Technical Report CS73008-R

FOL: A LANGUAGE FOR IMPLEMENTING FILE ORGANIZATIONS FOR INFORMATION STORAGE AND RETRIEVAL SYSTEMS

Billy G. Claybrook

November 1973

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061.

ABSTRACT

The language FOL is described. FOL facilitates the implementation of file organizations for IS & R systems. FOL is implemented in a list processing language LPL. Files in FOL are interepreted as a list of records, where each record is equivalent to a node structure. A description of LPL is also included.

I. INTRODUCTION.

This paper describes a language (FOL) that facilitates the implementation of file organizations for Information Storage and Retrieval (IS & R) systems. In many IS & R applications the designer, initially, does not know how best to organize his data base for efficient retrieval and ease of updating and modifying. We feel that FOL will provide the designer with the tools for implementing, quickly, different file organizations during the experimentation stages of data base design.

FOL allows the user to describe the structure of records in a file as nodes in a list. FOL provides an easy and illustrative scheme for describing the format of records in a file. The technique for record description is similar to that used in COBOL [1], but FOL has some additional features that allow record description to be more flexible.

FOL is implemented in a list processing language (LPL). LPL allows any type of node structure to be defined and thus appears to be a convenient language for implementing FOL. The advent of virtual memory machines allows this approach to the development and implementation of file organizations to be feasible. The designer can proceed with the important aspects of file organization design and omit much of the drudgery of file system programming.

The primary purposes of this paper are: (1) to present a brief description of FOL, (2) to describe its intended uses, and (3) to describe the basic concepts involved in its development. Since we are primarily interested in presenting concepts and ideas, we do not include a complete description of the syntax of FOL; however, examples are given and comments on the syntax are included where appropriate.

In Section II, we discuss the design of LPL to provide the reader with

the background for the discussion of FOL. And in Section III, we provide a description of FOL and give some examples of its use.

II. DESCRIPTION OF LPL.

This section describes LPL and presents the basic ideas in its design and implementation. LPL has been used in several algebraic manipulation programs, e.g. writing an interpreter for the first-order predicate calculus language, polynomial manipulations, and binary tree processing.

Overview of LPL

The initial implementation of LPL is in PL/1 [2] as an extension to PL/1. Programs written in LPL are scanned during a pre-processor pass and translated into a PL/1 program compiled by the PL/1 compiler. LPL requires the use of dynamic storage allocation so the choice of PL/1 was natural for a first implementation (this implementation was possible in PL/1 only after some tricky and imaginative uses of PL/1 based structures).

LPL differs from other list processing languages, e.g. SLIP [3] and LISP [4], because it allows different types of nodes, that can vary in size and structure, to be defined by the user. This allows the user to define nodes that are natural for his applications, thus allowing him to concentrate on ideas and avoid the clumsiness of trying to determine a way to represent a structure in a less flexible list processing language.

Description of LPL

LPL allows any number of node types to be defined by the user. LPL uses a template (for each type of node) to describe the attributes of each node

type. The node structure in LPL is developed to facilitate garbage collection, copying of lists, etc. Each node in LPL can have the following fields ((1)-(4) are common to every node):

(1) a type field (TYPE) that contains a pointer to the template describing this type of node,

(2) an activity bit (ACT) that is used to indicate whether a node is active (appears in a user list) or inactive (does not appear in a user list and can be freed by the RELEASE command).

(3) a copy bit (COPY) for use in copying lists, especially recursive lists,

(4) two link fields, MLP and MRP, that link all nodes allocated in a doubly linked "super" list (this list is processed by the garbage collector to release inactive nodes), and

(5) the other six entries are optional and any of them can be included in the node by using the DEFINE statement discussed below (these entries define the type(s) of values and number of link fields the user desires in a node structure).

Before we discuss the format of LPL statements, we feel that two other comments are appropriate. First, LPL is designed so that a considerable amount of error checking is performed for the user. Secondly, the user cannot access any of the node entries (1) - (4) described above; instead only the LPL programming system can manipulate them.

The following attributes are available in LPL (all normal PL/1 data types are also available):

NODETYPE - Attribute of a variable used for representing the type of a node

REAL - Standard definition

INTEGER - Standard definition

ALPHA - Attribute of a variable having an alphanumeric character string as its value

IDENT - Attribute of a variable having the identification attribute (e.g. the ID field in SLIP)

RFCNT - Reference count attribute

POINTER - Indicates the link or pointer attribute

LIST - Declares a variable to be a pointer to a list

STACK - Declares a variable to be a pointer to a stack

QUEUE - Declares a variable to be a pointer to a queue.

A detailed discussion of the LIST, STACK,and QUEUE attributes follow in a later subsection.

Definition of a Node Structure

The DEFINE statement in LPL allows the user to describe the nodes, and hence their structure, that he will use in an application. The format of the DEFINE statement is illustrated in the following example:

DEFINE NODE (ITYP, RV(I), IV(J), CV(K), ID(L), RC(M), LK(N)),

where the variables in this argument list must appear in a declaration statement such as the one below (the declaration statement must precede the DEFINE statement):

```
DCL     ITYP  NODETYPE,
          RV REAL,
          IV INTEGER,
          CV ALPHA(20),
          ID IDENT,
          RC RFCNT,
          LK POINTER;
```

The mode of the variables in the declaration statement indicate to the DEFINE statement the attributes of the fields to be included in a node of type ITYP. Not all of the seven parameters must appear in the DEFINE statement. Only those parameters necessary to describe the structure of a node are included.

Let us look at the DEFINE statement in detail. ITYP has attribute NODETYPE, thus indicating the type of node being defined (the value of ITYP is an integer quantity). Each node can have three kinds of values REAL, INTEGER, or ALPHA.

The user selects only those required for a node of type ITYP. The subscripts I, J, and K are the number of REAL, INTEGER, and ALPHA values in a node, respectively. The number of values can be variable or constant. ID(L), RC(M), and LK(N) are the identifier, reference count, and link entries, respectively, with L, M, and N being the number of each such fields in a node. It is difficult to visualize a node that requires more than one identifier or reference count field, but the option exists.

As an example the following DEFINE statement

```
DEFINE NODE (ITYP, IV(2), CV(3), LK(2))
```

describes a node of type ITYP that has two integer values, three alphanumberic string values with 20 characters each, and two links or pointers. The entries in the DEFINE statement can appear in any order.

LPL Statement Forms

The format of each LPL statement (with the exception of the DEFINE statement) is given and briefly described below (each LPL statement is delimited by a semicolon). Nodes pointed to by P, where P has attribute POINTER, are designated as node P; and [ ] indicates the contents are optional.

1. GET P,I; - Allocates a node P of type I.
2. DELETE P,T[,R]; - Deletes node P in list T, P follows node R.
3. INSERT P,T,R; - Inserts node P in list T after node R.
4. RELEASE P; - Frees node P, this statement succeeds only when the activity bit for node P is turned off and the reference count for node P is zero.
5. ENTER expression,T; - Enters the value of expression into T (T must be a STACK or QUEUE).
6. REMOVE X,T; - Remove an entry from T (T must be a STACK or QUEUE) and place it in X.

7. RV(N),P=expression; - Sets the Nth (N is an integer) real value of node P equal to the value of expression (the attribute of a value in node P can be REAL, INTEGER, ALPHA, IDENT, RFCNT, or POINTER).

8. X=RV(N),P; - X is the Nth REAL value in node P (a value can have attribute REAL, INTEGER, ALPHA, IDENT, RFCNT, or POINTER).

9. COPY S,T; - Makes a copy of list S in list T.

10. P=N,T; - P is the pointer to the Nth node in list T.

11. ERASE T; - Erase list T (the list will be erased only if every node has its activity bit turned off and its reference count is zero.

12. CONCATENATE S,T; - Concatenate lists S and T in the order given.

13. SPLIT T,P; - Split list T at node P.

14. SPLIT T,N,P; - Split list T at the Nth node, P is the pointer to the new list.

15. P, {SINGLY / DOUBLY / L-R / MULTI} ; - Declares node P to be a singly-linked, doubly-linked, left-right linked, or multi-linked (more than two links) node [5].

16. CREATE T {STACK / QUEUE / LIST} {SINGLY / DOUBLY / L-R / MULTI} {REAL / INTEGER / ALPHA / POINTER} ;- The modes REAL, INTEGER, ALPHA, POINTER apply only to STACKS and QUEUES.

17. COLLECT; - Invokes the garbage collector (garbage collection is also done automatically by LPL).

Some of the LPL statements require a more detailed description. The GET statement allocates a node of a specified type. During the allocation of this node the template for this node type is interrogated to determine the node's structure. The DELETE and INSERT statements cause the deletion (insertion) of a node from (into) a list. These statements automatically handle nodes that are SINGLY, DOUBLY or L-R (left-right) linked. For multilinked nodes, the

user must specify the specific pointers to modify during insertion or deletion. The INSERT statement sets the activity bit in a node, and the DELETE instruction resets the activity bit.

The CREATE statement (#16) allows the user to declare a variable to be a pointer to a LIST, STACK, or QUEUE. Any POINTER variable can also point to a list, stack, or queue. However, the CREATE statement declares explicitly what type of structure is pointed to and also indicates the format of the list structure, e.g. singly-linked, doubly-linked, etc. All nodes that appear in these structures must agree with the format given in the CREATE statement. Pointers to lists that have a nonhomogeneous mixture of nodes should not be referenced in a CREATE statement.

The CREATE statement also implies that the top of a STACK and the front and rear of a QUEUE are automatically set as new entries are added to them. The CREATE statement for a STACK causes a structure, pointed to by T, to be set up. This structure contains a pointer to the stack; this pointer (initially NULL) is set as entries are added to it and removed from it. The variable T in the CREATE statement for a QUEUE points, not directly to the queue, but to a structure that contains the front and rear pointers to the queue. The variable T in the CREATE statement for a LIST points to a list head that contains an identifier (the QUEUE and STACK structures also contain an identifier) indicating the format of the list and also a pointer to the first node in the list.

Associated with STACK structures is a function TOP(T) that returns as its value a pointer to the top of STACK T. Also FRONT(T) and REAR(T) are functions that return as values pointers to the front and rear, respectively, of QUEUE T.

III. DESCRIPTION OF FOL

FOL is an extension of LPL. FOL statements are translated into LPL by a

preprocessor and executed. Again, no attempt is made here to define all of the FOL syntax; instead, we introduce the reader to the basic concepts of FOL. FOL allows the user to describe the structure of records in a file, and hence the file, in a descriptive manner similar to that of COBOL. However, FOL allows the structure of a record to change more readily than COBOL. Although COBOL allows a file to have different types of records, the size of these records cannot be easily changed and administered as FOL allows. Also, since FOL represents records in a file as nodes in a list, no input/output problems are encountered directly by the user.

FOL allows the user to describe his own file organizations with a minimum of effort. Most programming systems support only a few standard file organizations, e.g. sequential, indexed-sequential, and direct. However, some IS & R applications require multilist, inverted, or other more esoteric file organizations to allow efficient retrieval of information. FOL is useful because it allows the user to define his own file structure, vary record sizes, and program his own methods for updating and modifying the files. Techniques for updating and modifying the files are relatively simple since a file is interpreted as a list of records.

The implementation of FOL in LPL makes the approach to file organization design described in this paper feasible. LPL is the basis for FOL and allows the user freedom and convenience in manipulating files.

File Description in FOL.

A file is described in FOL in the following manner:

```
DCL  X FILE MULTI,
     X CONSISTS OF[quant0] RECORDS Y,
     Y CONSISTS OF[quant1] FIELDS Z1,Z2,Z3,Z4
     Z1 CONSISTS OF[quant2] KEYS W ALPHA,
```

```
 W CONSISTS OF[quant3] BYTES;
Z2 CONSISTS OF[quant4] LINKS P POINTER;
Z3 CONSISTS OF[quant5] DATAREA D REAL,
 D CONSISTS OF[quant6] BYTES;
Z4 CONSISTS OF[quant7] DATAREA E ALPHA,
 E CONSISTS OF[quant8] BYTES;
```

The above description of file X can be interpreted as a tree structure (similar to structures in PL/1). The entries that represent the terminal nodes of the tree are terminated by semicolons (the terminal nodes are W's, P's, D's, and E's). Other entries in the file description are separated by commas. The words underlined in the declaration are descriptive words. The user can add his own descriptive words by using the function DESCRIPT (descriptive word). DESCRIPT adds user descriptive words to a table used during the preprocessor scan.

FILE is a FOL attribute and MULTI is a user supplied name for the file organization. This name can be used by the programmer for associating his update and modification routines with the file organization named MULTI.

The $quant_i$ in the description of file X are optional in some entries (they must appear in the lowest level entries such as W, P, D, and E). However, if a $quant_i$ appears in the declaration, it must be an integer constant or variable. A fixed record structure can be indicated by giving $quant_1$-$quant_8$ constant values (rarely does the user care to specify $quant_0$ a priori); in this case the user can allocate a record by executing ALLOC Y. If he desires five records, he can specify ALLOC Y(5).

If $quant_i$ is a variable quantity, then its value must be set prior to allocating a record or file. If $quant_i$ is not specified for FIELDS Z1, Z2, Z3, and Z4 above, then the following sequence must be given (the constants are set only for this example):

```
ALLOC Z1(3); ALLOC Z2(3); ALLOC Z3;
ALLOC Z4;
ALLOC Y;
```

This sequence of statements allocates a record with three keys, three link fields, one real data area, and one alphanumeric data area.

The above discussion for allocating records is very flexible because it allows the user freedom in describing the record structure. As an example of the convenience of this method for describing a file organization consider the following task - adding a key to an existing record in a multilist organization. A record with the proper structure is allocated, and the information in the old record is copied along with the new key (and presumably a corresponding link field) into the new record. The old record is deleted from the list and the new record is inserted into the list in its place. These operations are easily performed within the scope of LPL.

In the description of FOL given above, we have made no attempt to provide the user with "canned" organizations, but instead we have attempted to provide him with the facility for describing his own organizations.

IV. SUMMARY

Since LPL is an extension of PL/1 and FOL is a slight extension of LPL (the only current extension being the capability to describe the organization of a file as a list of records), the user has a powerful language with which to work. However, FOL is primarily designed for implementing file organizations and for providing the facilities, through LPL, for updating and modifying files. A possible extension to FOL is a means for allowing the user to describe the structure of directories in a manner similar to that used in describing files.

REFERENCES

1. Fiengold, Carl. Fundamentals of COBOL Programming, W.C. Brown Company, Dubuque, Iowa, 1973, 691 pp.

2. IBM System/360 PL/1(F) Language Reference Manual, GC28-8201-4.

3. Smith, D.K. "An Introduction to the List-Processing Language SLIP", Programming Systems and Languages, Rosen, Saul (ed.), McGraw-Hill, New York, 1967, pp. 393-417.

4. McCarthy, John, Abrahams, Paul, Edwards, D.J., Hart, T.P., and Levin, M.I. LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Mass. 1969, 106 pp.

5. Knuth, D.E. The Art of Computer Programming, Vol. I, Addison-Wesley, Reading, Mass., 1968, 634 pp.