Technical Report CS73009-R

LPL: A GENERALIZED LIST PROCESSING LANGUAGE

Billy G. Claybrook

September 1973

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061, (703) 951-5420

ABSTRACT

The paper describes LPL, a generalized list processing language. LPL allows the user to define multiple cell structures and cell sizes at runtime, thereby allowing nonhomogeneous list structures. The paper examines the problems associated with list tracing in systems allowing multiple cell-types. Complex list tracing during garbage collection in LPL is avoided: (1) by creating a doubly-linked super list of all allocated cells and (2) by using a reference count scheme. No marking phase is required for garbage collection.

The problem of developing insertion and deletion procedures for lists with cells having multiple types of pointer structures is discussed, and LPL solutions are given. LPL statements can handle singly-linked, doubly-linked, left-right-linked, and some multi-linked pointer structures automatically.

The design philosophy and the data organization for LPL are discussed in detail. Examples of the definition of cell structures are given, and all of the LPL list manipulation and creation statements are examined and discussed.

KWIC index terms: programming language, list processing, and data manipulation.

1.0 INTRODUCTION

Many programming applications, e.g. symbolic and algebraic manipulation applications and artificial intelligence applications, require the use of list structures. Several list processing languages or languages that provide for list processing have been developed, e.g. LISP 1.5 [5], LISP 2 [1], SLIP [7], PL/1 [8], ALGOL 68 [6], etc. With the exception of ALGOL 68 and LISP 2, none of these languages provide the user with the capability to define the cell size and/or the configuration of the list cell at runtime (PL/1 attempts to do this with BASED structures but falls short because only one component in a BASED structure can have its dimension altered at runtime). ALGOL 68 has the disadvantage that it is not yet readily available for most users, and it suffers from list tracing complexities during garbage collection. LISP 2 was an attempt to allow variable cell configurations, but the project was aborted.

The language (LPL) described in this paper is an attempt: (1) to provide a generalized list processing language available to a wide range of users, (2) to provide the user with the capability to define cells with varying characteristics that are natural for his applications, (3) to allow garbage collection to proceed easily without the complexities associated with list tracing inherent in list structures with cells having a variable number of links, and (4) to provide the user with a set of statements, e.g. COPY, REMOVE, etc., that perform operations for which the user normally has to generate the code. Anyone who has tried to use LISP 1.5 or SLIP to program solutions to problems whose data are not homogeneous in size and format knows the importance of the existence of languages with the capabilities of LPL and ALGOL 68. Fenichel [2]

describes the importance of allowing multiple cell types in list structures. LPL allows the user to concentrate on ideas and avoid the clumsiness of trying to determine a way to represent a structure in a less flexible language. LPL is designed for ease of use and ease of implementation. It is an attempt to exploit the advent of virtual memory systems.

Cells in LPL can be members of many lists; thus, when a cell is inserted (deleted) into (from) a list, a knowledge of which pointers to modify is required. Insertion, deletion, and list tracing are the most difficult problems to handle in languages allowing multiple cell structures. We discuss the solutions to these problems in Section 3.0.

Section 2.0 gives an overview of LPL. Section 3.0 gives a complete description of the LPL data organization, data types, and statement forms.

2.0 OVERVIEW OF LPL

An LPL program consists of a sequence of statements separated by semi-colons. The initial implementation of LPL is in PL/1 as an extension to PL/1. Programs written in LPL are scanned during a preprocessor pass, translated into a PL/1 program, and compiled by the PL/1 compiler. LPL requires the use of dynamic storage allocation so the choice of PL/1 was natural for an initial implementation (this implementation in PL/1 was possible only after some imaginative uses of BASED structures).

The existing LPL implementation allows any PL/1 statement to be included in an LPL program. Current design of a second implementation does not include any existing language as a base language, i.e. LPL will exist as an individual processor with the capability to perform arithmetic operations, conditional transfers, etc., in addition to the list processing capabilities described in this paper.

Each LPL cell has certain fields in common that reduce the problems inherent in copying recursive lists, garbage collection, etc. LPL uses a "super list" to hold all nodes allocated by the user and a reference count system for garbage collection; hence, simplifying list tracing considerably and eliminating the marking phase of garbage collection. The use of the super list for garbage collection is explained fully in Section 3.1.

LPL allows the user to invoke the garbage collector or to release cells individually. One might wonder why we allow the user to do this since LPL basically uses a reference count approach to freeing inactive cells. Normally, in systems using the reference count method, the storage for a cell is released as soon as its reference count becomes zero. When a cell in LPL is removed from all lists by the REMOVE statement, its reference count becomes zero. However, it is possible that the user may want to insert this cell into another list that it was not a member of when the REMOVE statement was executed. In order to allow this flexibility and to allow the user to develop his own garbage collection scheme, if he desires, we simply consider the cell to be inactive until its storage is released by execution of the RELEASE or COLLECT statements.

3.0 DESCRIPTION OF LPL

LPL allows any number of cell types to be defined by the user. Each cell type has a template associated with it to describe its characteristics. LPL allows considerable flexibility in creating list structures since it: (1) allows the designer to define any cell type he desires, (2) allows an arbitrary number of pointers per cell, (3) allows multiple types of values per cell, and (4) provides the user the capability to implement cells with identifier (as in SLIP [7]) and reference count fields (the user identifier and reference count fields are not to confused with the SYSID and SYSRC fields described below).

3.1 Data Organization in LPL

Each cell allocated in LPL has the following fields in common:

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

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

(3) a system reference count field (SYSRC),

(4) a system identifier field (SYSID), and

(5) two link fields, MLP and MRP, that link all cells allocated by the user into a doubly-linked "super list" for use by the garbage collector.

The user cannot access directly any of the five fields listed above.

A description of each of the common fields in a cell is necessary to establish the basic concepts of LPL data organization. The TYPE field points to the template. The template gives the number of components and their type in a cell. Fig. 1 gives an example of the format of templates (the format is similar to that of Fenichel [2]). Besides the common entries ((1) - (5) above) in each cell, the user can select up to seven other types of entries (given in Section 3.2). Each template consists of a variable number of words (at least two) with the contents of each word, except the first, giving the type and the quantity of that type of entry. Word 1 always contains the number of components in a cell (the number of arguments in the DEFINE statement minus one for the type is the contents of word 1); therefore, the contents of word 1 plus one gives the number of words allocated for the template. Codes in the first half of each word except the first indicate the type of entry (the codes are given in Fig. 1). Entries in a template can appear in any order (they actually appear in the order they appear in the DEFINE statement). Templates are not duplicated in LPL.

		7
Word 1		
Word 2	RL	# REAL Values
Word 3	IT	# INTEGER Values
Word 4	AL	# ALPHA Values
Word 5	ID	# IDENT Fields
Word 6	RF	# RFCNT Fields
Word 7	LK	# POINTER Fields
Word 8	KY	# KEY Fields

Fig. 1. Template Structure

The COPY bit is used to mark records during the copy process. One problem associated with list processors is garbage collection. We have tried to minimize this problem in LPL by using the system reference count (SYSRC) and the super list. The number of lists in which a cell is a member is maintained by the LPL system in the SYSRC field of the cell. As long as the SYSRC is greater than zero, the corresponding cell is active and is a member of one or more lists (not counting the super list). During garbage collection the super list is traversed and the storage occupied by inactive cells is released. The super list is maintained: (1) to alleviate the system from maintaining a record of pointers to all accessible lists created by the user, (2) to avoid the process of tracing all accessible lists during garbage collection and avoiding the marking phase, and (3) to simplify the implementation of certain functions, such as those performed by the REMOVE and REPLACE statements. List tracing [2] in list structures having cells with an arbitrary number of link fields per cell is a problem to be avoided if possible.

LPL allows the user: (1) to release the storage occupied by cells using the RELEASE statement (provided the SYSRC of the cell is zero), or (2) to invoke the garbage collector by executing the COLLECT statement. The garbage collector is also automatically invoked by the LPL system when about 95 percent of the working space available for allocating cells is depleted. Note that garbage collection in LPL does not require any additional working space and is a very simple process when compared to other garbage collection schemes [3].

Finally, the SYSID field is used to specify the pointer structure in a cell. This field requires only three bits to specify whether the pointer structure in the cell is singly-linked (SYSID=0), doubly-linked (SYSID=1), left-right-linked (SYSID=2), multi-linked (SYSID=3), or key-linked (SYSID=4). Knowledge of the pointer structure by the INSERT and DELETE statements is necessary for each cell because LPL allows any mixture of the five pointer structures to exist in a single list. Thus, the INSERT and DELETE statements need only to access the SYSID field in each cell, instead of interrogating a template, to determine the pointer structure of the cell. One of the main distinctions between other list processors and LPL is that its INSERT and DELETE statements are provided with the capability to handle any of the four distinct pointer structures automatically.

One assumption is made with respect to the pointer structure in LPL cells. For cells with doubly-linked or left-right-linked structures, LPL assumes the first pointer is the backward or left pointer, and the second pointer is the forward or right pointer. If the user feels restricted with these simple assumptions inherent in the INSERT and DELETE statements, then he has the capability to develop his own procedures to replace them.

It is more difficult to provide generalized insert and delete operations for lists with cells having multi-linked pointer structures. The problem presented by multi-linked cells occurs because there is no automatic way to determine which pointers to modify during insertion and deletion unless specific facilities are provided by the list processor language. The easiest solution to this problem is to let the user develop his own insert and delete procedures and define any constraints that he desires. Another solution used by implementers of multilist file organizations [4], is to associate a key with each pointer. Therefore, when tracing a list with multi-linked cells, the user can specify, by presenting a key, which pointers to access. LPL provides the

user with a choice of either solution. With cells having the key-linked structure, the INSERT and DELETE statements assume the first key is associated with the first pointer, etc. The KEY attribute (discussed in Section 3.2) is used for identifying keys in a cell.

3.2 Data Types and Attributes

The language includes the data types REAL, INTEGER, ALPHA (equivalent to PL/1 CHARACTER), POINTER, LIST, STACK, and QUEUE. LPL has attributes NODETYPE, IDENT, RFCNT, and KEY. The declaration

```
DCL   ITYP   INTEGER   NODETYPE,
      ID     INTEGER   IDENT,
      RC     INTEGER   RFCNT,
      KY     ALPHA(5)  KEY;
```

declares ITYP to be an INTEGER variable indicating the type of cell, ID to be an INTEGER variable having the identification attribute (i.e. ID represents an ID field like that in a SLIP [7] cell), RC to be an INTEGER variable having the reference count attribute, and KY to be a five character key. The meaning of each of these attributes will be described further in Section 3.3 in the discussion of the DEFINE statement.

3.3 Definition of Cell Structure

The DEFINE statement provides the facility for the user to describe the structure of the cells that he desires for an application. Each time the DEFINE statement is executed it sets up a template (unless one already exists) describing that cell type. Suppose we have the declaration statement:

```
DCL   ITYP   INTEGER NODETYPE,
      RV     REAL,
      IV     INTEGER,
      CV     ALPHA(20),
      LK     POINTER,
      ID     INTEGER IDENT,
      RC     INTEGER RFCNT,
      KY     ALPHA(5) KEY;
```

The mode of the variables in this declaration statement indicate to the DEFINE statement the types of fields to be included in a cell. Multiple entries of each type can be present in the DEFINE statement (this will be reflected in the template); thus, allowing freedom in naming fields in a cell structure. The format of the DEFINE statement is illustrated in the example:

DEFINE CELL(ITYP, RV(H), IV(I), CV(J), ID(K), RC(L), KY(M), LK(N));

The variables in the argument list must appear in a declaration statement preceding the DEFINE statement. The H, I, J, K, L, M, and N INTEGER quantities (they could also be INTEGER constants) indicate how many entries of each type are in a cell. Each cell can have four types of values: REAL, INTEGER, ALPHA, and POINTER. The IDENT and RFCNT entries in a cell allow the user to have his own identifier and reference count fields. These two types of entries increase the flexibility of LPL by allowing the user to develop any list structure format that he desires. We expect that only one identifier and one reference count field will be included in a cell, but the facility exists for more than one of each. Not all variables must appear in the DEFINE statement. The statement

DEFINE CELL(ITYP, CV(3), LK(2), IV(2))

describes a cell of type ITYP with three alphanumeric values (with 20 characters each), two pointer fields, and two integer values. Fig. 2 shows the template corresponding to the above cell structure. When a variable with the KEY attribute is included in the DEFINE statement, this implies the cell has a key-linked pointer structure. The number of keys and pointers must be the same. A KEY variable can be any of REAL, INTEGER, or ALPHA data types. The arguments in the DEFINE statement can appear in any order.

Word 1		3
Word 2	AL	3
Word 3	LK	2
Word 4	IT	2

Fig. 2. Template Structure for Cell with Three Types of Fields

3.4 LPL Data Manipulation Statements

We provide LPL with enough high level manipulation statements to save the user time in writing programs, but we also give him the full capability of a list processor to program at a lower level. The tokens in each LPL statement are separated by commas with a blank following each keyword. LPL has the following data manipulation statements (a cell pointed to by POINTER variable P is designated as cell P; and [ ] indicates the contents are optional):

1.	GET P,I; -	Allocates a cell P of type I.
2.	DELETE P,T[(KEY)] [,R]; -	Deletes cell P from list T, cell P follows cell R (optional); if key KEY (optional) is specified then only pointers associated with KEY are modified.
3.	INSERT P,T[(KEY)],R; -	Same as DELETE except cell P is inserted in list T after cell R (KEY is again optional).
4.	RELEASE P; -	Frees the storage occupied by cell P. The SYSRC for cell P must be zero.
5.	ENTER exp,T; -	Enters the value of exp into T (T must be a STACK or QUEUE and the mode of the value of exp must agree with that indicated in the CREATE statement for T).
6.	TAKE X,T; -	Removes an entry from T (T must be a STACK or QUEUE) and place it in X.
7.	RV(N),P = exp; -	Sets the N^{th} REAL, INTEGER, ALPHA, POINTER, IDENT, RFCNT, or KEY field in cell P to the value of exp.
8.	X = RV(N),P; -	X becomes the N^{th} value of one of the fields in cell P.
9.	COPY S,T; -	Copies list S into list T.
10.	P = N,T[(KEY)]; -	P is the pointer (optionally associated with key KEY) to the N^{th} cell in list T.
11.	SPLIT T[(KEY)],P; -	Split list T at cell P. The use of KEY is optional like in the DELETE statement.
12.	CONCAT S[(KEY)], T; -	Concatenates lists S and T in the order given.

13. COLLECT; - Invokes the garbage collector.

14. ERASE T; - Erases list T (cells are deleted from T).

15. P, {SINGLY | DOUBLY | L-R | MULTI | KEY}; - Declares cell P to have one of the five pointer structures indicated.

16. CREATE T {STACK | QUEUE | LIST} [{SINGLY | DOUBLY | L-R | MULTI | KEY}] [{REAL | INTEGER | ALPHA | POINTER}]; - Declares T to be a pointer to a specific structure. The data types REAL, INTEGER, ALPHA, and POINTER apply only to STACKS and QUEUES.

17. REMOVE P; - Removes cell P from all lists of which it is a member.

18. REPLACE P,R[,T[(KEY)]]; - Removes cell P from all lists or from a single list T and replaces it with cell R. Only the pointers associated with KEY may be changed (optional).

A few of the LPL statements require a more detailed description; however, some such as COPY, CONCAT, etc. are self-explanatory and need no further discussion (algorithms for COPY, CONCAT, etc. are either well known or trivial and are not discussed). The GET statement allocates a cell of a specific type. During the allocation of this cell, the template for this cell type is interrogated to determine the cell's structure. Statement #15 above is required to fix the pointer structure of cells in a list not created by the CREATE statement. Failure to do this results in a cell having a singly-linked pointer structure automatically assigned to it. The INSERT statement increases the SYSRC by one for each execution, and the DELETE statement automatically decreases the SYSRC by one (other information about these two statements has been discussed in detail in Section 3.1). Statement #7 is important because it allows the user to place values in the respective fields of cell P.

The ENTER and TAKE statements operate on STACK and QUEUE structures defined by the CREATE statement. Before discussing these two statements it is necessary to describe the CREATE statement. The CREATE statement allows the user to set up LIST, STACK, or QUEUE structures with a homogeneous pointer structure. The variable T in the CREATE statement is of type POINTER and points to a descriptor. The CREATE statement clearly describes the format of the defined structure. The user does not have to set the pointer structure using statement #15 in each cell for these structures.

At compile time the code is generated to allocate and initialize each descriptor. Descriptor pointer fields are set to NULL. The format of these descriptors is given in Fig. 3.

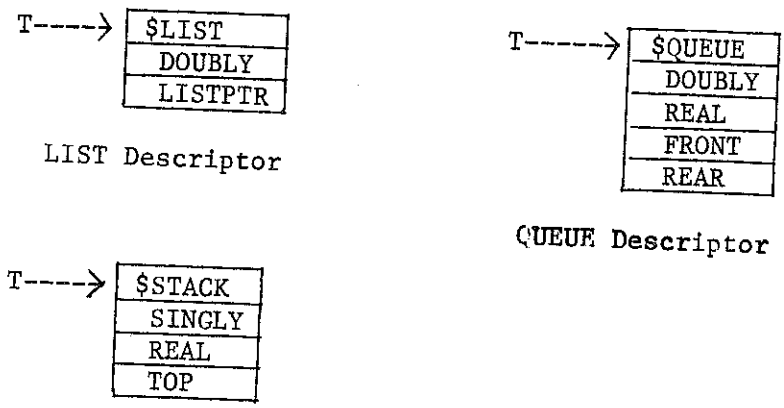

Fig. 3. LIST, STACK, and QUEUE Descriptors

The first word in each descriptor indicates the type of structure being described (actually the first word is not an alphanumeric string as indicated but is an integer identifier). The second word describes the pointer structure. The third word in both the STACK and QUEUE descriptors indicates the mode, e.g. REAL, of the values in the structure. The last word (or words) of each descriptor point to the structure.

At runtime the pointer entries in Fig. 3 are updated and modified as the structures are manipulated. As items are entered (removed) into (from) STACKs and QUEUEs, the pointers TOP, FRONT, and REAR are updated. Associated with STACK structures is a function TOP(T) that returns as its value the pointer to the top of stack T. FRONT(T) and REAR(T) are functions whose values are pointers to the front and rear, respectively, of QUEUE T. HEAD(T) is a function associated with LISTs. Its value is the pointer (LISTPTR) to the first cell in the list. These functions can only be used with structures declared in the CREATE statement unless the user sets up his own descriptors with the same format as LPL.

The ENTER and TAKE statements enter (remove) information into (from) the stacks and queues. The pointers to the TOP, FRONT, and REAR of the defined structures are automatically updated by the ENTER and TAKE statements.

The only other statements that we discuss are the REMOVE and REPLACE statements. Both are basically the same with the exception that REPLACE inserts another cell in place of the one removed. These statements perform several operations that would otherwise require a considerable amount of coding on the part of the user. At a given time there is no way to know in which lists cell P is a member (unless some very expensive, with respect to storage, bookkeeping tasks are carried out). But we do know the number of lists in which cell P is a member (by using the SYSRC value). There are two ways to attack this problem. The first way is to begin tracing the user created lists and determine in which lists cell P is a member. But as we have stated before, list tracing in lists with cells having a variable number of pointers is difficult and time consuming. Also storage is required to save pointers during the tracing. The second way (done by the REMOVE and REPLACE statements) is to trace the super list and determine which cells contain pointers to P. The examination of cells in the super list ceases when LPL locates the number of cells (containing pointers to

cell P) given by the value of the SYSRC field in cell P . In general, we expect less time to be consumed in tracing the super list (because of its simple pointer structure) than other user created lists; and no extra storage is required to save pointers.

4.0 SUMMARY

LPL is an attempt at providing programming convenience for the user. LPL is different from most other list processing languages because it allows the user to define multiple cell structures and cell sizes at runtime, thereby allowing list structures with multiple cell-types. LPL allows the user much flexibility in defining list structures. This capability proves useful when the user's data are not homogeneous. It also allows the user to represent structures in a more natural manner. While LPL is a flexible language that can be tailored to the user's requirements, it provides the complete and convenient programming facilities of a ready-made system.

List tracing is the primary obstacle, especially during garbage collection, to implementation of list processing systems allowing multiple cell types. LPL uses a super list containing all cells previously allocated and a reference count scheme to make garbage collection trivial. No marking phase is required; hence, complex list tracing is avoided. One of the primary design features of LPL is to avoid list tracing as much as possible.

LPL statements provide for handling the normal list processing operations, e.g. copying recursive lists, erasing lists, splitting lists, concatenating lists, garbage collection, etc. Other operations such as removing a cell from some or all of the lists of which it is a member (and possibly replacing it with another cell) are performed automatically by the execution of a single statement. All LPL statements can directly handle singly-linked, doubly-linked, left-right-linked,

and key-linked pointer structures. Special facilities are provided for the creation of stacks and queues.

Finally, LPL has been used by the author in artificial intelligence problems and in polynomial manipulations. These two problem areas require the use of several cell types.

REFERENCES

1. Abrahams, Paul W. "The LISP 2 Programming Language and System." AFIPS, 1966, FJCC, Vol. 29, Spartan Books, N.Y., pp. 661-676.

2. Fenichel, Robert R. "List Tracing in Systems Allowing Multiple Cell-Types." CACM, Vol. 14, August, 1971, pp. 522-526.

3. Knuth, D. E. The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Mass., 1968.

4. Lefkovitz, David. File Structures for On-Line Systems, Spartan Books, New York, 1969.

5. McCarthy, John, et al. LISP 1.5 Programmer's Manual. MIT Press, Cambridge, Mass., 1965.

6. Peck, J. E. L. (ed.). ALGOL 68 Implementation. North-Holland Publishing Company, Amsterdam, 1971.

7. Weizenbaum, J. "Symmetric List Processor." CACM, Vol. 6, September, 1963, pp. 524-544.

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