PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 12/76

IMPLEMENTING A DATA DEFINITION FACILITY DRIVEN BY GRAPH GRAMMARS

by

P. B. Barroso

and

A. L. Furtado

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rua Marquês de São Vicente 225 — ZC 19

Rio de Janeiro — Brasil

Series: Monographs in Computer Science
and Computer Applications

Nº 12/76

IMPLEMENTING A DATA DEFINITION FACILITY DRIVEN BY GRAPH GRAMMARS*

by

P. B. Barroso+

and

A. L. Furtado

Series Editor: Roberto Lins de Carvalho

October 1976

* This research has been sponsored in part by FINEP under contract CT/370.

+ Present address: SERPRO - Rua Teixeira de Freitas, 31 - Rio de Janeiro - RJ - BRASIL

Copies may be requested from:

Rosane Teles Lins Castilho
Head, Setor de Documentação e Informação
Depto. de Informática - PUC/RJ
Rua Marquês de São Vicente, 209 - Gávea
20.000 - Rio de Janeiro - RJ - BRASIL

ABSTRACT

The implementation of a data definition facility based on graph transformations is discussed. The theory of graph grammars allows a precise characterization of these transformations, facilitating proofs of correctness. The implementation consists of an extension to PL/I, and utilizes the standard PL/I pre-processor.

KEY-WORDS

Data definition facility, graph grammars, extensible languages, PL/I, program correctness.

RESUMO

A implementação de uma facilidade de definição de dados baseada em transformações de grafos é discutida. A teoria de gramáticas de grafos permite uma caracterização precisa dessas transformações, facilitando provas de correção. A implementação consiste em uma extensão a PL/I, e utiliza o pré-processador integrante dessa linguagem

PALAVRAS CHAVE:

Facilidade de definição de dados, gramáticas de grafos, linguagens extensíveis, PL/I, correção de programas.

CONTENTS

1. INTRODUCTION

This paper discusses an implementation of a Data Definition Facility (DDF), directed by graph grammars. The efforts to incorporate powerful devices for data definition into modern programming languages (SIMULA 67, ALGOL 68, PASCAL, CLU) illustrate the present concern with this problem.

Early research on DDFs [1], as well as later work contributing to certain extensible languages (ECL) went along the following lines [2]:

1. a number of data types are provided as primitives;

2. a mechanism is introduced, which can be described as a set of constructors, i.e. specialized functions receiving as arguments various primitive (or previously so constructed) data types, and returning a new data type as result.

Other approaches are proposed in [3] and [4], and more recent work can be found in [5] and [6].

In most cases the creation of new data types requires the elaboration of routines and involves the manipulation of generic or (preferably) restricted pointers.

Of special interest here is the attempt towards a graph-theoretical formalization in [7]. Although the some author has produced a procedural implementation [8], he suggests that an implementation in terms of graph transformations would be advantageous.

The theory of graph transformations was made more precise through the studies on graph grammars [9]. In [10] a fully developed graph grammar formalism for characterizing data structures has been proposed. The present work is mainly based on the theoretical framework of the latter reference; the complete research is reported in [11].

The DDF has been implemented as an extension to PL/I, using the PL/I pre-processor and a number of run-time routines. It has a high degree of non-procedurality, relying on the visual intuition of graph transformations. Proofs of correctness use well-known graph grammar techniques, to be illustrated later.

PL/I pointers are employed but they are transparent to the user. Since the only way to alter the structure of (an instance of) a data structure is to invoke rules of the attached graph grammar, as defined through the DDF, a strict discipline is imposed on the manipulation of data structures.

2. THE DATA MODEL

As data model we shall use a class of graph grammar formally defined in [12], as informally described in the sequel by way of examples.

If we have a data structure consisting of a character string linked to a list with a variable number of binary data items, one possible representation is the one indicated in figure 1.

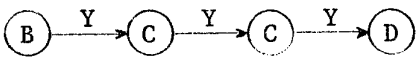


Fig.1: The graph of a data structure

where:

- the B node represents the character string;
- the C nodes represent the binary data items;
- the D node is a "trailer" of the list of binary data items;
- the Y edges link the nodes in a linear way.

Figure 2 shows a character string plus an empty list.

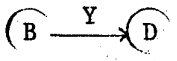


Fig.2: A "minimal" configuration

To add items to the list we can use the transformation* represented in figure 3

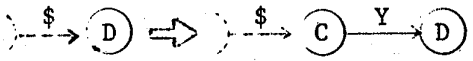


Fig.3: A transformation rule

which can be expressed as follows: "a D node is transformed into a C leading to a D through a Y edge, and the original 'context' of the left hand side D is now passed to the new C node." The context of a node is the set of edges pointing into or from the node.

Thus, two consecutive applications of this rule to the data structure of figure 2 would transform it into the configuration of figure 1.

Likewise, if we want to represent a binary tree with threads to the successor nodes in postorder traversal [13] we can do it through the graph of figure 4**.

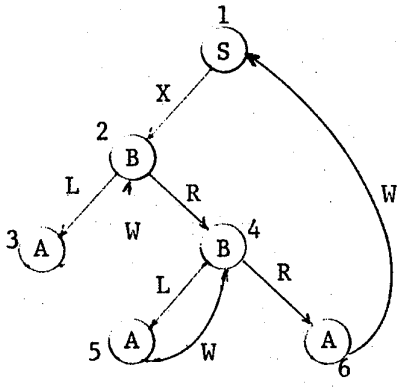


Fig.4: A binary tree with right threads

---

* To be called transformation rules, production rules, or simply rules.

** The positive integers enumerate the nodes; their use will be explained in section 3.

where:

- the S node is a header;
- the B nodes are data items;
- the A nodes are empty sub-trees.

A production rule that allows the inclusion of a new data item, at the same time making provision for further growth is given in figure 5.

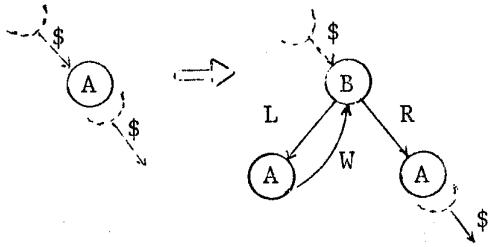


Fig. 5: A rule for expanding a tree

which transforms an empty sub-tree into a one node sub-tree (with two empty sub-trees), and transfers the outgoing context of the original empty sub-tree to the newly created right (empty) sub-tree, the incoming context being inherited by B.

If applied to the graph of figure 4, this rule could generate one of the graphs of figure 6 (among others).

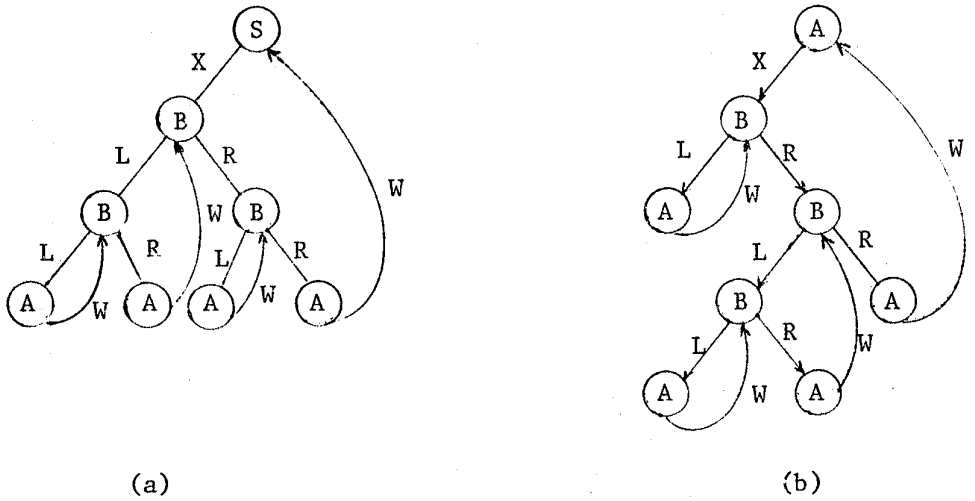


Fig.6: Two possible results of the application of a rule

Consequently, it is imperative that, for the application of a rule not to be ambiguous, a "base" node be indicated; this requires a method for locating nodes.

The chosen method involves a sequence of edge names (edge labels) from one special node to be called the root. This method has motivated the characterization of the class of graphs of interest, called rooted labelled digraphs.

In a rooted labelled digraph all nodes are reachable from the specified root, and in addition no two edges with the same label can leave any node.

Returning to the example, where the S node has been designated as the root, the rule of figure 5 applied to the graph of figure 4 with the sequence <X,L> yields configuration <a> of figure 6, whereas the sequence <X,R,L> yields configuration <b>.

3. DESCRIPTION OF THE DDF/DDL

Each new data type is described in the implemented DDF through a set of transformation rules (a grammar) capable of generating the desired set of graphs; also, another section of the DDF associates with the node labels a given PL/I (primitive) type, or a previously defined type, or still an "empty" type whenever the nodes with such labels have a purely structural meaning (the # DUMMY type). The remaining section merely lists the allowable edge labels.

Syntactically, a graph is "coded" by means of 4-tuples containing, for each node:

- its reference number, which is a positive integer uniquely associated with the node*;

---

* Notice that node labels are not unique.

- its incoming context;

- its label;

- its outgoing context.

Thus, the graph of figure 4 would be represented by:

<1 (W.6) S (X.2)>
<2 (X.1,W.3) B (R.4)>
<3 (L.2) A (W.2)>
<4 (R.2,W.5) B (L.5,R.6)>
<5 (L.4) A (W.4)>
<6 (R.4) A (W.1)>

The node references of the instances of a data type are internally created. Node references also appear in the coded rules as parameters, which means that when a rule is applied a correspondence between these node references and the (internally created) node references of the instance to be transformed is established.

The grammar for the trees of our example consists of two rules: the one in figure 5 and the rule in figure 7 below.

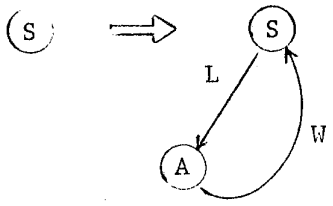


Fig.7: The other rule in the right-threaded tree grammar

We are now in position to show an example of the #TYPE statement, using the set of trees under discussion:

```
#TYPE((BTREE::
        NODES:  S  #DUMMY,
                A  #DUMMY,
                B  CHARACTER(12)VARYING;
        EDGES:  L,R,W,X;
        RULES:  1 <1()S()> → <2(W.3)S(X.3)>
                             <3(X.3)A(W.3)>ε

                2 <1($)A($)> → <2($1,W.3)B(L.3,R.4)>
                               <3(L.2)A(W.2)>
                               <4(R.2)A($1)>));
```

In order to declare variables of this new type we use the statement

```
#DECLARE ((BINARY_TREE BTREE));
```

which initializes the variable BINARY_TREE WITH <i( )S( )>, where i is some internally generated node reference.*

The two statements — #TYPE and #DECLARE — are in fact the DDF. The implementation also provides a Data Manipulation Languagem (DML), consisting of the statements to follow.

As we saw, in order to utilize the transformation mechanism it is necessary to specify sequences of edge labels, which is done as exemplified below:

```
#PATH(WAY = 'L' || 'R');
#PATH(W = 'X');
#PATH(P = );  /* EMPTY SEQUENCE */
#PATH(WAY =W || WAY);
```

the latter being equivalent given this sequence of statements to

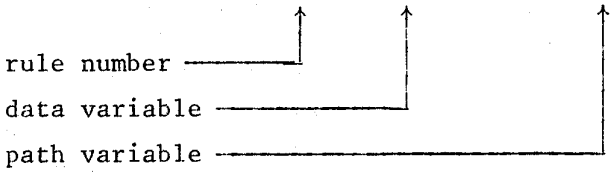
---

* Node label S is the start symbol of the grammar, besides marking in this example the root node.

```
#PATH(WAY = 'X' || 'L' || 'R');
```

Now, with the statement*

```
#APPLY(#1 : BINARY_TREE @ P);
```



we can generate the configuration shown in the right-hand side of figure 7, and then with

```
#APPLY (#2 : BINARY_TREE @ W);
```

the first "useful" (B-labelled) node can be added; other applications of rule 2, with appropriate assignments to the path variables, will continue this growth.†

When a (information bearing) node is created space is not automatically allocated for the values to be assigned to it. This is done through a statement

```
#ALLOCATE(BINARY_TREE @ W);
```

whereby a storage location is designated, taking into consideration the type of the node associated with its label.

---

* Note that the statement does not contain the name of the data type; thus the implementation ensures that only the rules of the associated grammar will be applied to the declared variable.

† Deletion rules could also be provided.

The actual placement of values in the allocated space is accomplished by statements like

```
#ASSIGN(BINARY_TREE @ W = X * Y);
```

some valid PL/I expression ——————————↑

and for retrieving a value one uses

```
#VALUE(VAR = BINARY_TREE @ W);
```

some valid PL/I variable ——————————↑

Other useful statements will be reviewed briefly:

```
#FREE(BINARY_TREE @ W);
```

releases the allocated space.

```
#LAB(VAR_CHAR = BINARY_TREE @ W);
```

assigns to the character string variable to the right of the "=" the label of the indicated node.

```
#DISPLAY(BINARY_TREE : GRAPH);
#DISPLAY(BTREE : GRAMMAR);
#DISPLAY(WAY : PATH);
#DISPLAY(GRAPHS);
```

are examples of the special output features provided by the implementation, which are particularly helpful for debugging; they cause the print-out, respectively, of the data structure instance graph, grammar, path variable, and all "active" data structure instance graphs of any type.

A more sophisticated and powerful version of the PATH statement is provided:

```
#PATH_V(W - @  BINARY_TREE = /<1($)B($,L.2)><2(L.2)B($)>
```

sub-graph pattern to be searched

```
/1 = 'NAME' ε  2 = 'M60' /.L);
```

required values (optional)

edge label to restrict the search (optional)

This pattern-directed search looks for a sub-graph matching the pattern and assigns the path variable (W) the path leading to its root node* as soon as one such sub-graph is found. One can also require that certain nodes of the sub-graph have specified values. Finally, it may be possible to avoid an exhaustive search by indicating that the root of the matching sub-graph lies along a path consisting of edges of a given label†; otherwise the search is performed in depth-first order |12| in the lexicographic order of the edge labels, the same criterion being used inside each (candidate) sub-graph.

---

* So these sub-graphs are actually rooted labelled sub-graphs; in fact the sub-graph patterns on both sides of production rules are also similarly constrained.

† The latter feature could be made more general. We are limiting ourselves to what is available in the current implementation.

4. THE ARCHITECTURE OF THE SYSTEM

The described DDF/DDL was implemented (both under PL/I-F and PL/I-X) as a PL/I extension, using the pre-processing features provides by this language, so that all its "statements" are actually regarded as procedures (macros) by the pre-processor.

The execution of these procedures causes the extraction from the program of the informations to be passed as arguments to a score of run-time routines. More precisely, the pre-processor replaces the "statements" by calls to the run-time routines.

So, in order to submit a program using the DDF/DMF one needs to have available:

- the body of the procedures, written in the pre-processor language (essentially an extended sub-set of PL/I); an appropriate %INCLUDE pre-processor statement will append this to the user's program;

- the run-time routines, to be link-edited; these would be in library mode.

Combinations of these run-time routines are used to achieve the effect of the several "statements". The same routine, for example, is shared by #APPLY and PATH_V for traversing a data structure instance graph and a pattern, while testing if a match occurs.

There is a global storage area, managed by run-time routines, where several elements needed by the system are kept, such as tables for the representation of the graphs, the several directories (for grammars, path variables, etc. and for their interrelationships) and so forth. The appropriate declarations are inserted in the user'program by the pre-processor when it finds the #INIT statement*.

---

* The user is allowed to specify several size information, which otherwise is determined by default.

The PL/I allocation mechanisms are used only for allocating and freeing space for storing values corresponding to information nodes. Based allocation is used, the pointer values being stored in the global area.

Several run-time checks and error messages are provided.

5. AN EXAMPLE

The example to be discussed involves two extended PL/I procedures for manipulating a symbol tables, as a compiler would use.

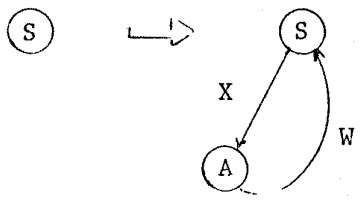
The first one is uded for constructing the table. At each invocation it receives a symbol to be installed and the number of the statement (in the program being compiled) where this symbols occurs, in order to enable the creation of a list of references to the symbols.

The second one prints the table in the alphabetic order of the symbols, each symbol being followed by its list of references.

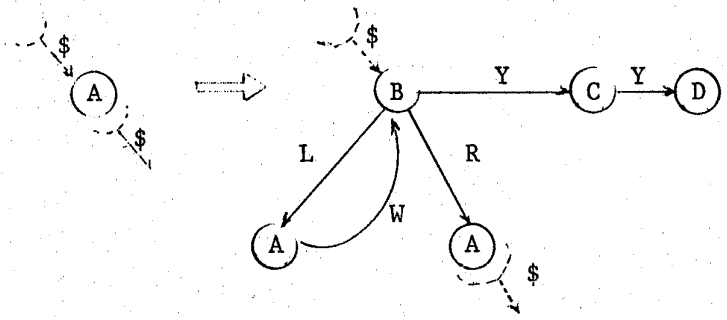
For the representation of the symbol table we use a data type which combines the two data types used thus far as examples. The binary tree will contain the symbols in alphabetic order (which in the tree corresponds to postorder traversal); the list of references will be a linear list linked to the node with the respective symbol.

As one would expect, the grammar will be, in a sense, the union of the two grammars previously introduced, as shown in figure 8.

1.



2.



3.

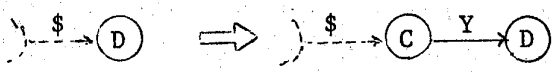


Fig.8: Grammar for the symbol table

As an illustration of the proof methodology, we give an outline of the proof that the above grammar generates exactly the intended set.

a. Only the valid configurations are generated.

This is shown by induction on the number of applications of the rules (stage of the generation).

Basis: if only one rule is applied it must be rule 1, which creates an empty tree.

Hypothesis: assume that after n applications only valid configurations have been generated.

Induction step: only rules 2 and 3 are applicable.

rule 3 - this rule is applicable at "trailers" (the D nodes) and it simply adds one element to the list;

rule 2 - this rule is applicable to empty sub-trees (the A nodes), and it transforms one such sub-tree into one B node leading to two empty sub-trees; the placement of the

thread caming from the left A node is correct by construction, whereas for the right A node the correctness is ensured by the correctness (by hypothesis) of its placement in the original empty sub-tree; the rule also initiates a one element list linked to the B node.

Note that rule 1 is applicable only once, because it requires that the S node have no context.

b. All the valid configurations are generated

This is shown by induction on the number of the nodes (size of the configurations). The proof is divided into two parts. For the size of the trees, we have:

Basis: the empty sub-tree is generated by applying rule 1;

Hypothesis: assume that all trees with m B nodes can be generated;

Induction step: any tree with m+1 nodes can be obtained by applying rule 2 to a tree with m B nodes, which is identical to the former except for the substitution of another B node* for an A node. In the desired tree with m+1 B nodes any B node with two attached A nodes can be arbitrarily chosen to be the missing one.

As to the size of the list, any number $k \geq 1$ of C nodes is acceptable. So, we have:

Basis: whenever a B node is created (by rule 2), a one element list is attached to it (one C node);

---

* With two A nodes and a one node list.

Hypothesis: assume that a given node of type B has a list with k C nodes;

Induction step: a list with k+1 C nodes can always be generated by applying rule 3 at the D node of a list with k C nodes.

The declarations and initializations in the main program and the two procedures are listed in the Appendix.

6. CONCLUSIONS AND FUTURE WORK

The proposed extension allows the definition of a broad class of data types through sets of transformation rules. Thus, not only the intended sets are defined but also the permissible ways of changing the structures are specified.

Once one has proved that the grammar corresponds exactly to the intended set, nor further proof of this fact is required throughout the program, where all that can be done to affect the structures is to apply rules of the associated grammar.

The notion of pointer is replaced by the notion of edge, which conveys the essential idea of accessibility within prescribed configurations.

One drawback of our approach is that for some more complex set the process of designing the grammar may require considerable skill. This may be partially remedied by providing a library of commonly (and possibly non-trivial) transformation. Future research could also try to develop a higher-level notation.

Another worthwhile effort would be to adapt the implementation to a graphics environment, where graphs and graph transformation rules would be represented directly.

APPENDIX

DECLARATIONS AND INITIALIZATIONS IN THE MAIN PROCEDURE.

```
#INIT
#TYPE((SIMBOL-TABLE::
                NODES:  S  #DUMMY,
                        A  #DUMMY,
                        D  #DUMMY,
                        B  CHAR(12)VAR,
                        C  FIXED BIN;
                EDGES:   L,R,W,X,Y;
                RULES:
1 <1()S()>→<2(W.3)S(X.3)><3(X.2)A(W.2)> ε

2 <1($)A($)>→<2($1,W.3)B(L.3,R.4,Y.5)>
             <3(L.2)A(W.2)><4(R.2)A($1)>
             <5(Y.2)C(Y.6)><6(Y.5)D()> ε

3 <1($)D()>→<2($1)C(Y.3)><3(Y.2)D()>  ));

#DECLARE((  TABLE  SIMBOL-TABLE  ));

#PATH(WAY=);

#APPLY(#1'::TABLE  @  WAY);
```

PROCEDURE THAT INSTALLS THE SYMBOL

```
INSTALA: PROCEDURE(SIMBOL, REFERENCE);
       DCL    LABEL  CHAR(6);
       DCL    SIMBOL CHAR(12) VAR;
       DCL    REFERENCE FIXED BIN;
       DCL    ONE_B     BIT(1) INITIAL('1'B);
       DCL    ALOC_SIMBOL CHAR(12)   VAR;
       #PATH(PATH = 'X');
       DO WHILE(ONE_B);
          #LAB(LABEL = TABLE @ PATH);
          IF LABEL = 'A'
          THEN DO;
               #APPLY(#2 : TABLE @ PATH);
               #ALLOCATE(TABLE @ PATH);
               #ASSIGN(TABLE @ PATH = SIMBOL);
               #PATH(PATH = PATH || 'Y');
               #ALLOCATE(TABLE @ PATH);
               #ASSIGN(TABLE @ PATH = REFERENCE);
               ONE_B = '0'B;
               END;
          ELSE DO;
               #VALUE(ALOC_SIMBOL=TABLE @ PATH);
               IF ALOC_SIMBOL = SIMBOL
               THEN DO;
                    #PATH(PATH=PATH|| (2)'Y');
                    DO WHILE (ONE_B);
                       #LAB(LABEL=TABLE @ PATH);
                       IF LABEL ¬= 'D'
                       THEN ONE_B = '0'B;
                       ELSE #PATH(PATH=PATH|| 'Y');
                    END;
               #APPLY(#3 : TABLE @ PATH);
               #ALLOCATE(TABLE @ PATH);
               #ASSIGN(TABLE @ PATH = REFERENCE);
               END;
```

```
          ELSE IF ALOC_SIMBOL > SIMBOL
               THEN  #PATH(PATH = PATH || 'L');
               ELSE  #PATH(PATH = PATH || 'R');
          END;
       END;
END INSTALA;
```

PROCEDURE THAT PRINTS - OUT THE SYMBOL TABLE

```
PRINT: PROCEDURE;
       PUT PAGE LIST('SIMBOL TABLE');
       DCL LABEL    CHAR(6);
       DCL SIMBOL   CHAR(12) VAR;
       DCL REFERENCE FIXED BIN;

       #PATH(PATH = 'X');
       DO WHILE('1'B);
          #LAB(LABEL = TABLE @ PATH);
          IF LABEL = 'A'
          THEN DO;
               #PATH(PATH = PATH ||'W');
               #LAB(LABEL = TABLE @ PATH);
               IF LABEL = 'S'
               THEN RETURN;
               #VALUE(SIMBOL = TABLE @ PATH);
               PUT SKIP LIST('SIMBOL', SIMBOL);
               #PATH(NEW = PATH);
               LABEL ='';
               DO WHILE(LABEL ¬= 'D');
                  #PATH(NEW = NEW ||'Y');
                  #LAB(LABEL = TABLE @ NEW);
                  IF LABEL = 'D'
                  THEN #PATH(PATH=PATH ||'R');
                  ELSE DO;
```

```
                    # VALUE(REFERENCE=TABLE  @  NEW);
                    PUT LIST(REFERENCE);
                    END;
               END;
               END;
          ELSE # PATH(PATH = PATH || 'L');
       END;
END PRINT;
```

REFERENCES

1 - STANDISH, T. A data definition facility for programming languages. Pittsburg, Pa.,Carnegie Institute of Technology, 1967. Ph.D. Thesis.

2 - WEGBREIT, B. The ECL programming system In: Proceeding AFIPS FJCC, 1972. vol. 39.

3 - JORRAND, P. Data types and extensible languages Paris, IBM France, Scientific Center, 1972.

4 - REYNOLDS, J. Gedanken: a simple typeless language based on the principle of completeness reference concept. CACM, 13; 308, May 1970.

5 - LISKOV, B & ZILLES,S. Specification techniques for data abstractions IEEE Transactions on Software Engineering , 1975.

6 - STANDISH, T. Data structures: an axiomatic approach. Cambridge, Mass., Bolt, Beranek & Newman, 1975.

7 - EARLEY, J. & CAIZERGUES, P. VERS manual, Berkeley, University of California, 1971.

8 - EARLEY, J. Towards an understanding of data structures. CACM, 14; 617, oct. 1971.

9 - PFALTZ, J. & ROSENFELD, A. - Web grammars. College Park, University of Maryland, 1969.

10 - GOTLIEB, C. & FURTADO,A.L. - Data schemata based on directed graphs, Toronto, University of Toronto, 1974.

11 - BARROSO, P. Discussão e implementação de um formalismo para definição de tipos de dados. Rio de Janeiro, PUC, 1975. M.Sc. Thesis

12 - GOTLIEB. C. Data structure definition Toronto, University of Toronto, 1972

13 - KNUTH, D. The art of computer programming. Reading, Mass., Addison- Wesley, 1972 .

14 - TARJAN, R.E. Depht first search and linear graph algorithms , Working paper Stanford University