n \in N_2$

The operation join is simple (s-join) if $\overline{N}_2 = N_2$. In such a case the result of the join operation is a single graph formed by the pair of graphs D_1 and D_2.

The derivation set $Y(G)$ defined by the data grammar G is a set of graphs over Σ and Δ which can be recursively defined as follows :

i. $Y(G)$ contains all the graphs D_o (the start graphs) equivalent to $(\{n\}, \phi, \varepsilon)$, where $\phi(n) = S$ and ε is the empty link labelling function

ii. let $D_1 = (N_1, \phi_1, \psi_1) \in Y(G)$, $\overline{n} \in N_1$, $\phi_1(\overline{n}) = A \in \Sigma_n$, $(A, D_2, I, O, W) \in R$, $D_2 = (N_2, \phi_2, \psi_2)$. $Y(G)$ contains also the graphs D' equivalent to $D = (N, \phi, \psi)$ constructed as follows:

1. let $D_1' = (N_1', \phi_1', \psi_1')$, where

   a) $N_1' = (N_1 - \{\overline{n}\}) \cup \{n_I, n_0\}$, $n_I, n_0 \notin N_1$

   b) - $\phi_1'(n) = \phi_1(n) \ \forall n \in N_1 - \{\overline{n}\}$
      - $\phi_1'(n_I) = \overline{X}$, $\overline{X} \in \Sigma$
      - $\phi_1'(n_0) = \bar{X}$, $\bar{X} \in \Sigma$

   c) - $(n_1,a,n_2) \in \psi_1' \ \forall n_1,n_2 \in N_1 - \{\overline{n}\}$ such that $(n_1, a, n_2) \in \psi_1$
      - $(n_0, a, n_I) \in \psi_1'$ if $(\overline{n}, a, \overline{n}) \in \psi_1$
      - $(n, a, n_I) \in \psi_1' \ \forall n \in N_1 - \{\overline{n}\}$ such that $(n, a, \overline{n}) \in \psi_1$.
      - $(n_0, a, n) \in \psi_1' \ \forall n \in N_1 - \{\overline{n}\}$ such that $(\overline{n}, a, n) \in \psi_1$.

2. let $D_1'' = (N_1'', \phi_1'', \psi_1'') \in$ join (D_1', D_2, W)

3. if j is the joint function $j : N_1' \cup N_2 \to N_1''$

then

a) $N = N_1'' - \{j(n_I), j(n_0)\}$

b) $\phi(n) = \phi_1''(n), \forall n \in N_1'' - \{j(n_I), j(n_0)\}$

c) $-(n_1,a,n_2) \in \psi \forall n_1,n_2 \in N_1'' - \{(j(n_I), j(n_0)\}$

$-(n,a,j(I)) \in \psi \forall n \in N_1'' - \{j(n_I), j(n_0)\}$ such that $(n,a,j(n_I)) \in \psi_1''$

$-(j(O), a,n) \in \psi \forall n \in N_1'' - \{j(n_I), j(n_0)\}$ such that $(j(n_0), a, n) \in \psi_1''$

$-(j(O), a, j(I)) \in \psi$ if $(j(n_0), a, j(n_I)) \in \psi_1''$

In general, the application of a rule to a graph D in Y(G) gives a result which depends on D, i.e. the operation is context dependent.

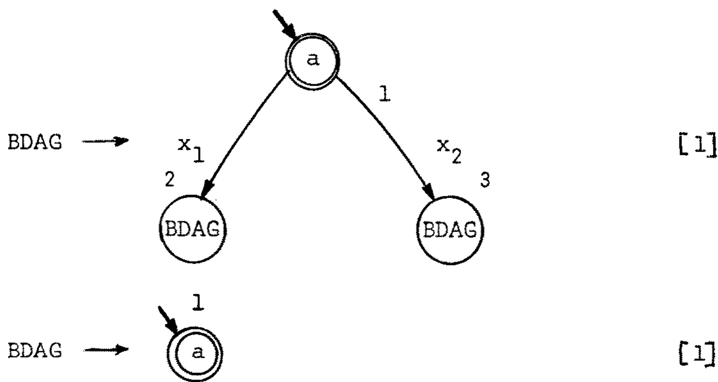
A DGG is a context-free data graph grammar (CF-DGG) if all the rules (A, D, I, O, W) where D = (N, ϕ, ψ) are such that W = N.

The data graph language (DGL) defined by a grammar G is :

$$L(G) = \{H | H = (N_H, \phi_H, \psi_H) \in Y(G) \wedge \phi_H(n) \in \Sigma_t \forall n \in N_H\}$$

Example 1

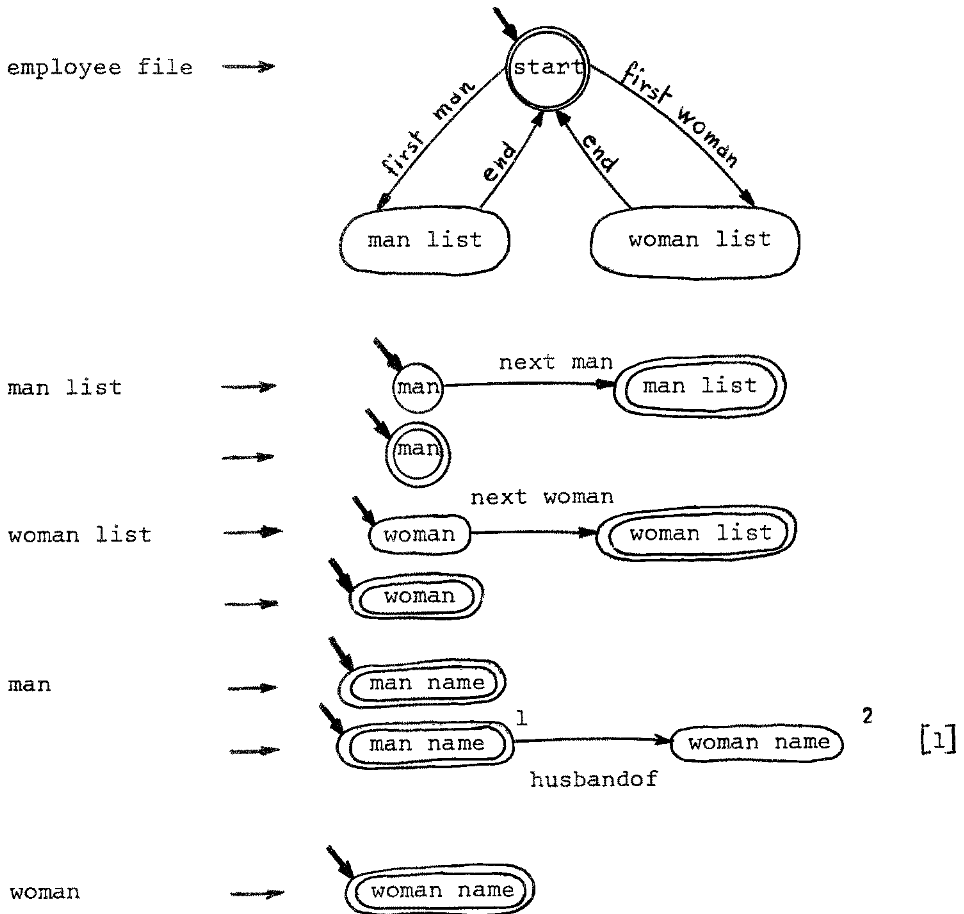
The following grammar (°) defines the set of binary directed acyclic graphs over $\Sigma = \{a\}$ and $\Delta = \{x_1, x_2\}$.



(°) The rule (A,(N, ϕ, ψ), I, O, W) is represented as A → (N, ϕ, ψ), where the input node is marked by an arrow, and the output node by a double circle.
The set of nodes W is bracketed by [and] . In the sequel, if no set W is listed, W = N is assumed.

Example 2

The following grammar generates the data structure shown in fig. 1, representing the employee file of a firm. Employees are grouped according to their sex.
If an employee is married, the name of the wife must be known; moreover, the system should record married couples of employees.



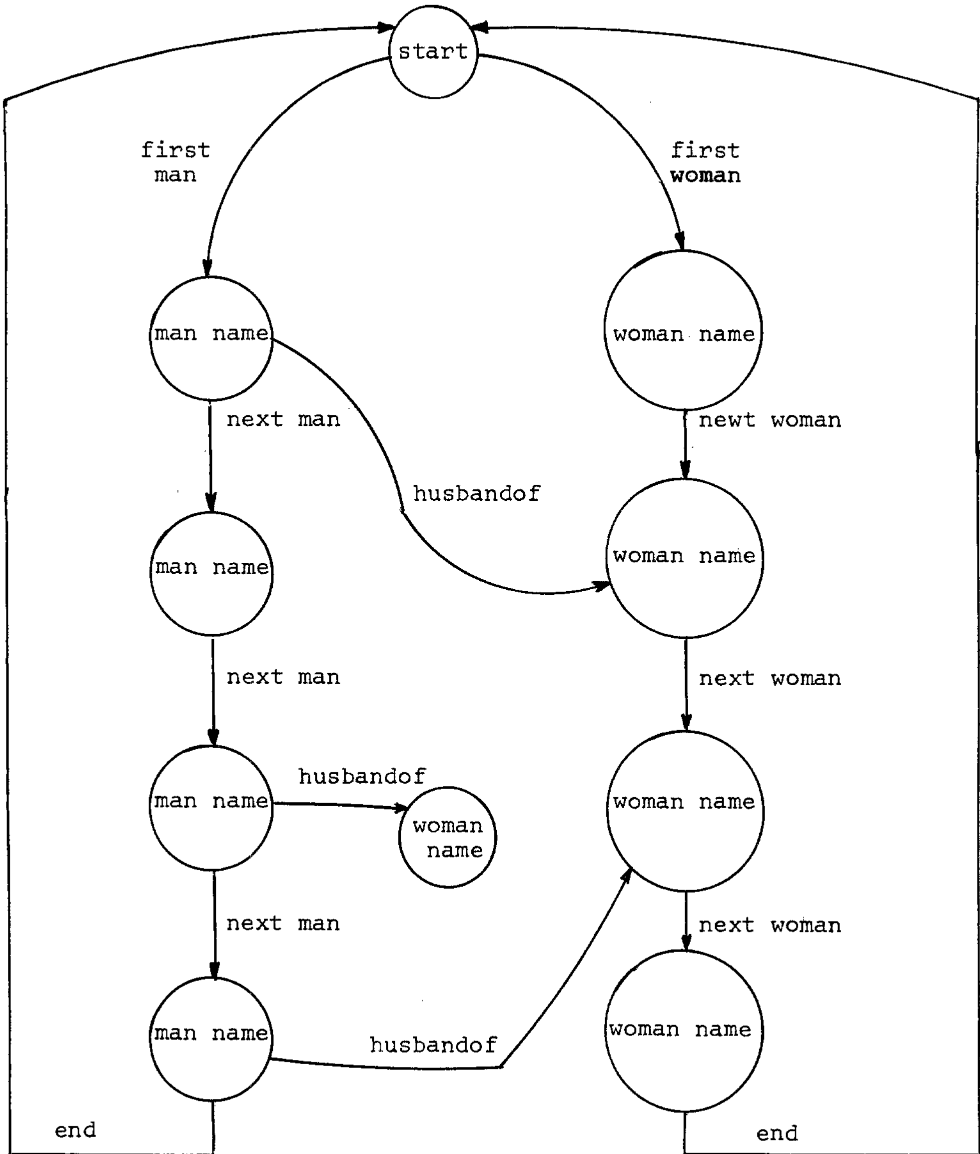


Figure 1

3. THE PARSING PROBLEM FOR DATA STRUCTURES

The formalism of DGG's should be viewed as a tool for describing data structures in a clean and rigorous way. This is a very important property, because it is well known that a clean documentation can greatly increase software reliability, as it becomes much more easy to prove program correctness and to maintain programs.

A number of questions naturally arise concerning the formal properties of DGG's. One of them regards the parsing problem for DGG's, i.e. the possibility of deciding whether a data structure is correct according to its formal definition. Several results on such problems for CF-DGG's are given in /11/ where suitable subclasses of CF-DGG's supporting efficient parsing algorithms are also studied.

As for the models described here it is possible to prove the following:

Proposition 1 - The parsing problem is decidable for DGG's having rules (A,D,I,O,W), where D = (N, ϕ, ψ), such that cardinality (W) $\geq$ 1.
In particular, it is decidable for CF-DGG's.

Proposition 2 - The parsing problem is undecidable for DGG's.

Moreover, given a CF-DGG, programs which test data structure correctness (data structure parsers) can be automatically constructed.

In what follows we shall restrict out attention to CF-DGG's

4. DATA GRAPH GRAMMARS AND TOP-DOWN PROGRAM DESIGN : AN EXAMPLE

In this section we give an example showing how DGG's can be used in the stepwise refinement of program construction.

Given a library organized in sections of different matters we develop an algorithm which computes ε, the set of empty sections. The data structure will be developed in parallel with the refinement of the search algorithm.

The program is written in an Algol-like language, with the following conventions for operations on the data structure: if A is an object of type α and a is a link exiting A, then :

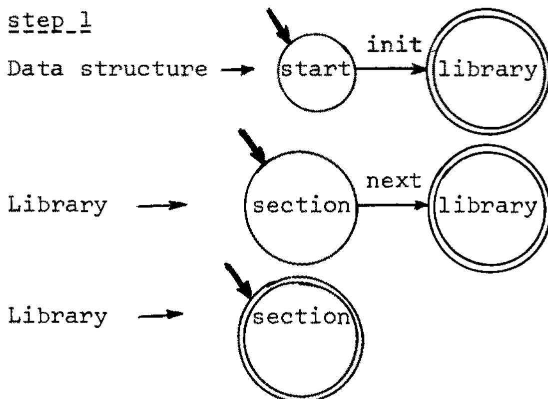
1. B:=a(A) means that the data structure control leaves object A following link a and the object reached by A under a is denoted by B;
2. is-link (A,a) is a boolean function which is true iff a link label-

led a leaves A;

3. if A denotes an object at step i whose type α is detailed by the rule $\alpha \to D$ at step i+k $(k \geq 1)$, then A denotes the input node of graph D at step i+k.

Data structure

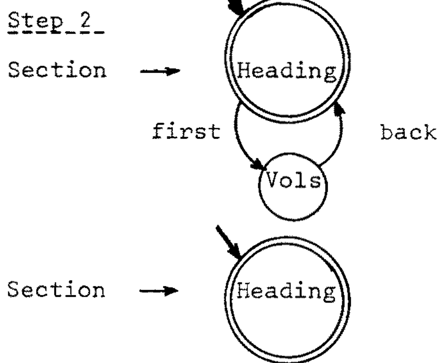
Program




```
/initially the current ob-
 ject is START/

Sect := init (START);
/successive integer numbers
are associated to successi-
ve sections/
i ← 0; ℰ ← ∅; scanned ← false;

repeat
   i ← i+1;
   if empty (Sect)
   then ℰ ← ℰ ∪ {i} ;
   if is-link (Sect, next)
   then Sect ← next (Sect)
   else scanned ← true
until scanned
```



We deatil empty(Sect)

```
Head ← Sect ;
if is-link (Head, first)
then empty ← false
else empty ← true
```

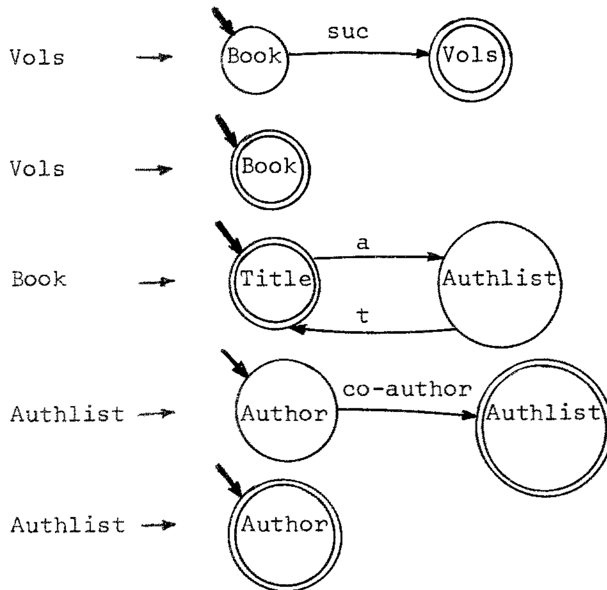
The reader should note that further refinement is required to detail step 4. The refinement implies:

1) definition and possible refinement of links;
2) concrete implementation of the data structure.

If we consider each link as a simple reference, no further refinement is required and we must simply map the abstract data structure onto the structures supported by the programming language.

On the other hand, we could consider links as invocations of algorithms yet to be detailed. For example, link next could extract from a secondary storage the file containing the next section.

On the other hand, even if the algorithm which computes ε does not require further refinements of the data structure, other queries about the data structure, such as the list of books written by an author all over the library, would require detailing the nonterminal Vols by means of the following productions



5. DATA GRAPH GRAMMARS AND PROGRAM SYNTHESIS

In this section we show how data graph grammars can be used for automatically synthesizing algorithms which perform computations running along the data structure.

We introduce here the formalism of Attribute-CF-DGG's which can be considered as an extension of similar concepts of /13/ /14/.

For each symbol $X \in \Sigma$ there is a set I(X) of inherited attributes and a set S(X) of synthesized attributes. The evaluation of the attributes is defined within the scope of a single production, by means of attributes rules. Attributes of the lefthand side nonterminal of the production are synthesized while attributes of the righthandside elements are inherited; attribute rules specify how a given attribute can be computed in terms of attributes of other elements in the same production.

As to the example described in section 4, we introduce the following synthesized attributes :

- ε, giving the set of empty sections;
- υ, giving the set of books written by a given AUTHOR;
- α, which is true iff AUTHOR has at least one book in the library;
- in, which is true iff AUTHOR is in the authorlist of a book;

and the inherited attribute

- n, which numbers each section of the library.

The Attribute-CF-DGG which represents the example is shown in fig. 2. The indices which appear in the attribute rules relate attributes to the elements of the productions.

Attributes can be evaluated by an algorithm which runs along the parse structure of the data structure; the values computed for the attributes of the starter of the grammar are the result of the data structure. In our example the evaluation of attribute ε of the nonterminal "Data structure" gives the same result as the program described in section 4.

The reader should note that using the formalism of Attribute-CF-DGG we simply specify, for each rule, how to compute an attribute, in terms of other attributes. In other words we do not give the formal specification of an algorithm, because the evaluation sequence of the attributes is not specified. The only constraint which must be satisfied by an effective algorithm is that an attribute can be evaluated only if the values of the attributes from which it depends are known.

It is possible to design an algorithm which, given the attribute-CF-DGG and a data structure satisfying the grammar, is able to find a suitable evaluation sequence (if it exists /13/) which allows:

a) to compute all the attributes in an interpretative scheme,

or

b) to generate an object program which computes the attributes.

In both cases, data types and operators used in attribute rules must be directly supported by the interpreter or by the programming language in which the object program is written.

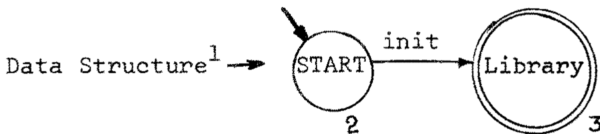
In the example, we have supposed that the object language supports data of type integer and boolean.

If we do not have a computer aided program design system, which is able to automatically construct a program which evaluates attributes, Attribute-CF-DGG's seem to play an useful role in giving a complete and clean documentation of data structures and algorithms which run along data structures.

It must be emphasized that this model is not suitable to represent operations which dynamically change data structures. Therefore whenever a data structure is modified it is necessary to re-parse the structure in order to obtain the new values of its attributes.

Attributes can also be used to impose restrictions on the class of data structures defined by a CF-DGG which cannot be specified by a CF-DGG or could be with a rather complicated grammar.

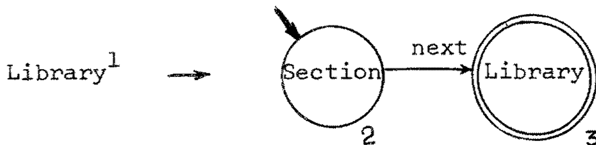
In the sequel we present an Attribute-CF-DGG for the example in Sec. 4



$n_3 \leftarrow 1 \quad v_1 \leftarrow v_3$

$\alpha_1 \leftarrow \alpha_3$

$\varepsilon_1 \leftarrow \varepsilon_3$

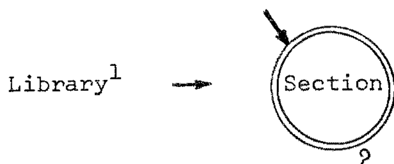


$n_2 \leftarrow n_1 \quad n_3 \leftarrow n_1+1$

$v_1 \leftarrow v_2 \cup v_3$

$\alpha_1 \leftarrow \underline{\text{if}}\ v_1 = \emptyset\ \underline{\text{then}}$
$\underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}}$

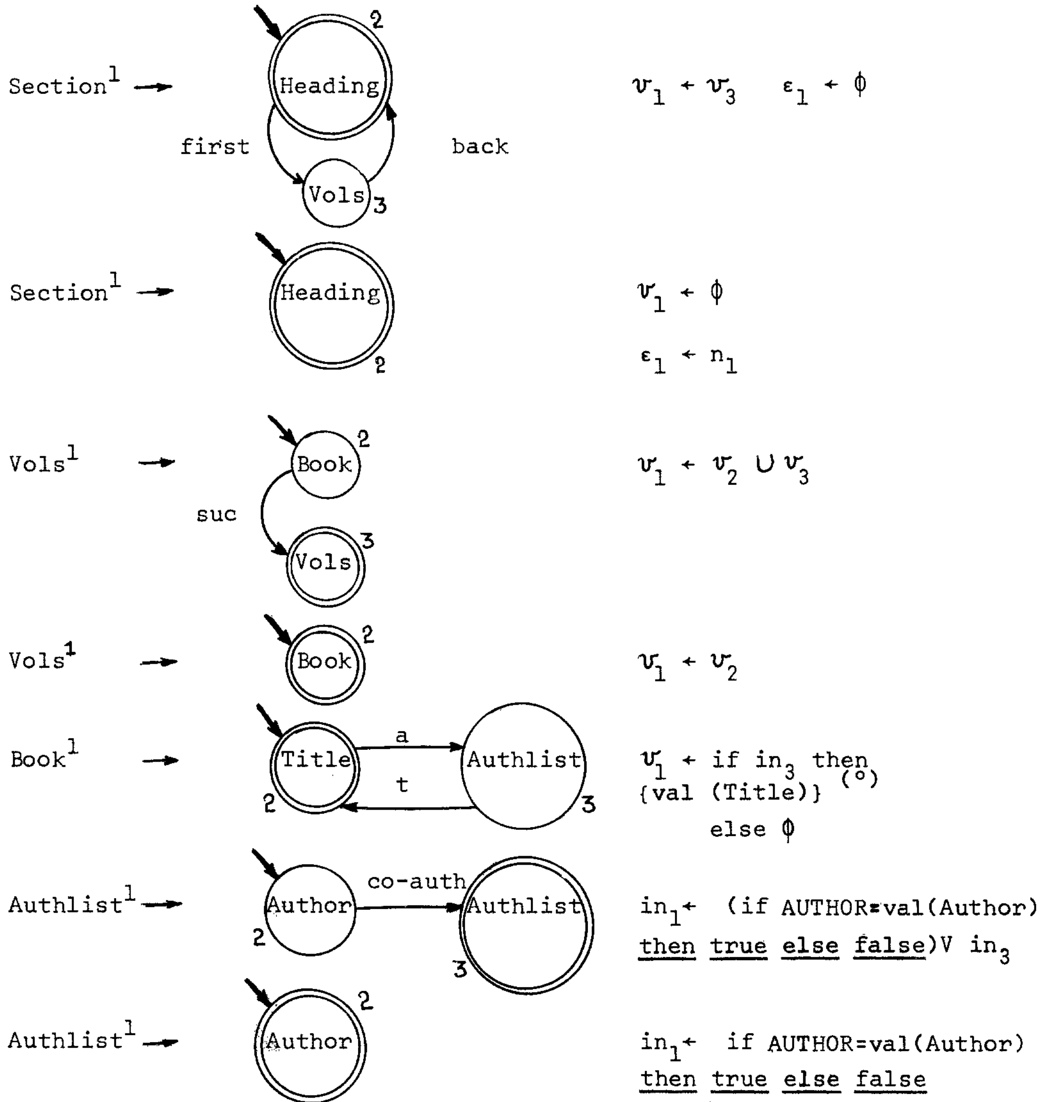
$\varepsilon_1 \leftarrow \varepsilon_2 \cup \varepsilon_3$



$n_2 \leftarrow n_1 \quad v_1 \leftarrow v_2$

$\alpha_1 \leftarrow \underline{\text{if}}\ v_1 = \emptyset\ \underline{\text{then}}$
$\underline{\text{true}}\ \underline{\text{else}}\ \underline{\text{false}}$

$\varepsilon_1 \leftarrow \varepsilon_2$



(°) Val (a) gives the value of the terminal a

6. CONCLUSION

In this paper we have given a formal definition of data graph grammars and we have discussed their relevance to data structure design.

In particular, we have restricted our attention to context-free data graph grammars, and we have shown that:

1) they give a complete and rigorous documentation of a data structure;
2) they describe in a clean and natural way stepwise refinements of data structures;
3) it is possible to verify data structure correctness, with regard to their formal (syntactic) definition;
4) it is possible to associate attribute rules to each production, so that algorithms which walk along a data structure can be automatically synthesized.

Further investigations are currently going on with regard to the following points:

1) dynamic change of data structures
2) data graph realization in a computer memory, with respect both to the automatic choice of efficient storage structures and restrictions on CF-DGG's which derive graphs more easily implementable /6/.

These points and a deeper insight into the practical relevance of the model are worth studying to support our belief that attribute data graph grammars can play an useful role in computer assisted program design.

REFERENCES

/1/ Liskov, B. "An introduction to CLU", Computation Structures Group Memo 136, MIT Project MAC, 1976.

/2/ Dahl, O.J., Dijkstra, E.W., Hoare C.A.R. "Structured programming" Academic Press New York, 1972.

/3/ Parnas, D.L. "On the criterion used in decomposing systems into modules", CACM 15, 12, 1053-58, 1972.

/4/ Earley, J. "Toward an understanding of data structures", CACM 14, 617-626, 1971.

/5/ Shneiderman, B., Scheuermann, P. "Structured data structures", CACM 17, 10, 583-587, 1974.

/6/ Rosengerg, A.L. "Addressable data graphs", JACM 19, 2, 309-340, 1972.

/7/ Pfaltz, J.L., Rosenfeld, A., "Web grammars" Proc. 1st Intl. Joint Conference on Artificial Intelligence, Washington, 609-19, 1969.

/8/ Montanari, U.C. "Separable graphs, planar graphs and web grammars", Information and Control, 16, 243-67, 1970.

/9/ Pavlidis, T. "Linear and context-free graph grammars", JACM 19, 11-22, 1972.

/10/ Milgram D.I. "Web automata", University of Maryland, Computer Science Center Technical rep. 271, 1973.

/11/ Della Vigna, P., Ghezzi, C. "Context-free graph grammars", Internal rep. 76-1, Istituto di Elettrotecnica ed Elettronica, Politecnico di Milano, IEEPM, 1976.

/12/ Pratt, T.W. "Pair grammars, graph languages and string to graph translations", JCSS 5, 560-595, 1971.

/13/ Knuth, D. "Semantics of context-free languages", Math. Systems Theory, 2, 127-145, 1968; Correction: Math.Systems Theory 5, 95-96, 1971.

/14/ Bochmann, G.V. "Semantic evaluated from left to right", CACM 2, 19, 55-63, 1976