AN ABSTRACT OF THE THESIS OF

Charles Chilton Weems Jr. for the degree of Master of Arts

in Computer Science presented on July 11, 1979

Title: An Experimental String Processing Extension to Pascal

Abstract approved: Redacted for Privacy

William S. Bregar

The need for and problems related to string processing are discussed and a definition of the term "string" is derived. A brief review of some existing string processing systems leads to the presentation of a design for a string processing extension to the programming language Pascal. A rationale for the design, which is based upon a linear list structure, is included with the design presentation as comments and narrative. Several examples of the use of the extension are provided, including demonstrations of how some of the more complex facilities can be developed using the primitive elements of the extension. Narrative comparisons of the proposed extension with two popular existing string processors are provided.

An Experimental String Processing Extension to Pascal

by

Charles Chilton Weems Jr.

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Arts

Completed July 11, 1979

Commencement June 1980

APPROVED:

Redacted for Privacy

Assistant Professor of Computer Science
in charge of major

Redacted for Privacy

Chairperson, Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented July 11, 1979

Typed by Kathi Miller for Charles Chilton Weems Jr.

TABLE OF CONTENTS

LIST OF FIGURES

An Experimental String Processing Extension to Pascal

I. INTRODUCTION

A Definition of String Processing

String processing will be defined here as the manipulation of linear lists of data elements. These strings will be referred to as "L-lists" and the data elements as simply "elements".

String processing has a very broad range of applications, however, the most common is in the manipulation of alphanumeric text. In such areas as translation of computer languages, understanding natural language, text editing, text formatting, etc. String processing need not be restricted to these areas since L-list elements can be of any data type. The only inherent feature of a string is its linear structure, thus it is possible to have strings of integers, strings of arrays, strings of records and even strings of files

The Problem

This definition of a string as simply a linear list, consisting of elements of any data type, is broader than the typical historical definition but no real reason exists as to why strings should be limited to some specific data type. It is perfectly reasonable to create a set of string operations which are independent of the element data type.

A survey of current string systems reveals, however, that no system of this type exists. Most of the existing systems fall into two

major classes: string processors and list processors. SNOBOL is typical of the string processors in that its strings are sequences of alphanumeric characters. SNOBOL permits operations such as concatenation, matching and conversion of numeric values to and from corresponding string representations. LISP, on the other hand, is typical of the list processors. It allows linear lists of atoms to be created. An atom is a more general data type which may be a number, a name or a pointer to another list. LISP lists more closely resemble our definition of an L-list, but are still somewhat restricted. The LISP language allows operations that combine lists, extract either the first element or all remaining elements and comparison of elements, to name a few.

These two languages are probably the most popular of the current string processing systems. They, along with their cousins, suffer from the same drawback of not being able to work with string elements of any data type. In addition, most of these systems lack processing capabilities found as standard features of the general purpose programming languages. This lack stems from the origins of these systems, which were created to fill a need that was not addressed by the general purpose languages. These special string processing systems were designed as tools to be used in exploring the applications of string processing, but not with the goal of becoming general purpose languages. Now that they have shown how useful string processing can be, it is time that their abilities be incorporated into the general purpose programming languages so that they may be used in everyday programming applications.

The string processing capabilities of most general purpose languages, such as FORTRAN, BASIC, COBOL or PL/1, are very limited. The

typical string construct is a variable length character array on which a few operations such as concatenation or extraction of a substring may take place. Some of the languages also have very specialized string handling abilities, such as the way in which COBOL formats its output. Obviously, these do not provide the same level of string processing ability as the special string processors, although they do provide some of the needed functions.

This then is the problem: Advanced string processing capabilities are needed and can be applied in many ways, both in special string oriented problems and simply to augment normal computer applications. Currently there does not exist a system which is general enough to handle string elements of any data type or that has, in addition to string capabilities, the features found in general purpose languages which make them so useful. No stand alone string system or language extension to date has filled this void. This is the issue which is addressed in this thesis.

The Objectives

This thesis develops a set of terms, constructs and operations for dealing with L-lists of data elements in the context of a general purpose programming language. Elements may be of any type allowed in the language. Further, it will be shown how the general model may be incorporated into an existing general purpose language in order to create an effective general L-list system, which also may be used effectively as a string processing system in the traditional sense.

No actual implementation of the design is done. Instead, a subset of the system primitives will be described by algorithms in narrative form and it will further be shown how some of the remaining non-primitive operations can be coded once the primitives are implemented.

The programming language Pascal was chosen as the base to which the general model will be adapted. There were several reasons for this, including the richness of control structures and data types found in the language and its ability to define new data types based upon the standard ones. The fact that the language is growing quickly in popularity but is still young enough to easily accept a modification, such as this, was another factor involved in the decision. Also important was the simplicity of the language design. The term "elegance" will be used in reference to this, meaning that the constructs are both simple and yet broad in scope and thus provide a great amount of power to the programmer. An attempt was made with the incorporation of the general string model into Pascal, to maintain the level of elegance currently found in the language.

II. DESIGN AND RATIONALE OF THE SYSTEM

Considerations in the Design of a String Processor

In designing an L-list processing extension for Pascal, there are several factors to be considered. First, the design must be general, that is, it must be applicable to all of the areas in which string processors are currently used and, if possible, to other areas as well.

Second, the design must be complete. It must be capable of performing all of the basic string operations. These are defined by Housden in (19) as:

1. Create a string of characters.
2. Concatenate two strings to form a new string.
3. Extract a segment of a string.
4. Search within a string for a given substring.
5. Compare two strings.
6. Delete a substring and replace it with another substring.
7. Insert a string within another string at a specified position.
8. Interrogate the length of a string.

Close examination of these reveals that they are not all primitive operations. Extraction, for example, is really just creation of a copy of the original string from which all of the elements have been deleted leaving only the substring which was to have been extracted. It can be seen that the following are primitive operations on strings:

1. Creation of a string.
2. Insertion of a string within another.
3. Deletion of a segment from a string.

4. Searching for a pattern within a string.

5. Comparison of two strings.

6. Length interrogation.

Concatenation, a common "primitive" in many systems, does not appear in the list because it is actually a special case of insertion (insertion at one end).

A third consideration in the design of a string processor is usability. The language should be easy to use from the programmer's standpoint. The terminology, symbology and structure should be clear, understandable, readable and yet concise. It should also be natural, in that a programmer should be able to develop a "feeling" for the "spirit" of the design and thus, even in the absence of references, be able to generalize that understanding and correctly predict the reactions of the system to new uses. To do this, the system must be based upon simple, fundamental concepts.

Fourth in the list of considerations is the efficiency of the system, both in terms of space and time. All of the benefits of a system can easily be overshadowed by a lack of efficiency.

Last, but perhaps the most important consideration of all, is that the design should be a natural extension to the language. It should fit cleanly into the language structure. One of the worst sins committed by people who design language extensions is that of simply tacking on a hodge-podge of bells and whistles which totally destroys any quality of unity in the original language design. This is especially true when dealing with Pascal since one of its finest features is its simple, unified design. It provides powerful operations which are

general in their scope of application. If a language has a certain amount of unity then any extensions should be designed to maintain or possibly enhance it, no matter how major or trivial they might be.

A Review of Existing Systems and Techniques

Most of the very early string processors (1950s and early 1960s) were rather simple. A common technique was to design what was effectively a string processing machine and develop an assembler language for it. Systems such as these, which include IPL-V (28), L-6 (21) and EOL (23), were fairly primitive, requiring the user to manually control free storage, work with a fixed number of string "registers" and various other low level operations. Some, such as LOLITA (4) and TRAC (27), were interactive and performed much like string-oriented programmable calculators. These were really the predecessors of today's interactive text editors.

The majority of the early systems were thus too low level to have much value as extensions to a high level language. They were interesting, however, for their choice of operations and data structures. Most of them were concerned with operations such as concatenation, deletion, copying and occasionally, scanning for a pattern match. Each one had its own unique group of operations, but despite their differences, they all agreed in one respect: strings were stored as linked lists of elements, each element being a character or symbol. Typically, singly linked lists were used. That is, each node contained an element and a link to the next node.

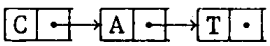


Figure 1. A typical string representation found in early string processing systems.

Some were rather sophisticated, such as IPL-V which included some extra status flags in each node, or L-6 which permitted multiple links, thus making it possible to build doubly linked lists, trees and other complex structures. The L-6 facility is comparable to what may be achieved through the combination of records and pointers in Pascal, although the operations provided by L-6 were more primitive.

In contrast to the linked list technique such systems as SASP-1 (2) and DASH (26) used arrays to store string elements. The advantage of the array storage method is that it makes efficient use of memory. The disadvantage is that insertions and deletions require a great deal of data movement. The linked-list method is nearly the reverse of this since it makes possible very fast insert and delete operations but requires two or three times as much memory space as does the array technique.

A compromise technique in which short arrays are linked into lists has been proposed by several authors, including Madnick (24) and Berztiss (3). This makes relatively efficient use of memory space and provides for reasonably fast insertion and deletion operations. The algorithms which manipulate the data are, however, considerably more complex than those used with the other two methods.

During the early 1960s three major non-numeric data processing systems were developed which were much more advanced and would become and remain quite popular. One of these was McCarthy's LISP systems

(25, 33) which was designed to manipulate lists or symbols, called atoms, which could range in complexity from characters or numbers to whole sentences. The design of LISP is quite simple and yet powerful. At one point in the design of this proposal, a LISP-like extension to Pascal was under consideration and another system, LITHP (17, 31), which is a LISP-like extension to ALGOL (a language similar to Pascal) was of great interest. Despite its elegance, LISP is not suited for integration with Pascal because of its radically different design philosophy. LISP is based upon the concept of recursively evaluated functions and although Pascal provides such facilities, the majority of the language constructs stress non-recursive techniques. Inclusion of a LISP subsystem would be a major break from the design philosophy of Pascal.

Despite the appearance that it is inapplicable to the current design, LISP is a popular language and there are several lessons which may be learned from it. LISP derives its power from the fact that it is based upon a mathematical foundation, in this case Church's lambda notation for functions. The popularity of LISP, especially in Artificial Intelligence circles stems from its generality. List elements in LISP are much more versatile than just single characters and so LISP finds its way into a very wide range of applications. One major complaint about LISP is that its heavily parenthesized notation makes programs difficult to read.

Another of these three systems is Weizenbaum's SLIP (34). Although it has seen a decline in popularity in the last few years, SLIP introduced a number of concepts which are worth examining. SLIP is based

upon the combination of a bi-directional linked list and a mechanism called a reader, which is used in traversing the list. Each element of the list can be of several different data types, including lists. This makes it possible to have sublists within lists. The reader mechanism, recognizing this, is more than just a simple index, but rather a stack of indices in which the top of the stack is the index in use at whichever level of sublist is being traversed. SLIP also points out some problems which shared lists and elements. The main problem is that if two lists have pointers to the same sublist, which one "owns" it? In other words, how can it be determined that a sublist is no longer in use and therefore should be returned to free storage? This is an even greater problem in LISP, where each element exists only once and is pointed to by all of the lists which "contain" it. Both SLIP and LISP solve this through some bookkeeping operations on the structures.

Although SLIP lacks the elegance we are looking for, it has much more of the Pascal flavor that is also desired. It gets this from the fact that it was indeed a language extension, the language being FORTRAN in this case. The structures and processes used in SLIP do complement Pascal much more than the structures found in LISP. The method by which it gains generality, allowing different data types as list elements, is worth noting.

SNOBOL (9, 11, 13) was the last of the three languages to come onto the scene, although it followed SLIP by only a few months. Unlike LISP or SLIP it is a string processor in the usual sense, dealing mainly with character data. Because it is not as general as LISP, it

has not become quite as popular, although it is widely used. Manipulation of single characters is not one of the strong points of LISP, although some lesser known extensions to LISP have allowed it to manage fairly well, these are a far cry from the usual implementation. Thus, SNOBOL does fill an essential niche.

The main feature of SNOBOL is its ability to examine strings of characters for matches with specified patterns. Unlike LISP or SLIP, it does this automatically with no additional user control required. Each string element in SNOBOL is a single character. Strings vary in length from being empty to filling all available memory space. Pattern matching is done via two indices, one of which is called the cursor, the other is called the needle. The cursor keeps track of the position in the string being searched of the current point where a match is being attempted. The needle is used to scan from this point, doing a character by character comparison of the string and the pattern. If a mismatch is found by the needle then the cursor advances and a new comparison is tried. The positioning of these pointers is not under user control, although a user command may be issued to fix them at the first character of each string. Normally, however, they are left free.

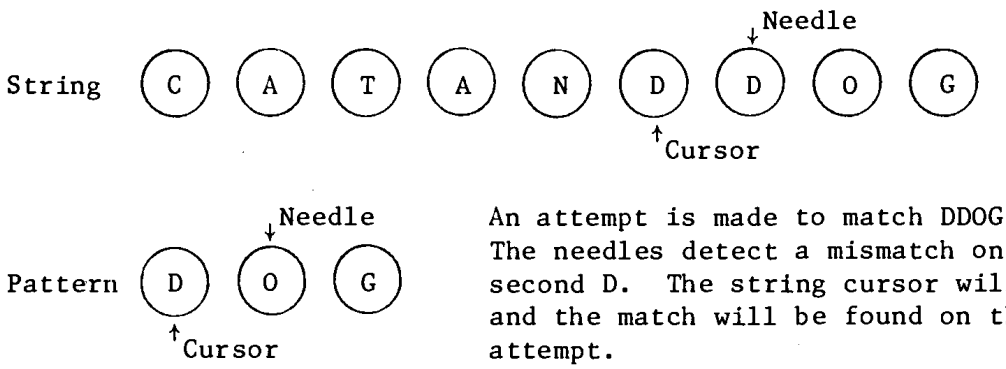


Figure 2. SNOBOL string representation showing cursors and needles.

SNOBOL also provides facilities for deletion and insertion of strings. There are several other interesting features in SNOBOL such as the ability to create patterns with alternate forms, any one of which may be matched, and procedures for transforming arithmetic values and expressions into string form and back again. In addition, it permits direct arithmetic operations on strings of numeric characters.

Although it is not very general, SNOBOL does point out the need for a simple pattern matching system and the ability to deal with characters on an individual basis.

It is important that Yngve's (35, 36) COMIT language be mentioned in our survey of existing systems. Although COMIT is no longer as popular, it contributed several important concepts to string processing, Pattern matching, for example, was originally introduced in its modern form in COMIT and in fact was the basis for the pattern matching constructs found in SNOBOL. COMIT also had a structure called a "subscript" which allowed the user to associate certain information with a pattern. This structure is now found in an analogous form in LISP as the much used property lists. Thus we can see that COMIT was an important contribution to string processing whose major features have been absorbed into the most popular of the modern string systems.

At this point the design considerations and information derived from previous systems may be summarized. The design should be general, complete, usable (and useful), efficient, simple but powerful and a natural extension to the language, in this case Pascal. Internal data structures offer various tradeoffs in terms of space and speed, and the desirability of each depends upon the application environment. Thus

the conceptual, user level design should be independent of the underlying data structure, permitting the implementors to use whichever structure is best suited.

Elegance may be achieved by using fundamental concepts such as algebras, calculus or topology as the mathematical foundation of the design. Generality is obtained by allowing a wide range of data types to be represented in the strings but the ability to manipulate individual characters must not be lost. The ability to create sublists can add a great deal of power to the system and readers can be a valuable tool in working with them. Finally, pattern matching is a useful operation which may have a place in a string system.

A Review of Pascal

The next step in the design process is to examine Pascal to see specifically what it already has to offer in the area of string and list processing. One of its primary data types is the character; however, variables of this type can contain only one character which limits their usefulness. Packed arrays of characters may be created and can be compared to identically dimensioned packed arrays of characters. There do not exist any provisions, however, for insertion, deletion, concatenation or any other combining operations between them. These must be done through programming character by character routines to form each operation. Another hinderance is that arrays are fixed in size and so it is often possible to run out of space in any particular array.

Pascal also allows for list structures through a combination of record and pointer data types. A record of from one to n characters with a pointer field leading to the next node and possibly one leading back to the previous node provides a mechanism for manipulating strings. Just as with arrays, however, the operations must be programmed at the lowest level which makes string processing overly inconvenient, especially for small applications such as output formatting or instructional programming projects.

A Comparison of Strings With Linear Lists

With these perspectives in mind it is now appropriate to return to the design process and begin to assemble a framework. First of all a conceptual model for strings should be found which encompasses all of the qualities which we desire in them. We must examine our string concept from an objective point of view and determine what structure is best used to represent it.

Knuth (22) defines a string as: "A finite sequence of zero or more symbols." This is more general than the traditional "sequence of alphanumeric characters" which is what most string systems are based upon. The use of the term "symbols" implies that a string may be made up of elements taken from any alphabet of symbols, not necessarily the alphanumerics. This is still too restrictive definition since it requires a specific alphabet. Our definition of a string would permit strings to be made up of elements of any type. It is possible to select element types which are not simple symbols or are symbols from a non-finite alphabet.

There is one structure which has a definition that matches our string definition exactly. This is also given by Knuth:

> A linear list is a set of $n \geq 0$ nodes X[1], X[2], ..., X[n] whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: the facts that, if $n > 0$, X[1] is the first node; when $1 < k < n$, the kth node X[k] is preceded by X[k-1] and followed by X[k+1]; and X[n] is the last node.

It can be seen that strings as defined by Knuth and in the traditional sense are actually a special case of the linear list. This is the model which we were looking for. It is simple yet general and it still permits us to do string manipulation in the usual sense.

We can see the completeness of this structure by comparing the following set of linear list operations, given by Knuth, with the set of essential string operations given by Housden.

1. Gain access to the kth node of the list to examine and/or change the contents of its fields.
2. Insert a new node just before the kth node.
3. Delete the kth node.
4. Combine two or more linear lists into a single list.
5. Split a linear list into two or more lists.
6. Make a copy of a linear list.
7. Determine the number of nodes in a list.
8. Sort the nodes of the list into ascending order based on certain fields of the nodes.
9. Search the list for the occurrence of a node with a particular value in some field.

Thus, we find that the linear list structure as it is formally defined satisfies all of the criteria which we established for a model

of a generalized string. In order to distinguish between strings as defined by Knuth and the type of string defined in this thesis all further usages of the word "string" will refer to the standard string while the term "L-list" will refer to our generalized string structure.

We should note that the first three operations as listed above are performed in relation to the kth element of the L-list. Knuth's definition indicates, however, that the positions of the elements are purely relative to each other, that is there is no actual numbering of the elements: the kth element occupies that position simply because it follows the (k-1)th element and so on. In order to find the kth element we will use an index or mark as an aid to traversing the L-list. This mark will keep track of the location at which we are currently looking in the L-list and thus provide a reference point from which we can step to the next node.

This notion of a mark being associated with an L-list will be formalized by extending our definition to say that an L-list will consist of an ordered pair <list, mark>, in which list is a linear list of data elements, as defined by Knuth, and the mark is an indicator of the current reference point element in the list.

If we carefully consider the insert operation as it is defined by Knuth we will discover another problem with the L-list structure: when n = 0 there is no kth element before (or after) which the insertion can take place. Since insertion is the primary method of constructing L-lists (recall that concatenation is a special case of insertion), we must provide a method for the "initial insertion" to take place. There are two ways to do this: we can redefine the insertion operation so

that it will handle the case where n = 0 or we can redefine the L-list structure so that n will always be greater than zero.

It turns out that the latter case will not only solve this problem but will also provide us with some beneficial side effects and thus we will choose to again modify the definition of the L-list. The result of this change will be that any L-list will always contain at least one node. This requires the definition of the X[1] or FIRST node as a special element which will always and only exist in the first position of an L-list and which will have a null data value.

Our definition of an L-list is now: An ordered pair <list, mark> where list is a set of $n \geq 1$ nodes X[1], X[2], ..., X[n] whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: the facts that X[1] is the first node and is a special null data element which may exist only in first position; when $1 < k < n$, the kth node X[k] is preceded by X[k-1] and followed by X[k+1]; and X[n] is the last node. Mark is an indicator of the current point of reference element in the L-list.

To complement this definition we will redefine insertion to be insertion of an element <u>after</u> the kth node. Also, because X[1] is a null data element it will not be counted in determining the number of nodes in an L-list, thus the length of an L-list will be n - 1.

The following illustration will help to clarify the structure of L-lists as we have defined them. For the sake of readability, the X[1] element will be referred to by the name FIRST in the remainder of this thesis. The symbol ϕ will be used to represent the null data value contained in the FIRST element.

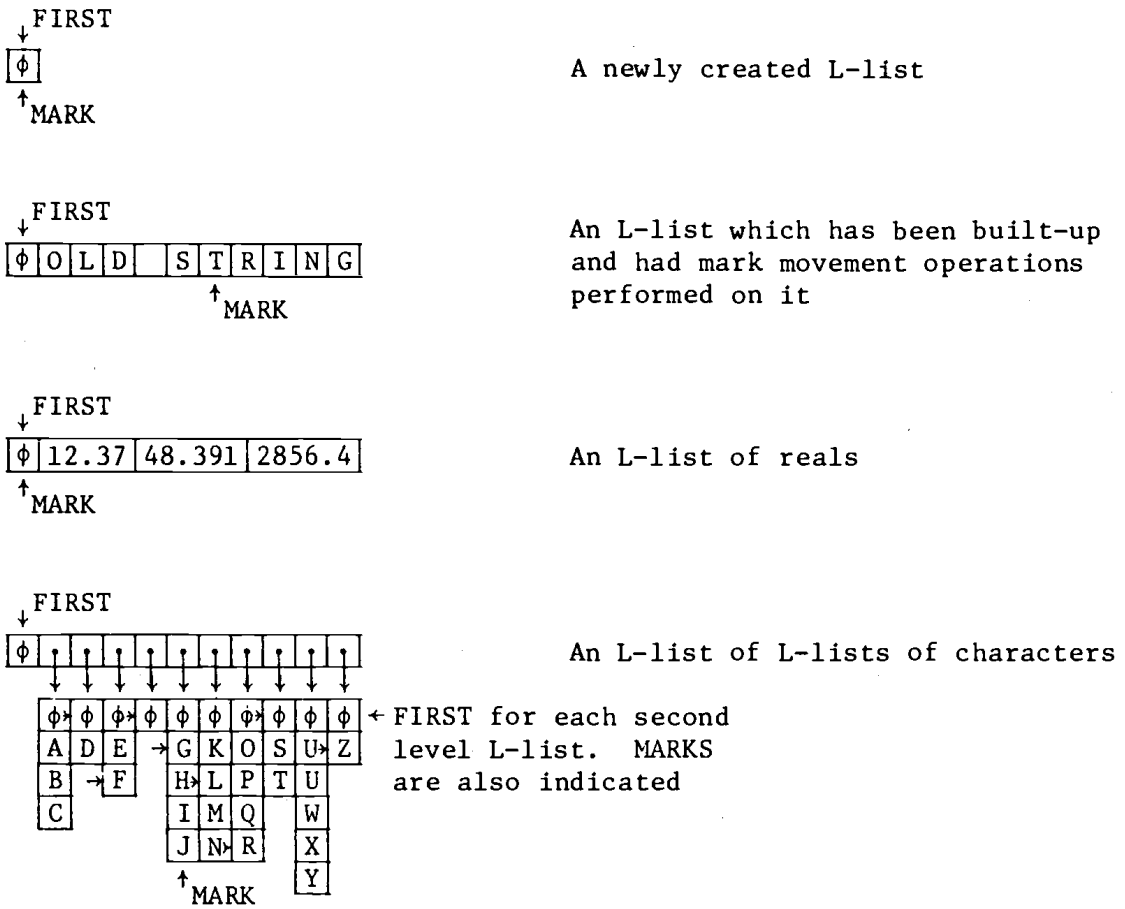


Figure 3. Some sample L-lists showing the marks.

The Proposed Pascal Extension

The proposed extension to Pascal consists of four parts: (1) a new standard data type, (2) a set of operations, (3) a new constant and (4) supporting subroutines (procedures and functions).

A New Standard Data Type

The new standard data type is referred to as the L-list type and is written in the form:

LLIST OF element-type

where the element-type is a Pascal standard or user defined data type.

From the user's point of view, this will appear to be a linear list or L-list as defined in the previous section. The underlying data structure which accomplishes this is of no concern here and is left up to the implementors.

In addition to the basic L-list of elements there are some additional constructs which will be used when we define the string operations. Each L-list is said to have two ends, these being the rightmost and leftmost elements. The element on the left end will be referred to variously as the first element, head or left-end of the string, or simply as FIRST. The rightmost element will then be referred to as the last element or right end of the string, or simply as LAST in the text of this thesis.

No L-list will ever be completely empty. Even when originally created there will be a single element in the L-list which signifies the head of the list. The head of every L-list will be the special null data element at all times. This is, however, a phantom element in most L-list operations. For example, in concatenation, although both L-lists have the null data element as their head, the resulting L-list has only one null data element, which will be its head. The rule for this is that every L-list will contain exactly one null data element which will be its FIRST element.

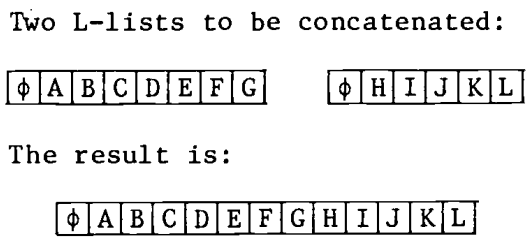


Figure 4. An example of L-list concatenation.

The FIRST element has no effect in pattern matching, insertion, deletion or equality comparison, nor is it included in the element count of the L-list, that is, the length of the L-list is the number of elements not including the FIRST element.

Each L-list also has associated with it a structure called the mark which specifies where certain operations take place on the L-list. The mark is similar to the SLIP reader, but only one mark is associated with each L-list in this system, thus the user must manage separate marks in any "sublists" that are used. The mark points at a particular element in the L-list which is referred to as the current position. All mark oriented operations take place at the current position or, as in the case of insertion, at the point immediately following the current element, between it and the element to the right of it. Note also that the mark may point to the FIRST element thus permitting such operations as insertion to the left of the leftmost non-null element. All mark movements are performed directly by the user through several mark movement operations provided in the system.

L-lists are created in a Pascal program either explicitly or implicitly in the VAR statement. L-list constants are not permitted in the CONST statement although type conversion, through the coercion routines provided, can virtually take the place of this, as will be seen later. When an L-list is created it consists of the FIRST element to which the mark points and has a length of zero. Through various operations the user may then build it up to any length within the bounds of available memory. The L-list itself is totally dynamic in terms of size, but its elements are fixed in type Even this may be partially overcome through

the use of records with variant fields, as the element type.

In the case of an L-list consisting of elements which are also L-lists the individual sublists are not explicitly declared in the VAR statement because the parent L-list is dynamic and it is thus impossible to know at compile time how many L-lists it will contain. These L-lists will be created through the BUILD function which will be discussed in detail in the section on procedures and functions.

A Set of Operations

The second part of the proposed extension to Pascal is a set of L-list operations for manipulating the new data type. These operations correspond to the standard operations in Pascal but are now simply extended (or overloaded) for L-list manipulation. Thus we will use the same symbols but they will now work with L-lists as well.

Assignment. The assignment operation for L-lists has the usual Pascal form:

L-list-variable := expression

The expression must evaluate to an L-list of the same type as the L-list variable. The current value of the L-list variable is replaced by the L-list resulting from the evaluation of the expression.

Addition. As mentioned earlier, concatenation is really just a special case of insertion. Thus, the addition operator will be redefined to include the more general operation of L-list insertion. This is a break from the traditional definition of addition as found in most string and list processing systems but we will see that the increase in usefulness obtained by choosing this definition will offset any inconvenience.

L-list addition (insertion) will thus take the form:

$$\text{L-list}_1 + \text{L-list}_2$$

The result of this is that all of the elements of L-list_2 (except its FIRST element) will be inserted into L-list_1 immediately following its current mark position. The resulting L-list will have its mark repositioned to the element which was the current position in L-list_2. Both L-lists must have exactly the same element data type.

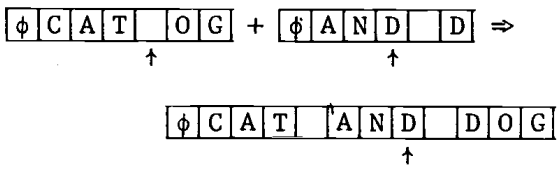


Figure 5. An example of L-list addition showing marks.

Note that L-list addition as defined above may include L-list variables or expressions which evaluate to L-lists. Also note that because the mark may be positioned at the FIRST element it is possible to insert elements at either end of the non-null portion of the L-list. This provides the ability to perform concatenation at either end of an L-list.

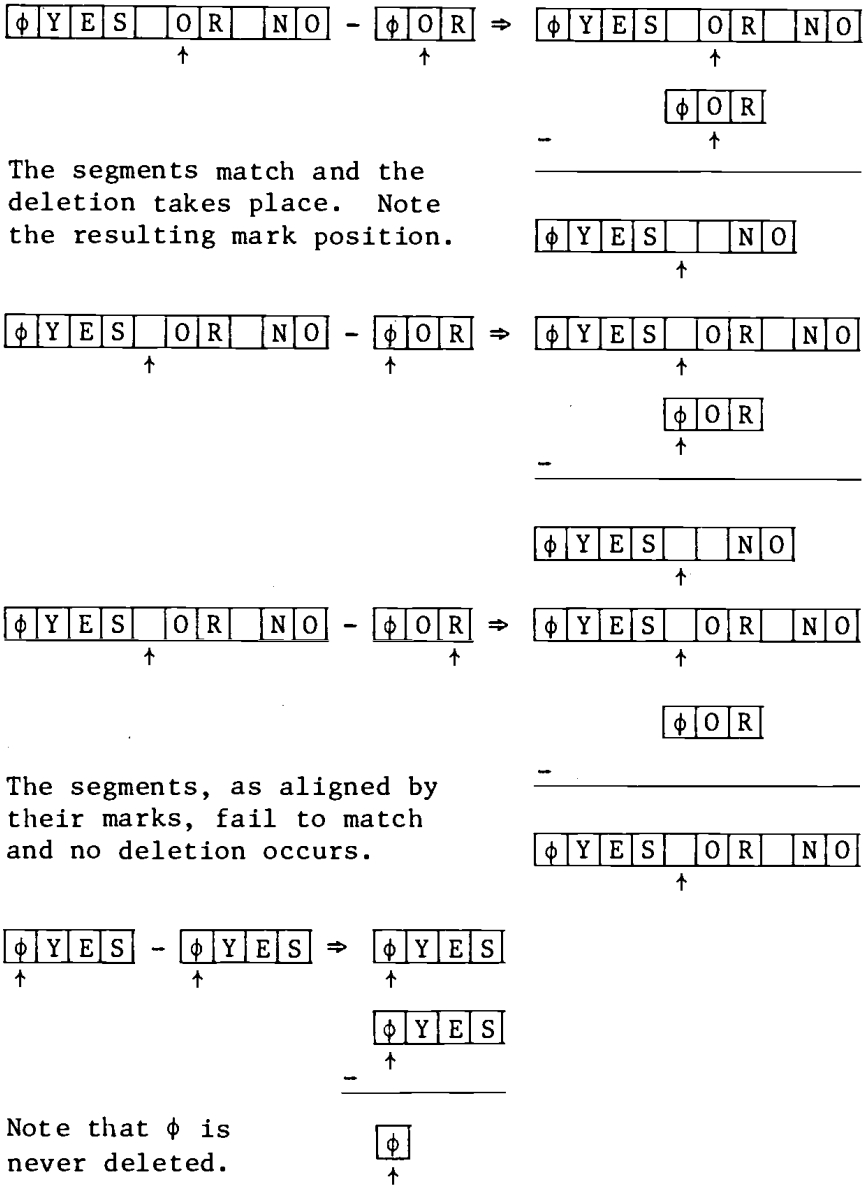
Subtraction. In the proposed Pascal extension the standard subtraction operator (-) will be augmented to work with the L-list data type as a deletion operation.

L-list deletion will thus take the form:

$$\text{L-list}_1 - \text{L-list}_2$$

where the result is that the two L-lists will be aligned by their marks and if and only if L-list_2 exactly matches a corresponding segment in L-list_1 (i.e. each pair of corresponding elements is identical) then

that segment will be deleted from L-list$_1$ (L-list$_2$ is unchanged). If the deletion occurs then the mark in L-list$_1$ will be repositioned at the element immediately to the left of the deleted segment. Note that such an element will always exist since the FIRST element must always be present in an L-list and thus can never be deleted. If the deletion does not take place the mark in L-list$_1$ is, of course, left unchanged.



In L-list subtraction, it is important to note that the FIRST elements never take part in either the matching phase or the deletion phase.

Figure 6. Some examples of L-list subtraction.

Just as in addition, both L-lists must have exactly the same element data type or a compile time error will occur.

The effect of aligning the L-lists by their marks is much like aligning the decimal points on two numbers which are being manually subtracted. This operation is valid for any L-list made up of any Pascal data type for which direct comparison for equality is possible. Real numbers, for example, may provide unpredictable results due to round-off error. This operation is roughly the inverse of addition; however, because neither is commutative the inverse property is only true when the operations are applied in an order exactly opposite their original application.

Multiplication. The multiplication operation in this system is written:

integer * L-list or L-list * integer

where the result of either notation is the same (i.e. it is commutative). The effect of this is a repeated self insertion of the L-list, integer - 1 times, at the current mark position.

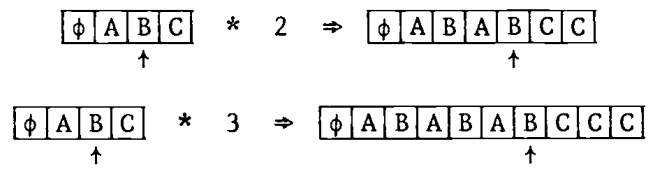


Figure 7. Two examples of multiplication of an L-list by an integer.

A particularly interesting case of this is when the mark is at either end of the L-list, since the result is repeated concatenation.

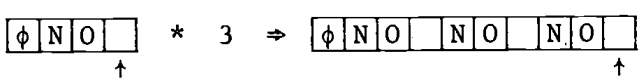


Figure 8. An example of L-list multiplication simulating repreated concatenation.

Division. L-list division comes only in one form:

L-list DIV L-list

The result of this is an integer which indicates how many times the L-list to the right of DIV can be deleted (subtracted) from the L-list to the left. This uses the same definition of L-list subtraction as given earlier and does not induce any other mark movements. L-list division is the inverse of L-list multiplication, in the same sense as subtraction is the inverse of addition: only if applied in reverse order.

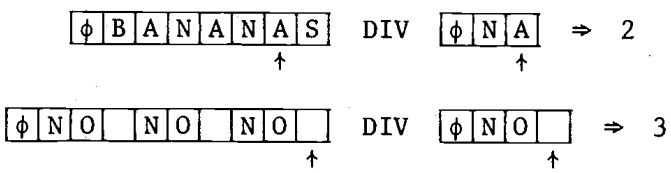


Figure 9. Two examples of L-list division.

The Pascal / operator is not defined for L-lists.

Modulo. In order to complement the DIV operator, the Pascal MOD operator has been defined to include L-lists. The result of:

A MOD B

where A and B are L-list variables, is the L-list A with B deleted from it A DIV B times. In other words, A MOD B is the remainder from A DIV B and by coincidence happens to be equal to:

A - ((A DIV B) * B)

which is the standard Pascal definition of MOD for the other data types on which it works.

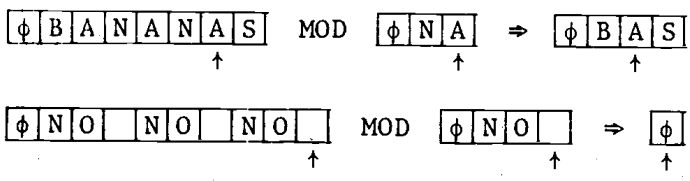


Figure 10. Two examples of the MOD operation.

Pattern Matching. Pattern matching in this system is performed by the Pascal IN operator which is extended to work with L-lists. The result of

A IN B

is a boolean indicating whether the L-list A can be found anywhere in the L-list B. Note that is is defined only for L-lists with element types for which the standard Pascal equality test applies (=). Of course, the L-lists must have identical element types for it to work at all.

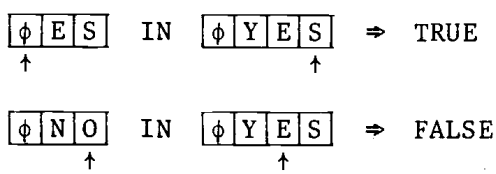


Figure 11. Two examples of the IN operation.

Note that in all of the comparison operations (including IN) the comparison is performed over the entire L-lists, independent of the positions of their marks.

The standard Pascal relational operators are extended to recognize L-lists in a manner similar to the way that they work on packed arrays. The standard L-list relations will thus be:

A = B, A >= B, A <= B, A > B, A < B, A <> B

where each one is an element by element comparison of the two L-lists

to determine whether the relation is true. These tests are only valid when both L-lists are of the same element type where that type is also one of the types for which the comparisons are predefined in Pascal.

When two L-lists are compared for exact equality they are first checked to see if they have the same length, if not then they cannot be equal. If the lengths match then comparison begins by comparing the elements that follow FIRST, in each of the two strings, using the standard Pascal comparison algorithm for the element type. If those two elements are equal then comparison proceeds to the next pair and so on, left to right, across the entire length of both L-lists. If all of the pairs of corresponding elements are equal then the L-lists are equal. If, however, an unequal pair is discovered then comparison immediately stops and the L-lists are unequal.

It should be mentioned that this definition of comparison permits direct testing of not only L-lists but nested L-lists, such as LLIST OF LLIST OF INTEGER. This is because the L-list comparison will automatically be recursively applied if the elements are themselves L-lists. In the case of nested L-lists, comparison then proceeds in a recursive depth-first manner and thus all of the corresponding pairs of lower level L-lists must also be equal, for the overall structures to be equal.

When one of the relational operators other than equality is applied between two L-lists the comparison proceeds in a manner similar to that for equality, except that their lengths are ignored and when two unequal elements are encountered the appropriate relational test is applied to them and the result of that comparison is taken as the result of the

overall L-list comparison. This is similar to the way in which we manually compare integer values: starting with the high-order (left-most) digits and proceeding rightward until we find a pair of digits that are unequal -- whichever integer contains the greater of the two unequal digits is then said to be greater than the other integer. If two L-lists are compared for an inequality relation and are of different lengths but are equal for all corresponding pairs of elements then the longer L-list is said to be greater than the shorter one.

Note that an empty L-list (containing only the FIRST element) will be less than any other L-list except another empty L-list (to which it will be equal).

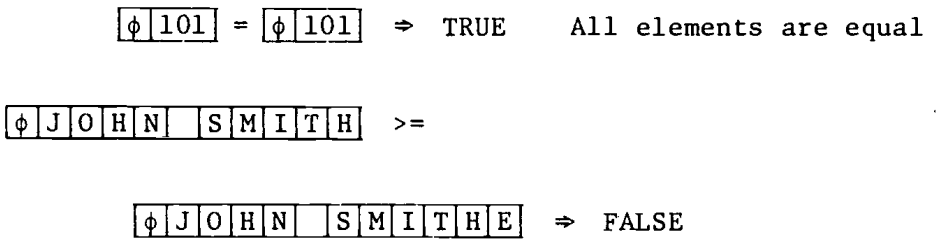


Figure 12. Two examples of string comparisons.

To tie all of the operators together it should be noted that for all of them, despite their extensions, the Pascal precedence rules are maintained. This includes the ability to alter precedence with parentheses. If this is done, however, the programmer must be sure that the result of a parenthesized expression is of the desired data type.

A New Constant

In addition to extending the definitions of most of the Pascal operators, this system also provides a new standard Pascal constant. This new constant is called EMPTY and is simply an L-list with no elements other than FIRST.

This constant provides a convenient method for performing such operations as resetting L-lists to their original empty state, checking to see if an L-list is empty, etc. EMPTY is also the additive identity element for L-lists.

Supporting Subroutines and Functions

The fourth and final part of this proposed extension is a set of functions and procedures which allow some additional manipulation of the L-lists or simply provide additional convenience. We will first examine the general functions then the general procedures, the extended matching functions and lastly the extended matching support procedures.

LENGTH (L-list) is a function which returns an integer, the number of elements in the L-list. The FIRST element is not counted.

SUCC (L-list) is a standard Pascal function which is extended to handle L-lists. Its result is the L-list with its mark moved one element right. If the mark is already at the last element, it aborts. Note that the result is a copy of the L-list.

PRED (L-list) is also a standard Pascal function. It is the reverse of SUCC and in regard to L-lists causes the mark to move one element to the left.

MATCHAT (L-list 1, L-list 2) produces a boolean result indicating whether L-list 2 matches a corresponding segment of L-list 1 when their marks are aligned.

BUILD (array, length, start) produces an L-list of the specified length made up from the array, starting at the specified array index. The array must be of the same element type as the L-list. The L-list mark is left at the element following FIRST. BUILD is one of the primary coercion routines for transforming data values into L-list form.

UNBUILD (L-list, length, start) produces an array from the L-list, starting at the current L-list position, and at the specified start element in the array, for the number of elements specified by length. The L-list and the array must be of the same element type. UNBUILD is another coercion routine but, unlike BUILD, it transforms L-lists into simple data values.

DISTTO (L-list 1, L-list 2) attempts to find a match of L-list 2, to the right of the current position in L-list 1. It returns an integer indicating how far the mark must be moved to reach the matching segment. If no match is found LENGTH (L-list) + 1 is returned.

DISTBACK (L-list 1, L-list 2) is the opposite of DISTTO. It searches to the left of the current position in L-list 1 for a match of L-list 2 and returns the distance that the mark must be moved back in order to reach it. If no match is found the result is LENGTH (L-list 1) + 1. L-list 2 may be EMPTY which will give the distance to FIRST.

ENCODE (numexpr) or ENCODE (numexpr:numexpr) or ENCODE (numexpr:numexpr: numexpr) transforms the result of the leftmost numeric expression into an L-list of characters. The conversion is done using the same algorithm as in the Pascal WRITE procedure.

LAST (L-list) chops off the tail of the L-list and returns it as a new L-list with the mark set to its last element. More formally, the function copies the argument, starting with the current element and continuing to the last element. This copy, with the mark left pointing to the last element, is then returned as the result of the function. This function is particularly useful in three cases: (1) DISTTO (L-list, LAST (L-list)) will return the distance to the last element of the L-list; (2) LAST (LAST(L-list)) will return the last element of L-list; (3) LAST (L-list) is quite useful whenever it is desired to recurse on the remainder of an L-list.

MARK (L-list) returns an integer value which is the distance from the FIRST element to the mark (the number of elements, including FIRST, preceding the mark). If the mark is at FIRST, the result is zero.

SHIFT (L-list, dist) moves the L-list mark the specified distance where dist is an integer. When dist is positive, movement is to the right and when negative it is leftward. If the distance given would cause the mark to be shifted part either end of the L-list, then it is left pointing to the element at that end of the L-list.

CURRENT (L-list) returns the value of the current element in the L-list. The result will be of the same data type as the L-list elements.

Current is primarily used as a coercion routine to convert data in L-list node form to pure values of the element type.

ENCLOSURE (L-list$_1$, L-list$_2$, L-list$_3$) is a bracketted masking function. L-list$_1$ is searched from its current position rightward for a match of L-list$_2$. If no match is found the function result is EMPTY. When a match is found the function begins copying elements of L-list$_1$ starting with the element following the segment that matched L-list$_2$. Copying continues until a match with L-list$_3$ is detected in L-list$_1$. Copying terminates with the element preceding the segment that matches L-list$_3$. If no match is found the entire remainder of L-list$_1$ is copied. If the match is found immediately after the L-list$_2$ segment then no elements are copied and the result will be EMPTY. The mark of the copy will be positioned at its last element. The L-list resulting from all of this is returned by the function. The basic purpose of ENCLOSURE is to allow the user to extract any sequence of elements enclosed by two other specific sequences of elements. By appropriate manipulation of the L-lists, however, the user may also extract segments bounded at only one end.

DECODE (L-list) uses the same algorithm as the READ statement to transform the characters into a real number which is returned as the result. The L-list must be an LLIST OF CHAR. The conversion starts at the current position in the L-list. The mark is left positioned on the element following the last character of the real number.

In addition to the functions, there is one new procedure:

MOVE (L-list, dist, flag) moves the mark of the L-list the specified distance where dist is an integer. When dist is positive, movement is to the right and when negative it is leftward. The flag is a boolean variable indicating whether it was possible to move the mark that distance without going past the end of the L-list. (TRUE indicates success.) If the move fails the mark is left at the same position. Note that unlike SHIFT, SUCC or PRED, MOVE does not create a copy of the L-list but actually moves its mark directly.

In addition to these functions and procedures, for general L-list precessing, this system provides a set of special routines for extended matching comparison. These are based upon a special structure called a Multi-Valued List which is actually just an LLIST OF LLIST OF elements but is given special treatment by this set of seven routines. The shorthand term MV-list will be used to refer to Multi-Valued Lists in this thesis. (MV-lists are not a predefined type but must be declared by the user as LLIST OF LLIST OF data-type.)

Each element of an MV-list is itself an L-list and will be referred to as an alternate value of the MV-list. When comparison is done between an L-list and an MV-list (as when using the ANYMATCH function described below), the L-list is compared to each of the alternate values of the MV-list and may match with any of them. This construct is most useful in building such structures as symbol tables, dictionaries or any other application where it is desired to automatically look up a value in a large table. The RETRIEVE function allows the user to

associate two MV-lists in which one is a set of key values and the other is a set of data values (a relationship which may be reversed). Using RETRIEVE, the user can then search the keys and retrieve data which corresponds to the desired one.

There are four extended matching functions: ANYMATCH, RETRIEVE, ANYIN, and LOOKUPIN.

ANYMATCH (MV-list, L-list). The L-list is compared for equality against each alternative value of the MV-list, starting with the alternate following the FIRST element of the MV-list and proceeding rightward until a match is found or the LAST alternate has failed to compare. If a match is found, ANYMATCH returns TRUE, otherwise FALSE. The MV-list alternates and the L-list must be of the same data type.

RETRIEVE (MV-list$_1$, MV-list$_2$, L-list). An ANYMATCH type search is performed on MV-list$_2$ for a match with L-list. When a match is found the position of the matching alternate in MV-list$_2$ is noted and RETRIEVE then looks up the alternate in the corresponding position in MV-list$_1$ and returns this (an L-list, by definition) as its value. It no match with the L-list is found RETRIEVE returns EMPTY. If there is no alternate in MV-list$_1$ which corresponds to the match position in MV-list$_2$ the RETRIEVE aborts with a fatal execution error. The purpose of this function is to provide a basic data-base/table look-up facility for the user.

ANYIN (MV-list, L-list) is another comparison routine, similar to ANYMATCH, which scan the L-list for a <u>segment</u> that will match any of the alternates in the MV-list. Starting with the current

position in the L-list, ANYIN looks to see if a segment of elements will match any of its alternates. If no match is found then ANYIN advances to the next element of the L-list and tries to find a matching segment that starts there. ANYIN continues scanning the L-list and tries to find a matching segment that starts there. ANYIN continues scanning the L-list until it finds a position at which a match with one on the alternates occurs or until it reaches the LAST element of the L-list. If it finds a match ANYIN returns TRUE, otherwise FALSE. ANYIN is especially suited for any sort of parsing operation where it is desired to look ahead to see if a key element is present in the list.

LOOKUPIN (MV-list$_1$, MV-list$_2$, LOlist, Dist) is another retrieval function similar to RETRIEVE which uses an ANYIN type of search to locate a segment in the L-list which matches one of the alternates in MV-list$_2$. It then looks up the MV-list$_1$ alternate that is in the same position as the MV-list$_2$ alternate that was matched. If no match is found, EMPTY is returned, otherwise the MV-list$_1$ alternate (an L-list) is returned. The Dist parameter must be an integer variable. LOOKUPIN will return in Dist the number of positions the L-list mark must be moved forward to reach the segment that matched the MV-list$_2$ alternate. If no match is found the value of Dist will be set to LENGTH (L-list) + 1. It should be noted that just as in RETRIEVE both MV-list parameters may be the same. This results in the functions simply returning the alternate that matched the L-list or L-list segment. LOOKUPIN is, like ANYIN, expecially suited to parsing operations.

There are three special procedures provided which allow the user to build and manipulate MV-lists.

INSERT (MV-list, L-list). A copy of the L-list is inserted into the MV-list as an alternate, following the current position in the MV-list. The MV-list mark is then repositioned to point to the newly inserted alternate. Note that this operation is not the same as addition since the + operator would require that both operands be MV-lists.

EXTRACT (MV-list, L-list). The L-list (which must be a variable) is set equal to the current alternate in the MV-list and the alternate is deleted from the MV-list. The MV-list mark is repositioned to the alternate which preceded the one that was just removed. If the current position of the MV-list mark was at FIRST, then the L-list is set to EMPTY and the MV-list is unaffected.

PERMUTE (MV-list, L-list). The MV-list is set equal to the set of alternates consisting of all of the possible permutations of the elements of the L-list. This is primarily useful when a user wishes to check an L-list to see only if it contains contiguously all of the elements of some set, independent of the order in which they might appear.

Many of the general L-list routines may also be applied to L-lists. It should also be noted that the standard Pascal procedures WRITE and WRITELN are extended to work with the LLIST OF CHAR data type, much as they do for PACKED ARRAY OF CHAR.

III. SOME EXAMPLES

Derivation of Non-primitives Using the Primitives

Since this system will not be implemented it was felt that it would be appropriate to show how some of the functions and procedures could be written using the basic operations. First we must divide the constructs into primitives and non-primitives. The primitives are those which cannot be built from any of the other constructs but must be programmed at the bottom level.

Primitives:

A := B	MOVE(A,D,F)
A + B	BUILD(A,I,L)
A - B	CURRENT(L)
A = B	EMPTY
A > B	

Non-primitives:

A IN B	* LENGTH(A)	ENCODE(A)
A DIV B	SHIFT (L,D)	DECODE(A)
A MOD B	* SUCC(A)	LAST(L)
A < B	* PRED(A)	MARK(L)
A >= B	MATCHAT(A,B)	* DISTBACK(A,B)
A <= B	UNBUILD(L,I,N)	ENCLOSURE(A,B,C)
A <> B	* DISTTO(A,B)	ANYMATCH(M,L)
A * B	RETRIEVE(M,N,L)	INSERT(L,M)
	EXTRACT(L,M)	PERMUTE(L,M)

Those non-primitives marked with a * will be shown in this section as Pascal procedures and functions.

The LENGTH Function

This function basically sets the mark to the head of the L-list and then moves it to the last element, keeping count of the number of moves required. The result is the number of elements in the L-list. Because the single parameter is a value parameter (pass by value), the position of the mark in the original L-list is unaffected.

```
FUNCTION LENGTH (A: LLIST OF element-type): INTEGER;
VAR COUNT: INTEGER;
   FLAG: BOOLEAN;
BEGIN
   FLAG := TRUE;
   WHILE FLAG DO
      MOVE(A,-1,FLAG);
   FLAG := TRUE;
   COUNT := -1;
   WHILE FLAG DO
      BEGIN
         MOVE(A,1,FLAG);
         COUNT := COUNT+1
      END;
   LENGTH := COUNT
END;
```

The SUCC and PRED Functions

These two functions are fairly simple. They basically move the mark one element left or right respectively. If the move fails the run aborts through the HALT routine. These functions are simply extensions of the standard Pascal SUCC and PRED functions to handle L-lists.

```
FUNCTION SUCC (A: LLIST OF element-type): LLIST OF element-type;
VAR AFLAG: BOOLEAN:
BEGIN
   MOVE(A,1,AFLAG);
   SUCC := A;
   IF NOT AFLAG THEN HALT
END;

FUNCTION PRED (A: LLIST OF element-type): LLIST OF element-type;
VAR AFLAG: BOOLEAN;
BEGIN
   MOVE(A,-1,AFLAG);
   PRED := A;
   IF NOT AFLAG THEN HALT
END;
```

The DISTTO and DISTBACK Functions

These two functions are based upon the idea that the subtraction operator will refuse to delete until the L-lists are positioned so that the subtrahend is exactly aligned with the matching segment in the L-list

from which it is to be deleted. By counting the number of times the string mark must be shifted until this deletion occurs, we will get the distance to the next occurrence of the pattern.

```
FUNCTION DISTTO (LIST, PATTERN: LIST OF element-type): INTEGER;
VAR
   DIST: INTEGER;
   FLAG: BOOLEAN;
BEGIN
   DIST := 0;
   FLAG := TRUE;
   WHILE (LIST - PATTERN = LIST) AND FLAG DO
      BEGIN
         DIST := DIST + 1;
         MOVE(LIST,1,FLAG);
      END;
   IF FLAG
      THEN
         DISTTO := DIST (*FOUND A MATCH*)
      ELSE (*NO MATCH*)
         DISTTO := LENGTH(STR) + 1
END;

FUNCTION DISTBACK (LIST, PATTERN: LLIST OF element-type): INTEGER;
VAR
   DIST: INTEGER;
   FLAG: BOOLEAN;
```

```
BEGIN
   DIST := 0;
   FLAG := TRUE;
   WHILE (LIST - PATTERN = LIST) AND FLAG DO
      BEGIN
         DIST := DIST + L;    (* COUNT NUMBER OF MOVES *)
         MOVE (LIST,1,FLAG);
      END;
   IF FLAG
      THEN DISTTO := DIST
      ELSE     (* CHECK TO SEE IF WE HIT FIRST *)
         IF PATTERN = EMPTY
            THEN DISTTO := DIST - 1
            ELSE DISTTO := LENGTH(LIST) + 1
END;
```

A Simple Pascal Program Formatter

The previous examples demonstrated how the system primitives could be used to create some of the non-primitives. Although this is useful information, it is also valuable to see how the system functions on a more natural application. With this in mind the following simple Pascal program formatter was written. The purpose is to indent the programs such that each successively deeper block will be indented one level further.

The program scans the Pascal code looking for keywords that start or end blocks. It takes advantage of the MV-list structures and the

ability of ANYIN and LOOKUPIN to detect alternates. Each time a block starting keyword is found the formatting procedure increments the indentation level counter and recursively calls itself to process that block of the program L-list. Each time a block end is found the indentation level is decremented and the routine returns to the next higher recursion level. If it runs out of block ends the procedure prints the remainder of the code and then repeatedly returns until it finally exits to the main program.

The purpose of the main program is simply to build the MV-lists, input the program as a single long L-list and call the routine.

```
PROGRAM FORMAT (INPUT, OUTPUT);
VAR BLOCKEND, BLOCKSTART: LLIST OF LLIST OF CHAR;
   PROG: LLIST OF CHAR;
   CH: ARRAY[1..1] OF CHAR;
   I, INDLEV: INTEGER;
   FLAG: BOOLEAN;
BEGIN
   BLOCKEND := EMPTY
   INSERT(BLOCKEND, BUILD(' BEGIN ',7,1));
   INSERT(BLOCKEND, BUILD(' BEGIN;',7,1));
   INSERT(BLOCKEND, BUILD(' CASE ',6,1));
   INSERT(BLOCKEND, BUILD(' WITH ',6,1));
   INSERT(BLOCKEND, BUILD(' RECORD ',8,1));
   INSERT(BLOCKEND, BUILD(' REPEAT ',8,1));
   BLOCKSTART := BLOCKEND;
   INSERT(BLOCKEND, BUILD(' END ',5,1));
```

```
   INSERT(BLOCKEND, BUILD(' END;',5,1));
   INSERT(BLOCKEND, BUILD(' END.',5,1));
   WHILE NOT EOF DO
      BEGIN
         READ(CH);
         PROG := PROG + BUILD(CH,1,1);
      END:
   INDLEV := 0;
   FORM(PROG, INDLEV)
END.
```

The main program starts by building two MV-lists. BLOCKEND has included all of the keywords which begin or end blocks. BLOCKSTART has a subset of these keywords: those which start blocks. The program then reads in the Pascal code, character by character, building up one long L-list, PROG, which contains the entire Pascal program to be formatted. Next, the indentation level variable, INDLEV, is initialized to zero and then the formatting procedure is called.

```
PROCEDURE FORM (VAR PROG: LLIST OF CHAR; VAR INDLEV: INTEGER);
VAR SEGMENT: LLIST OF CHAR; DIST: INTEGER;
BEGIN
   SEGMENT := LOOKUPIN(BLOCKEND, BLOCKEND, PROG, DIST)
   IF DIST <= LENGTH(PROG)
      THEN  (* THERE ARE MORE KEYWORDS IN THE TEXT *)
         BEGIN
            FOR I := 1 TO DIST - 1
               BEGIN  (* WRITE OUT EACH CHARACTER UP TO THE NEXT    *)
```

```
      IF CURRENT(PROG) = ';'              (* KEYWORD       *)
         THEN  (* END OF A LINE *)
            BEGIN
               WRITELN(';');     (* TO TERMINATE THE LINE *)
               WRITE(' ':(INDLEV*3))  (* INDENT START OF *)
            END                          (* NEW LINE      *)
         ELSE
            WRITE(CURRENT(PROG));
                              (* WRITE OUT THE CURRENT CHAR *)
      MOVE(PROG,1,FLAG)    (* AND ADVANCE TO THE NEXT ONE *)
   END;
WRITELN;   (* FORCE THE KEYWORD TO START ON A NEW LINE   *)
IF ANYMATCH(BLOCKSTART, SEGMENT)
   THEN    (* ITS A BLOCK STARTING KEYWORD               *)
      BEGIN
         FOR I := 1 TO LENGTH(SEGMENT)
                              (* FOR LENGTH OF THE KEYWORD  *)
            BEGIN     (* WRITE EACH CHAR OF THE KEYWORD  *)
               WRITE(CURRENT(PROG));
               MOVE(PROG,1,FLAG)
            END;
         WRITELN;      (* FORCE NEW LINE AFTER KEYWORD    *)
         INDLEV := INDLEV + 1;   (* DROP DOWN A LEVEL     *)
         FORM(PROG,INDLEV)
      END
   ELSE   (* ITS A BLOCK ENDING KEYWORD *)
```

```
                BEGIN
                   FOR I := 1 TO LENGTH(SEGMENT)
                      DO      (* WRITE OUT EACH CHAR OF THE KEYWORD *)
                         BEGIN
                            WRITE(CURRENT(PROG));
                            MOVE(PROG,1,FLAG)
                         END;
                      WRITELN:      (* FORCE NEW LINE AFTER KEYWORD *)
                      INDLEV := INDLEV - 1      (* POP UP ONE LEVEL *)
                   END
             END
          ELSE   (* WE'VE RUN OUT OF KEYWORDS                          *)
             BEGIN
                FOR I := 1 TO DISTTO(PROG, LAST(PROG)) DO
                   BEGIN (* WRITE OUT EACH REMAINING CHAR IN THE TEXT *)
                      IF CURRENT(PROG) = ';'
                         THEN WRITELN (';')     (* END THE LINE *)
                         ELSE WRITE(CURRENT(PROG))
                                                (* WRITE THE CURRENT CHAR *)
                      MOVE(PROG,1,FLAG)    (* ADVANCE TO THE NEXT CHAR *)
                   END;
                WRITELN                    (* TERMINATE THE LAST LINE  *)
             END
END;
```

The comments provided with the procedure should be sufficient to explain how it works.

IV. COMPARISONS WITH OTHER SYSTEMS

The LISP Language

LISP is the most widely used of the stand alone list processors. Since the proposed string processing extension to Pascal is capable of performing many similar functions, a short comparison of the two systems is in order.

The LISP system has only one data type, the atom. An atom, however, may be a symbolic name, an integer or a real. In the Pascal L-list system no comparable type exists but we may declare a similar one through variant record fields:

```
TYPE ATOM = RECORD
                CASE TYPE OF
                   NAME : (NATOM: DICTIONARY);
                   INT : (IATOM: INTEGER);
                   REALNO : (RATOM: REAL);
            END;
```

Where DICTIONARY is another user defined type that is a scalar of all symbolic names that will be used. This presents one drawback of Pascal: since it is not a dynamically running system, names are fixed at compile time and cannot be created and deleted at will as in LISP. One solution would be to make the NATOM an L-list OF CHAR which then permits the dynamic name handling capability. This will also require the user to provide a few extra overhead routines to manage these names so that they function properly.

The LISP list structure is also a non-standard Pascal type, however a rough equivalent may be created by:

```
TYPE LINK= ↑LIST;
     LIST= RECORD
               CAR: RECORD
                       CASE CARFORM OF
                          LNK: (CARLNK: LINK);
                          ATM: (CARLNK: ATOM)
                    END;
               CDR: RECORD
                       CASE CDRFORM OF
                          LNK: (CDRLNK: LINK);
                          ATM: (CDRATM: ATOM)
                    END
          END;
```

Once again, management of this structure would be up to the user but it can be seen that with the exception of some of the dynamic and interpretive capabilities of LISP the Pascal system can be made to function in a similar manner. (Of course we could always just write a LISP interpreter in Pascal.)

Of more interest to us is the case where the LISP lists are linear rather than tree structures. Here we find that the proposed extension is more comparable to the LISP structure. A simple LLIST OF DICTIONARY or LLIST OF LLIST OF CHAR will provide a very similar form.

List manipulation facilities in LISP are very simple and also quite powerful. Although direct insertion and deletion are not provided, such functions as CAR, CDR, CONS and APPEND make it possible to create these operations relatively easily.

Pattern matching in LISP is almost all up to the user to provide. These are no provisions for an equivalent to alternate matching L-lists. The LISP user can create equivalent constructs but only with some difficulty. Property lists in LISP are quite similar to the MV-list/ RETRIEVE combination in the Pascal extension.

Compared to Pascal, LISP has a smaller set of control structures. LISP is highly dependent upon function calls and especially recursive function calls to manage flow of control. A fairly simple n-way sequentially evaluated branching statement is used to control flow within functions. In addition, the PROG feature provides a simple form of labels and GOTO's. There are none of the more structured looping constructs found in Pascal, which does permit those control structures along with n-way branches, recursive function calls (including the ability to pass functions as parameters to other functions) and labels and GOTO's. It should be noted that most modern LISP implementations have been extended to include structured control structures, similar to those in Pascal. These have not been standardized between implementations, however.

Although LISP lists can become quite a bit more complex in structure, Pascal L-lists can contain a much wider range of data types which can themselves grow to be quite complex.

Finally, it must be noted that LISP presents a problem in readability and is much less readable than the proposed Pascal system. This is primarily due to the language's basis in the lambda notation which, although it presents a concise form for representing lists, is a bit obscure to the more casual reader.

In summary, it should be noted that LISP and the proposed extended version of Pascal are directed toward two different areas. Despite this, each is capable of substituting for the other with some difficulty. Each of the systems has its good points. LISP is especially suited to dynamic operation where data may be interpreted as instructions and where structures may by dynamically allocated and de-allocated. The system is also better suited for handling more complex list structures such as trees.

The proposed Pascal L-list system is more oriented toward general processing with general data types. Alternate pattern matching is another of its strong points.

In the applications where the two systems overlap it seems that the Pascal system would be easier to use both because of its generality and its readability. It is thus that, in a wide range of applications, the Pascal extension would be found to be better suited than a LISP system.

The SNOBOL Language

SNOBOL is probably the most popular of the stand alone string processors. Unlike LISP, which deals with lists of atoms, SNOBOL is designed to handle strings of alphanumeric characters. Because its

designers concentrated on developing its ability to handle this data type, the language is rather lacking in other areas.

As compared to Pascal, SNOBOL is very limited in the range of data types that it permits. The standard SNOBOL types are integers, reals and strings. Booleans, Scalars, Sets, Subranges and Files are some standard Pascal types not provided in SNOBOL. Pointers are provided in a roughly equivalent form by the ability of SNOBOL to do indirect reference through string variables whose values name other variables. A form equivalent to the basic Pascal RECORD type is provided by the programmer-defined data type in SNOBOL. Variant field definitions are not permitted in SNOBOL as they are in Pascal, nor does the SNOBOL literature indicate whether nested record definitions are allowed. Arrays in SNOBOL are comparable to those provided in Pascal although only integer indices are allowed. SNOBOL does, however, permit the programmer to initialize all elements of an array to some specific value.

SNOBOL does have several data types not found in Pascal; these include PATTERN, TABLE, NAME, EXPRESSION, CODE and EXTERNAL. Expression variables contain strings which are unevaluated expressions. Code variables contain executable SNOBOL statements. Externals link to external routines. SNOBOL does not distinguish named constants as a special form, different from variables.

Flow of control in SNOBOL is fairly primitive: Statement labels, conditional and unconditional GOTO's are the only provisions for modifying sequential flow of control. The goto construct is simply a label enclosed by parens, following a colon at the end of any statement.

Preceding the label by an F or an S turns the goto into a conditional goto which branches on failure or success of the statement to which it is appended. Pascal, on the other hand, allows GOTO's IF-THEN, IF-THEN-ELSE, WHILE, REPEAT and FOR flow of control constructs. Functions in SNOBOL are comparable to those of Pascal but no equivalent to procedures is provided.

Strings in SNOBOL are most comparable to the extended Pascal type LLIST OF CHAR as defined in this thesis. These two structures are similar in that they are both chains of characters, however, the SNOBOL string has no empty element preceding it. Also, the SNOBOL equivalent of the mark is called the cursor and is automatically controlled by an element of the system called the scanner. The only control permitted to the user is that the cursor may be fixed in one location and then later be allowed to move again under control of the scanner. Because of the way in which the SNOBOL system functions, the automatic cursor movement poses no problem to the user in terms of control capability and makes the string operations very convenient to use. The proposed Pascal system, on the other hand, requires the user to manage cursor positioning. This gives the user a great deal of control over the system, providing abilities to limit the range of operations, and because of the overall design causes only minor inconvenience.

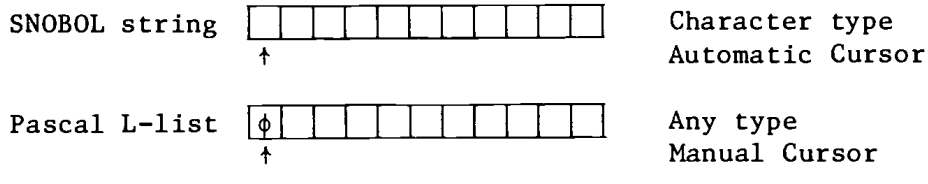


Figure 13. Comparison of SNOBOL and Pascal string forms.

Both systems provide an empty string construct. SNOBOL allows arithmetic to be performed directly on string representations of numbers, whereas the Pascal extension does not. SNOBOL allows strings to be built from literals directly by assignment and in expressions. The Pascal system requires the use of the BUILD function.

SNOBOL; variable = 'string literal'

Pascal: L-list variable := BUILD ('string literal', length start)

Concatenation in SNOBOL is a single operation, independent of cursor position, whereas the Pascal system must do an insertion which may require that the mark of one of the L-lists be moved to an end. In contrast, SNOBOL does not provide for direct insertion or deletion; however, the REPLACE function may be made to perform these operations. The Pascal system does not provide for direct replacement although delete and insert may be easily combined to substitute for this.

Pattern matching in SNOBOL is performed automatically by the scanner. Patterns may include alternate strings similar to the MV-lists of the Pascal system. Pattern matching is also performed automatically in the Pascal system although the user may also use the cursor position to limit the scope of the match. Alternates may be created in the Pascal extension through the use of an MV-list variable with the alternate L-lists inserted in it. Both systems permit scanning for a match of one pattern within another, testing two patterns for equivalence and comparing two strings for relative value (the Pascal system only permits relational comparison on L-lists with base types for which such comparisons are defined).

SNOBOL provides another construct, which Pascal does not, in the form of conditional assignment wherein an assignment can be set up which will not take place until a condition is met. When the condition occurs the assignment then takes place as a phantom operation. SNOBOL also provides a large number of functions and operations for manipulating strings which contain values that are really character representations of numbers. These provide the ability to detect such strings and to convert them to and from numeric variables. Also provided in SNOBOL is an EVAL function which takes a string which contains an unevaluated expression and evaluates it, returning the result. The Pascal system does not directly provide any of this since it is oriented toward the more general L-lists which may have elements of any data typed. To augment this to some extent, since it is recognized that LLIST OF CHAR will form a major portion of the system's usage, the ENCODE and DECODE functions were provided to allow the equivalent of printing or reading from one internal buffer to another with the associated type conversion performed between.

In summary, it would appear that these two systems simply take two different approaches to providing roughly the same amount of string processing power. The Pascal system, however, also provides the general data processing abilities of Pascal and L-lists which are more generalized. It is felt that this makes the proposed system much more suitable in terms of general applications than SNOBOL.

V. SUMMARY AND CONCLUSIONS

The system developed in this thesis provides a substantial amount of list and string processing power. It is at least equal in its string handling capabilities to the two currently most popular list/string processing systems. The combination of this with the general data processing abilities of Pascal makes it a very usable list/string processor.

Because of this the system is not restricted to the usual academic sort of applications. It is indeed capable of being applied for natural language understanding, computer language translation, recognizing formal grammars and the like but it may also be used in such mundane applications as parsing program commands, fancy formatting of output (such as what COBOL does) or even something as simple as splitting a person's name into first, last and middle initial fields.

This L-list construct can be used anywhere that a linear array or a manually maintained linear linked list could be used in a Pascal program. In addition, it is possible to simulate many nonlinear structures using this system. This is not to say that L-lists should be used in place of these other structures but simply that they may be. It is up to the programmer to decide when it is best to use an L-list instead of another structure.

The proposed design of the Pascal extension purposely made use, whenever possible, of the existing operators, functions and symbols in order to minimize the disruption of the language. Thus the extension turns out to be merely the addition of a new data type, a new constant, procedures and functions associated with it and redefinition of the

basic operators to recognize that type. Because the extended meanings of those operators were chosen to be similar to the corresponding interpretation when applied to the standard data types, it is felt that this system would provide a very natural means of augmenting Pascal with a list/string processor. It manages to maintain the "elegance" of the language.

BIBLIOGRAPHY

1. Abrahams, P. W., Symbol Manipulation Languages, Advances in Computers 9, Academic Press, New York, N.Y., 1968.

2. Bailey, M. J., Barnett, M. P., Burleson, P. B., Symbol Manipulation in FORTRAN - SASP I Subroutines, Comm. ACM 7, 6 (June 1964), 339-346.

3. Berztiss, A. T., A Note on Storage of Strings, Comm. ACM 8, 8 (Aug. 1965), 512-513.

4. Blackwell, F. W., An On Line Symbol Manipulation System, Proc. ACM 1967 National Conf., 203-209.

5. Bowlden, H. J., A List-Type Storage Technique for Alphanumeric Information, Comm. ACM 6, 8 (Aug. 1963), 433-434.

6. Bowles, K. L., Microcomputer Problem Solving Using Pascal, Springer-Verlag, New York, N.Y., 1977.

7. Breuer, J., Introduction to the Theory of Sets, Prentice Hall, Englewood Cliffs, N.J., 1958.

8. Druseikis, F. C., Doyle, J. N., A Procedural Approach to Pattern Matching in SNOBOL 4, Proc. ACM 1974 National Conf., 311-317.

9. Farber, D. J., Griswold, R. E., Polonsky, I. P., SNOBOL, A String Manipulation Language, J. ACM 7, 2 (Jan. 1964), 21-30.

10. Green, J., Symbol Manipulation in XTRAN, Comm. ACM 3, 4, (Apr. 1960), 213-214.

11. Griswold, R. E., Extensible Pattern Matching in SNOBOL 4, Proc. ACM 1975 National Conf., 248-252.

12. Griswold, R. E., String Analysis and Synthesis in SL5, Proc. ACM 1976 National Conf., 410-414.

13. Griswold, R. E., Poage, J. F., Polonsky, I. P., The SNOBOL 4 Programming Language, Prentice Hall, Englewood Cliffs, N.J., 1971.

14. Grogono, P., Programming in Pascal, Addison-Wesley, Menlo Park, Cal., 1978.

15. Guzman, A., McIntosh, H. V., Comm. ACM 9, 8 (Aug. 1966), 604-615.

16. Halmos, P. R., Naive Set Theory, Van Nostrand Co., Princeton, N.J., 1960.

17. Hammersly, P., A Note on the Implementation of LITHP on the ICT 1905, The Computer Journal 9, 2 (Aug. 1966), 173-174.

18. Harrison, M. C., Implementation of the Substring Test by Hashing, Comm. ACM 14, 12 (Dec. 1971), 777-779.

19. Housden, R. J. W., On String Concepts and Their Implementation, The Computer Journal 18, 2 (May 1975), 150-156.

20. Jensen, K., Wirth, N., Pascal User Manual and Report, Springer-Verlag, New York, N.Y., 1974.

21. Knowlton, K. C., A Programmer's Description of L6, Comm. ACM 9, 8 (Aug. 1966), 616-625.

22. Knuth, D. E., the Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison Wesley, Menlo Park, Cal., 1973.

23. Lukaszewics, L., EOL - A Symbol Manipulation Language, The Computer Journal 10, 1 (May 1967), 53-59.

24. Madnick, S. E., String Processing Techniques, Comm. ACM 10, 7 (July 1967), 420-424.

25. McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part L, Comm. ACM 3, 4 (Apr. 1960), 184-195.

26. Milner, R., String Handling in Algol, The Computer Journal 10, 4 (Feb. 1968), 321-324.

27. Mooers, C. N., TRAC, A Procedure Describing Language for the Reactive Typewriter, Comm. ACM 9, 3 (Mar. 1966), 215-219.

28. Newell, A., Tonge, F. M., An Introduction to Information Processing Language V, Comm. ACM 3, 4 (Apr. 1960), 205-211.

29. Perlis, A. J., Thornton, C., Symbol Manipulation By Threaded Lists, Comm. ACM 3, 4 (Apr. 1960), 195-204.

30. Smith, J. W., Syntactic and Semantic Augments to ALGOL, Comm. ACM 3, 4 (Apr. 1960), 211-213.

31. Trundle, R. W. L., LITHP - An ALGOL List Processor, The Computer Journal 9, 2 (Aug. 1966), 167-172.

32. Wegstein, J. H., Youden, W. W., A String Language for Symbol Manipulation Based on ALGOL 60, Comm. ACM 5, 1 (Jan. 1962), 54-61.

33. Weissman, C., LISP 1.5 Primer, Dickenson Pub. Co., Belmont, Cal., 1967.