MEMORANDUM
RM-3842-PR
OCTOBER 1963

A COMPARISON OF LIST-PROCESSING COMPUTER LANGUAGES

Daniel G. Bobrow and Bertram Raphael

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-3842-PR
OCTOBER 1963

A COMPARISON OF LIST-PROCESSING COMPUTER LANGUAGES

Daniel G. Bobrow and Bertram Raphael

This research is sponsored by the United States Air Force under Project RAND—contract No. AF 49(638)-700 monitored by the Directorate of Development Planning, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this Memorandum should not be interpreted as representing the official opinion or policy of the United States Air Force.

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA

PREFACE

This Memorandum is a comparison of four well-known list-processing computer languages, which are considered representative of the various list-processing languages available. List-processing languages are designed to handle problems involving the manipulation of complex data structures and which impose computer memory requirements that change in an unpredictable manner during computation.

The research, sponsored under U. S. Air Force Project RAND, was aimed at characterizing and evaluating list-processing languages in general and the four languages specifically considered; at isolating the areas of application for which particular list-processing languages are best suited; and at providing some criteria for the potential user in selecting one of the languages for his particular problem.

The authors, consultants to The RAND Corporation, are on the staff at the Computation Center, Massachusetts Institute of Technology.

SUMMARY

This Memorandum presents a detailed comparison of COMIT, IPL-V, LISP 1.5, and SLIP—four well-known computer programming languages which, among them, exhibit all the principal characteristics of existing list-processing languages. Important common features of list-processing languages are reviewed: forms of data structures which are manipulated; necessity for dynamic allocation of storage; use of pushdown stores; and use of recursive operations.

Principal differences between the four languages under consideration are detailed: representations of data, both by the programmer and within the machine; methods for storage allocation; programming formalisms and special processes available, including arithmetic facilities; and usability in terms of availability, documentation, learning aids, and debugging facilities. A rough comparison shows that the languages discussed are all of approximately the same speed.

Finally, the authors give some heuristics to aid in the selection of one of these languages for use in particular problem applications, concluding that no one of the languages considered is distinctly superior over all possible list-processing applications.

CONTENTS

I. INTRODUCTION

In the past five years computer research projects in such areas as artificial intelligence, simulation of human cognitive processes, mechanical translation, information retrieval, and operations research have generated problems involving a form of information processing which cannot be handled conveniently in any of the conventional computer programming languages.

"Symbol-manipulating" or "list-processing" computer languages have been designed to handle these special processing needs. We shall refer to these languages as "LP" languages. Many list-processing systems have been and are being developed, both as independent computer languages and as extensions of existing languages originally designed for other special problem areas; e.g., algebraic computation and job-shop simulation.

Four well-known LP languages — COMIT, IPL, LISP, and SLIP — have been selected for detailed study, since they illustrate almost all of the characteristics of existing list-processing languages. FLPL,[1] an early example of an LP language, contained many features found in the four languages chosen, but since it is no longer in use, it will not be considered. We shall discuss those versions of the four systems chosen for which documentation is available and which are implemented at the present time—i.e., COMIT 1,[2-4] IPL-V,[5-6] LISP 1.5,[7] and SLIP.[8] The first three of these languages have been in use for several years. All four have well-defined formalisms for describing problem solutions and are currently implemented on existing digital computers.

The advantages of list-processing languages and some common properties of these languages have been discussed by Green.[9] This Memorandum provides some definitions of terms, a review of the common features of LP languages, a discussion of the principal differences between the four languages under consideration, conclusions, and a comparison chart for reference.

II. DEFINITIONS OF TERMS

LP languages have been designed to facilitate the representation and processing of lists, strings, list structures, and binary trees. These terms are usually defined as follows:

A list is any sequence of elements.

A string is a list whose elements are not lists. A string is similar to a vector or array; that is, a string is an ordered sequence of elements. However, the number of elements in the string is not preassigned and may vary during a computer run. Strings may be used as a convenient representation of the sentences of a natural language; i.e., strings of characters or strings of words.

A list structure is a list whose elements may themselves be lists. Decision trees, mathematical equations, and list-processing programs have all been conveniently represented as list structures.

A binary tree is an extension of the idea of an ordered pair, with the further provision that each of the two elements of the pair may itself be an ordered pair, and so forth to an arbitrary finite depth. Such trees have been used to represent syntactic structures of sentences in natural language.

III. COMMON FEATURES OF LP LANGUAGES

A. DATA REPRESENTATIONS

The most important common feature of LP languages is that they deal with a special kind of data. Historically, the data used in computers have been numerical, in the form of either numbers or fixed-size vectors and arrays of numbers. Recent areas of computer research require the manipulation of more complex data structures. The data may contain symbolic as well as numeric information, and information is carried by the relational structure as well as the symbolic content of the data. The basic form of a data structure in an LP language formalism is usually a string, list structure, or binary tree. Occasionally, the most natural data form for a particular problem representation is not the same as the basic form used in the LP language which has been selected. This is usually not a serious problem since binary trees can be used to represent list structures, and vice versa, and strings can be represented by either.

B. STORAGE ALLOCATION

An important common feature of LP languages is that memory space for data structures need not be preassigned. Storage for each structure is allocated as it is needed. Cells are assigned to a structure dynamically, and are usually not sequential memory registers. Each new cell is added to the structure by creating a link or pointer from within the structure to the cell.

Since it must be possible to reassign the use of

memory cells during execution of a list-processing program, every LP language must contain the following:

(a) A store of cells available for use.

(b) Mechanisms for obtaining "new" cells from, and returning unneeded cells to, that store.

The languages under consideration all maintain the store of available cells in the form of a list. Basic processes within these languages automatically obtain cells from this "free storage list," thereby shortening the list. A cell (or list of cells) may be returned to the free storage list, that is, "erased," when it is no longer needed for computation. LP languages differ, as will be discussed below, in the ways in which cells are erased and the amount of attention which the user must devote to this erasure process.

The amount of space conveniently available for programs and data is limited by the size of the computer's core memory. Additional space can be obtained by using auxiliary storage, generally magnetic tapes. The auxiliary storage facilities available in the systems under discussion will be described below.

C. PUSHDOWN STORES

All four LP languages make use of pushdown stores or stacks. A pushdown store may be thought of as a string with the property that only the first element of the string is accessible. An element may be added to the store only by "pushing down" the store and placing the new element on the top of the stack. Conversely, when the first element is removed — i.e., the stack is "popped up" — the next element becomes available by becoming the first element.

D. RECURSIVE OPERATIONS

Programs which operate recursively are often necessary for constructing or processing list structures of arbitrary complexity. A recursive subroutine R is one within which the subroutine R itself may be called. If this happens, the old values of the arguments and partial results of R must be preserved while the inner computation with R proceeds. The values to be saved are placed in a pushdown store. The most recently saved partial results will be the first to be "popped up" — which is the desired order for performing recursions. Methods for defining recursive subroutines in particular LP languages will be discussed below.

IV. DIFFERENCES BETWEEN LP LANGUAGES

A. DATA STRUCTURES

In this section, we shall describe in detail the data structures of the four list-processing languages: COMIT, IPL, LISP, and SLIP. For each language, the programmer's representation of structures and the representation of these structures within the machine are considered.

COMIT

Programmer's Representation. The basic structures manipulated by a COMIT program are strings. The principal string being manipulated at any time is called the COMIT workspace, and is represented by a linear sequence of elements, or "constituents," separated by "+". We have, for example:

A + SIMPLE + LINEAR + STRING + OF + CONSTITUENTS

This string may be read from punched cards in the format shown, or the phrase may be written in the following simpler format:

A SIMPLE LINEAR STRING OF CONSTITUENTS

When the latter phrase is read, a COMIT string is created in which each letter and space is considered a constituent (where "—" represents the character "space"); e.g.,

A+—+S+I+M+P+L+E+—+...+O+F+—+C+O+N+S+T+I+T+U+E+N+T+S

Processes available to the programmer allow him to compress a number of constituents in the workspace into one constituent; e.g., the individual letters of a word into the word. All the characters on the keypunch are available to the programmer, and there is no restriction on the length of an individual constituent.

Constituents in the COMIT workspace may be given any number of subscripts, and these subscripts given values. An example of a constituent written with subscripts is:

TOM/ .25, INTERESTS WINE WOMEN SONG, JOB PROGRAMMER

In this example "25" is a numerical subscript and has no associated value. (The only way a number may appear in COMIT is as a numerical subscript.) "INTERESTS" is a subscript with three associated values; "JOB" has one. Constituents, subscripts, and values of subscripts may all be manipulated within COMIT, but although constituents in the workspace are considered to be ordered, subscripts and their values are manipulated as unordered sets.

In a limited sense, the COMIT workspace may be considered a list structure with three hierarchical levels: constituents, subscripts, and values. List structures of indefinite depth cannot be directly represented in COMIT.

Internal Representation. In the IBM 7090 implementation of COMIT, strings are represented internally by linked two-word blocks. One word of the pair contains BCD code for a constituent, or for part of a constituent if the constituent is longer than six letters. The other word contains a pointer to the next two-word block and markers indicating whether the item coded is a complete constituent, the last constituent in a string, the beginning of a constituent, etc.

IPL

Programmer's Representation. List structures defined by lists of symbols are the basic units of data used in IPL. The name of the list (an IPL symbol), and

the string of symbols composing that list, determine a data list. All symbols are either "regional" or "local." Regional symbols, written as a letter followed by up to four numerals, may be the names of lists or storage cells. Local symbols, written as a 9 followed by up to four numerals, are special symbols used for naming sublists belonging only to the list structure in which they appear. Symbols may also be labels for "data terms," cells containing numeric or alpha-numeric data. Data terms are manipulated by special built-in routines through reference to their labels.

Internal Representation. In the internal representation of IPL structures, each computer word is made up of four segments — two address-size fields, called "SYMB" and "LINK"; and two fields of three bits each, called P and Q. P and Q indicate whether the word is an element of a list or a data term. If it is a list element, SYMB contains an IPL symbol and LINK contains the address of the cell containing the next symbol in the list (an address that the programmer need not be concerned with). A cell whose LINK = 0 is interpreted as the last cell in a list. If the word is a data term, P and Q indicate the kind of data in the cell — i.e., a numerical or BCD term — and SYMB and LINK contain the data.

Corresponding to each external symbol is the address of some cell within the computer; that cell may itself be a list cell. A list cell contains a symbol (address) in SYMB and the address of another cell in LINK. Thus, except for data terms, IPL words can be thought of as binary trees; i.e., ordered pairs of addresses, each of which can name another ordered pair. However, by convention the IPL programmer generally uses the LINK of a cell only to point to the cell

containing the next item in a list, or to mark the end of a list. Therefore, the usual internal IPL data structure is a binary tree representation of a list structure.

LISP

Programmer's Representation. The basic elements in LISP are called atomic symbols, or atoms, and are strings of not more than 30 letters and/or numerals. The LISP programmer can link atomic symbols in binary trees or in list structures.

A binary tree may be represented explicitly by means of the "dot notation," where (A.B) is an ordered pair of the atoms A and B. In place of A or B (or both) the programmer may place any dotted pair of dotted pairs, etc., and thus explicitly define a binary tree; e.g.,

((E . F) . (H . (J . K)))

A LISP programmer can represent lists of atoms and lists of lists, etc., by means of the "list notation," which uses commas or spaces as syntactic markers. One example of the notation for a list acceptable in LISP is:

(PEOPLE, (BOYS, (TOM, DICK)), (GIRLS, (JANE)), (OTHERS,()))

Sublists are not named, but rather inserted directly into a list. The above list contains four elements; the last element is a list of two elements — one the atom OTHERS, and the other an empty list.

The list notation and the dot notation may be used interchangeably and simultaneously, with the understanding that a list (S1, S2,...,Sn) is equivalent to the dotted pair

(S1 . (S2 .... (Sn . NIL) ...))

The atom NIL plays a special role as a terminator of lists (it is defined to be equivalent to the empty list), just as a LINK of 0 serves as a terminator in IPL.

Internal Representation. LISP is similar to IPL in its internal representation, in that each data structure cell uses two address-size fields for pointers to sub-expressions. The internal representation for a binary tree in dot notation is an equivalent binary tree of computer cells. The representation for a list structure is the same as the representation for its equivalent dotted-pair binary tree.

An atomic symbol is represented internally by a special type of list called a property list which contains a special mark in its first cell. This list contains the external representation of the atom and some other special information. All of these are accessible to the programmer.

Atoms whose names begin with a numeral, such as 1.342, are made into special atoms within the system, and are recognized as numbers by distinctive markers on their property lists.

SLIP

Programmer's Representation. The basic data elements in SLIP are numbers and alphanumeric character strings whose length depends upon the word-length of the computer being used. These elements are organized into list structures which may be manipulated by the program.

SLIP differs markedly from the other languages discussed in that it is not an autonomous system, but

rather a set of subroutines embedded within a FORTRAN-type language. As a result, numerical data may be represented and manipulated using FORTRAN conventions. Also, the external representation of data is not fixed. Hollerith data can be read and inserted into lists according to a format statement. A routine is available which will create internal list structures from an external form similar to the LISP list notation. (Both IPL and LISP have special mechanisms for reading information in any format, but for most problems it is more convenient to put the data into the standard format than to write a special "read" program.)

Internal Representation. The internal representation of a list structure is also different in SLIP. An item on a list is represented by a pair of adjacent cells in memory. The first cell of the pair contains two address-length fields and a two-bit field which identifies the type of item. The address-length fields, called the left and right links of the module, contain pointers to the previous item on the list and the next item on the list, respectively. The second cell of each pair contains the actual list item, which may be either a full word of data or a pointer to a sublist. For each list a special pair of cells is created, called the header of the list. All references to a list as a whole are actually pointers to its header. The links of the header point to the first and last elements of the list, giving access to both ends of this symmetric list. Thus, internally a list structure in SLIP is a circular symmetric structure which may be traversed easily in either direction.

B. STORAGE ALLOCATION

In this section we shall discuss the mechanisms available in each of the four LP languages for returning unneeded cells to a free storage list, using pushdown stores and auxiliary storage.

COMIT

Maintenance of Free Storage. The COMIT formalism makes it impossible for the programmer to keep track of the actual list structures manipulated internally by the computer. Thus, erasures must be handled by the system. In the course of carrying out the list processing specified in a COMIT program, the COMIT interpreter keeps track of all storage cells used and returns them to the free storage list completely automatically as soon as their contents become unnecessary.

Pushdown Storage. In COMIT a string may be manipulated in the workspace, or it may be transferred to or from a temporary storage area called a "shelf." While the string is in the workspace, any part of it is accessible for manipulation by the program. However, a string on a COMIT shelf may only be used as a form of pushdown store. It is somewhat more general than a normal stack, in that elements may be "pushed down" onto either end of the string, although they can only be "popped up" from the beginning of the string.

Auxiliary Storage. COMIT has a mechanism for writing out onto tape and reading back into the workspace arbitrary data strings.

IPL

Maintenance of Free Storage. In IPL the programmer

has complete control of (and responsibility for) creating and erasing list structures. By convention, every sublist is "local" to a single list structure. The programmer usually specifies that the structure be erased (or otherwise processed) as a unit. Failure to explicitly erase unneeded list structures in IPL is a programming error and results in a loss of memory space for further computation.

Pushdown Storage. The IPL system provides several instructions for operating on pushdown stores. However, since no distinction is made in internal representation between stacks and lists, the programmer may treat either structure either way. One usually decides to consider certain IPL symbols as the names of push-down storage cells and others as the names of lists. However, it is possible, for example, to delete the first element of a list by executing a "pop-up-stack" instruction with the name of the list as the input.

Auxiliary Storage. IPL has a mechanism for writing out and reading back arbitrary data structures. The use of this method for obtaining more free space is simplified by a "trap-when-available-space-is-low" mechanism. Programs also may be stored in blocks on auxiliary storage. A block is automatically brought into core memory as soon as any program in it is needed.

LISP

Maintenance of Free Storage. LISP is similar to COMIT in that it is unnecessary, and usually extremely difficult, for the programmer to keep track of the actual storage used by his program. The ways in which cells become linked during a LISP run is so complicated that even the LISP system does not keep track of the

cells. Instead, the LISP system keeps using new cells until free storage is depleted, at which time a "garbage collector" sweeps through memory identifying those structures which are referenced by the program in its present state and reclaiming the rest for a new free storage list. This garbage collection process is somewhat time-consuming, but seems to be worthwhile in view of the fact that it frees both the programmer and the system from the detailed bookkeeping which would otherwise be necessary.

Pushdown Storage. In LISP, although the user is not directly concerned with specific pushdown operations, the system requires pushdown storage for its own purposes (e.g., controlling recursion). This pushdown storage is maintained, not as a list of cells linked from one to the next by pointers, as in most of the data lists described above, but rather as a sequential block of memory locations with a single pointer to its current "top." This procedure introduces an efficiency in the use of pushdown operations at the expense of minor rigidity in space allocation; namely, an arbitrary division of total storage into "available (list structure) space" and "pushdown storage."

Auxiliary Storage. LISP has no auxiliary storage facility.

SLIP

Maintenance of Free Storage. As in IPL, SLIP leaves responsibility for erasing unnecessary lists with the programmer; however, no distinction is made between main lists and sublists. The system allows multiple use of any list, and solves the erasure problem by placing, in the header of every list, a "reference count"

which indicates the number of times that list is named in existing data structures. Each SLIP erasure of a cell naming a list reduces the reference count of that list, and when this count reaches zero the list cells are returned to the free storage list. The process of returning a list of cells to the free storage list is much faster in SLIP than in IPL because in IPL it is necessary to read through all the cells of a list in order to locate its end; in SLIP the location of the end of a list is immediately available from the header of the list. Furthermore, the SLIP system does not take time to erase a sublist of an erased list structure until the cells of that sublist are really needed; i.e., until the cell naming the sublist appears at the top of the free storage list.

Pushdown Storage. No distinction is made in SLIP between pushdown storage and ordinary lists. Pointers may reference the interior of any list, and information may be added to or removed from either end of any list through use of "push" and "pop" instructions.

Auxiliary Storage. SLIP, like COMIT, can transmit arbitrary data structures between core memory and auxiliary storage.

C. PROGRAMMING FORMALISMS

Most computer programming languages may be described in the following terms:

(1) Description of processes which operate on the data;

(2) Specification of flow of control from one process to another;

(3) Communication of arguments (inputs) from one process to another;

(4) Combination of processes into more complex processes (use of subroutines).

In addition, LP languages have various degrees of facility for handling special kinds of processes, including recursive, self-modifying, and arithmetic procedures. In the following we shall describe these features of the four languages from the programmer's point of view.

COMIT

Basic Program Format. The basic unit of a COMIT program is the COMIT "rule" which usually defines a desired transformation of a particular data string (called "the workspace"). The transformation is specified, not by giving a procedure (sequence of instructions) for modifying the data, but rather by describing the desired input and output forms of the data. When a rule is executed, the workspace is searched until a substring is found which matches the input form of the rule. The COMIT interpreter then transforms that substring into the output form specified by the rule, and control is transferred to a rule whose name is given at the end of the current rule. If no matching substring can be found, the workspace is left unchanged and control proceeds to the next rule in sequence. Rules may also specify selection of workspace elements by means of their subscripts, and specify changes to the subscripts and subscript values. The string in the workspace is the only input for transformation rules, but rules may also give instructions for transferring data strings between the workspace and various temporary storage "shelves." Groups of rules may be used as subroutines, if properly organized according to somewhat awkward linkage conventions.

Special Processes. Recursive processes may be

described in COMIT but the techniques, which involve saving partial results and subroutine returns on push-down shelves, are clumsy to express in the formalism.

Programs are self-modifiable only in the limited sense that pre-programmed rules may be chosen for execution by computations which find names of those rules in the workspace.

Arithmetic is extremely awkward in COMIT since arithmetic arguments and results can only be integers and must be located as subscripts of workspace items.

However, the need for these special kinds of processes seems to arise very rarely in those problems best suited to COMIT; i.e., those which involve string rather than list structure manipulations.

A special feature of COMIT which is useful in language processing is the "list rule" — a device which enables rapid dictionary search. When workspace items are to be looked up in a dictionary-like list, the list may be made into a "list rule." The elements in the list are sorted by the COMIT system, which then performs searches in a logarithmic rather than a linear fashion.

IPL

Basic Program Format. An IPL routine consists of a list of instructions with occasional two-way branches. The flow of control is always to the next sequential instruction, except at a branch point, where the choice is determined by the contents of a test-cell which may be set by the program. A routine terminates when the end of the branch being executed is reached. An instruction may be the name of a basic operation which, when executed, changes the state of a storage stack or data list, or it may be the name of another routine. In the latter case, the named routine is executed until termination

and then control returns to the next instruction in the calling routine. The executed routine may in turn contain instructions which execute other routines, and so on indefinitely. There is no distinction between a main program and a subroutine, since any routine may execute any other. The IPL interpreter keeps track of the current instruction location by means of a push-down store, and terminates only when the highest-level routine terminates. The programmer is responsible for explicitly communicating arguments between routines (by placing them in a pushdown store called the "communication cell"), and avoiding conflicts in references to temporary results (usually by including **instructions in appropriate places to "push down" or "pop up" storage cells which are used independently at different levels).**

Special Processes. Recursion is reasonably easy in IPL since any routine may execute itself as a subroutine, provided it first "pushes down" all its partial results.

Program modification is possible since the names of routines are ordinary IPL-V symbols and the system provides an instruction for executing a routine whose name is obtained from the data. Basic operations included in the IPL system make it possible to construct arbitrary lists; in particular, an IPL program may construct a list at run-time which is in the correct format for a routine, and then execute that routine. Since a program may also manipulate existing routines (by treating them as special data lists), IPL programs are completely self-modifiable.

Arithmetic is made possible in IPL through use of a special group of instructions for manipulating data

terms. The formalism is somewhat confusing since one must always refer to the IPL symbol which names the cell containing a number, rather than refer to the number itself. Since the P and Q parts of a data term cell are used to specify the kind of data contained, the data itself only occupies part of the cell and must be in some special format. IPL arithmetic instructions must interpret these formats as well as carry out their operations, and are therefore extremely slow.

Two additional special features have proved quite useful to many IPL programmers — "description-list processes" and "generators." Any list may have associated with it a "description-list" — a list of pairs of elements. Basic IPL operations can add pairs to description lists; others retrieve the second element of a pair on a description-list, given the first element and the names of the main list. Thus, description-list operations simulate an associative memory containing arbitrary descriptive information for any IPL list.

Frequently a programmer wishes to perform the same operations on each member of a sequence of symbols, where just the process of producing the next symbol in the sequence requires a complicated program. This latter program may be written in the form of a "generator," which can be used to generate a particular kind of sequence of symbols as inputs for any other program. For example, one may write a generator to generate the third symbol on each sublist of any specified list structure. The generator frees the programmer from having to worry about changing contexts; i.e., conflict between generator and main program in the use of temporary storage.

LISP

Basic Program Format. LISP programs consist of functions, rather than sequences of instructions or descriptions of data forms. In order to be evaluated, a function must be given its specified number and types of arguments; from these it constructs a single data structure which is its value. The basic functions in the system produce values which are easily obtained from their arguments; e.g., the value may be a particular subexpression of an argument, or a list of the arguments. New functions may be defined by combining built-in functions in certain ways described below. A complete LISP program generally consists of a set of function definitions followed by the application of these functions to particular data structures as arguments. The values of these functions are the results of the computer run. Functions may be combined in the following ways:

(1) Composition. The value of a function may itself be an argument of some function, so that the inner function must be evaluated before the main function can be evaluated. This nesting of functions may occur to any depth.

(2) Conditional Expressions. This is an n-way branch, similar to the Algol statement,

$$\underline{\text{if}}\ p_1\ \underline{\text{then}}\ e_1\ \underline{\text{else if}}\ p_2\ \underline{\text{then}}\ e_2\ \ldots,$$

of arbitrary length, where the p's are predicates (functions whose possible values are the special truth-value symbols "T" and "F"), and the e's are any LISP expressions. The value of the conditional is the value of the e following the first p whose value is "T".

(3) Recursion. A LISP recursive function is a function defined by a conditional

expression, part of which requires an evaluation of the entire function (for different values of its arguments). For example, the factorial function may be defined by a LISP translation of the following:

"n! is defined by,

if n = 0 then value is 1;

else value is the value of n·[(n − 1)!]"

The second use of the "!" symbol automatically refers to the entire definition. We have used the function "factorial" here because of its familiarity. Although "factorial" can be defined without using recursion, other functions which can only be defined recursively are just as easily expressible in LISP. The programmer need never be concerned with such details as precisely how data structures are manipulated, which temporary results may be erased, and what information must be saved in pushdown storage.

Special Processes. Recursion is extremely easy to use in LISP, and in fact is one of the basic methods for defining functions (see above).

In the LISP formalism function definitions are represented by list structures; i.e., functions are structurally equivalent to data. Also, the basic system includes an evaluate function which interprets its argument as a list representation for a function, and executes the function. Therefore, programs are completely self-modifiable in the sense that, at execution time, function definitions may be changed, or created, and then applied to arguments.

The LISP system contains a fairly complete set of basic arithmetic functions which are easy to use if one can become accustomed to Polish prefix, rather than the usual infix notation for algebraic expressions. However, since LISP numbers are atomic symbols whose

numeric values must be found on their property lists before any actual arithmetic is performed, LISP arithmetic is significantly slower than, say, FORTRAN arithmetic.

A useful property of LISP is that one may include functionals, or functions whose arguments may include function definitions. For example, one may define a function of two arguments "map[x;f]," whose value is a list of the values of the function f applied to each element of the list x.

Another special feature of LISP is the "program feature." This is useful for those occasions when operations should be performed which are awkward to express in the basic function and conditional-expression notation; e.g., labeling temporary results to avoid duplicate computations, and executing functions which are needed for their effects rather than values, such as input-output operations. In the "program feature," a function is defined by a list of program steps with FORTRAN-like mechanisms for assigning values to variables and controlling sequence of operations.

SLIP

Basic Program Format. SLIP consists of a set of pseudo-functions which may be added to any basic FORTRAN system ("pseudo" because they change internal structures as well as return values). SLIP programs are like those in IPL in the sense that they consist of sequences of instructions to manipulate internal structures; the programmer has the same problems of keeping track of processed expressions, erasing or protecting data, etc.

The syntax of SLIP is that of FORTRAN, with flow of control determined by "IF" and "GO TO" statements. The FORTRAN nesting of functions gives SLIP a LISP-like power for composition of list-processing functions. The SLIP "reader" mechanisms give the programmer the ability to set up pointers which can automatically be stepped forward or backward through list structures in fairly complicated ways. Whether the additional power and flexibility offered by these mechanisms are worth their cost in additional memory space taken up by the SLIP internal data representation, and therefore how SLIP compares to IPL as a pure list-processing formalism, will not become clear until more programming experience is gained in using the SLIP language.

Special Processes. Recursion in SLIP may be done almost as easily as in IPL through use of a special "visit" function and the FORTRAN "assign" statement. The programmer still must specify explicitly the arguments to be transmitted and the temporary results to be saved.

A SLIP program may be "self-modifying" in the following limited sense: The order in which preprogramming segments of routines are executed may be determined at run-time by means of arbitrary symbol-manipulating procedures. More general program modification is not possible since the FORTRAN compiler is unavailable at run-time.

The arithmetic facility of SLIP is exactly that of FORTRAN, which of course is superior to that of any of the pure LP languages. If a problem involving extensive arithmetic processing were programmed in COMIT, IPL, or LISP, then the arithmetic portion of the program would be harder to read than the corresponding FORTRAN

statements. In addition, if SLIP FORTRAN-compiled arithmetic operations were to be used for "production runs," much computer time would be saved compared to using the corresponding operations in the other LP languages. SLIP, which provides list-processing facilities as part of an algebraic compiler, is designed for those problems which require an intermingling of symbol manipulation and extensive arithmetic processing.

A substantial part of the SLIP system consists of functions for character-manipulation and bit-manipulation. These partial-word operations may be used in conjunction with or independently of the SLIP list-processing mechanisms.

D. USABILITY

The "usability" of a computer language depends upon the availability of the language on an accessible computer, and the difficulty of learning the language and debugging programs in it. The relative ease with which new programming languages may be learned and used depends upon several things, including the quality of the documentation, the quality of the instructor (if any), and the background and interests of the student, as well as the particular characteristics of the languages. More objective factors include extent of documentation and automatic debugging aids.

In this section we shall discuss the available implementations, the documentation, and the special aids available for learning and using each of the four languages.

COMIT

Implementation. COMIT is implemented on the IBM 7090 and IBM 704 as follows: A compiler translates the COMIT rules into an intermediate machine representation; an interpreter then executes this form of the program. Instructions are available for installing a COMIT system under an existing "monitor" or "automatic operator" system.

COMIT II, which will be released shortly, will have a much faster compiler than the present COMIT. It will also allow the use of FORTRAN-compiled subroutines and more flexible use of subscripts.

Documentation and Difficulty. COMIT is probably the easiest LP language to learn, due to both its simple formalism and its excellent manual, Introduction to COMIT Programming. The well-organized COMIT Programmers Reference Manual is also available. A program which grades the solutions to COMIT programming problems and an elaborate set of diagnostics built into the system also aid the novice programmer. There are no automatic devices for helping a programmer locate logical errors, but various debugging aids may be easily written in COMIT.

IPL.

Implementation. IPL is available as an interpretive system on the IBM 7090, Control Data 1604, UNIVAC 1105, Bendix G-20, Philco 2000, AN/FSQ-32, and IBM 650.* A compiler for the 7090 version was written experimentally, but was discarded since the resulting object code occupied much more space and ran only slightly faster than the interpreted version.

The IBM 7090 version of the IPL system is itself a subroutine so that it may easily be included in any monitor system.

Documentation and Difficulty. IPL is an easily obtainable and widely used LP language. The programming manual[5] consists of two parts,** "Elements of IPL

---

*See Refs. 5 and 6 for further information.

**A new edition of the IPL-V Manual is being completed at the time of this writing.

Programming" and "Programmers Reference Manual." The "Reference" section is quite usable. The "Elements" is an adequate teaching aid, but leaves some things to be desired if it is to be used without a teacher. Coding problems, their solutions, and "TIPL," a grading program for these solutions will be available with the new edition of the manual. Experience at summer institutes held at The RAND Corporation indicates that these problems and TIPL can be of great assistance in an IPL programming course. The IPL system has several convenient built-in facilities for aiding in program debugging. These include both selective and snapshot trace facilities. IPL is very much like a machine language for some symbol-manipulating machine; therefore, an experienced programmer would probably find it much easier to learn than would a novice in the computer business. Since an IPL program describes symbol manipulation at a very basic level, the programmer has to do considerable detailed bookkeeping. The lack of mnemonic symbols is an annoying anachronism.

LISP

Implementation. The only LISP 1.5 system currently available consists of a monitor, interpreter, and compiler for the IBM 7090 (although other LISP systems are being prepared). The compiler may be erased, ignored, or used to translate function definitions from list structure form into machine language subroutines. The monitor controls restoration of available space and initiation of independent function evaluations. Compiled and interpreted functions communicate with each other automatically. Annotated "FAP" listings are available which describe the changes which

must be made in order to install LISP under a monitor system.

Documentation and Difficulty. LISP is quite different from any other programming system; this probably largely accounts for the popular belief that it is an extremely difficult language to learn. Actually, for a completely naive user, LISP is probably no more difficult to learn than IPL or SLIP; it may be less difficult if the user is mathematically inclined. A programmer experienced in conventional computer languages may have particular difficulty in breaking old habits in order to think in LISP terms. Unfortunately, the only documentation for LISP is the LISP 1.5 Programmer's Manual[7] which is adequate for reference purposes but not very good as a text. However, considerable digging for information may pay off in the long run since the LISP formalism has certain distinct programming advantages. LISP gives the programmer a powerful notation for describing recursive processes, yet frees him from concern about protection of arguments and maintenance of free storage. Thus, for one who knows the language, it is significantly easier to describe many complex symbol-manipulating processes in LISP than in any other available formalism. Since programs are described at a high level and most detail work is handled automatically, LISP programs are generally quite easy to debug, especially with the use of LISP's "trace" feature. This trace gives the values of the arguments of a function each time the function is entered, and the value obtained for these arguments.

SLIP

Implementation. SLIP consists of a set of FORTRAN subroutines, most of which are written in FORTRAN. In order to get a SLIP system working on any machine which has a FORTRAN or FORTRAN-like compiler available but for which SLIP is not yet available, one must hand-code a few special subroutines. This is an easier job than that of writing a complete interpreter or compiler for one of the other LP languages. Notice that SLIP is then automatically included in any FORTRAN monitor system and thereby avoids a frequent source of conflict between user and computer installation management. Since SLIP is just a set of subroutines, only those which will be used must actually be provided for any particular computer run. Thus, for example, if character-manipulation functions are not needed they may be omitted, leaving room for a longer initial available space list. Complete SLIP systems already exist for IBM 7090 FORTRAN II, Control Data 1604 FORTRAN, and the Stanford version of BALGOL. Others are in preparation.

Documentation and Difficulty. SLIP is a new system. Its only documentation at this writing is a preliminary report, Symmetric List Processor,[8] which is somewhat incomplete. A SLIP manual is scheduled to be published soon in the Communications of the ACM. Various improvements in the language and its implementations are now in progress.

However, the language is really just FORTRAN with some frills; any FORTRAN programmer (and there are a great many!) can become a list-processing programmer simply by learning the frills, a much easier process than learning a completely new "foreign" language. Unfortunately, the "mnemonic" names for SLIP functions are obscure since they must obey FORTRAN naming restrictions. Actual list-processing operations are carried out at the same level of detail in both SLIP and IPL, and therefore may be just as difficult conceptually in both.

No special debugging aids are available in SLIP except the basic FORTRAN diagnostics.

E. EXECUTION TIMES

It is extremely difficult to obtain definite information concerning the relative speeds of LP languages. One would have to choose problems which could be solved equally well (whatever that means) in each of the languages being compared; obtain independent solutions in each language by programmers of the same caliber; record accurate times in systems with no built-in timing mechanisms; and decide how to weight time for execution vs. compilation, assembly, tape-spinning, etc. To our knowledge, no such comparison has yet been attempted. However, in order to justify an intuitive feeling that running times for the various languages are at least in the same ballpark, and therefore that speed of execution is only a minor consideration in comparison with ease of programming and debugging, especially for research applications, the following experiments were conducted:

(1) Ackerman's function,[10] a highly recursive arithmetic function of two arguments, was programmed and executed for the same arguments in COMIT, IPL, LISP, and SLIP (IBM 7090 versions).

(2) A routine used in the Logic Theorist[11] was programmed and executed in IPL and LISP. This program creates a list structure of names of Boolean expressions based on the structures of these expressions.

The results of these experiments are given in Table I. The reader is reminded that Table I gives approximate information about specialized problems.

Table I

APPROXIMATE TIME COMPARISONS

Language	Ackerman's function of 3,3				List-structure building	
	COMIT	IPL	LISP	SLIP	IPL	LISP
Time to compile or assemble	15 sec[a]	10 sec[a]	8 sec[a]	20 sec[a]	b	10 sec[a]
Execution time	20 sec[a]	9 sec	5 sec	2.6 sec	2 min	40 sec
Print-out Time					1 min	10 sec[a]
Approximate number of debugging runs[c]	0	2[a]	0	1[a]	12[a]	6

[a]Particularly approximate estimate; i.e., ±50%.

[b]Pre-assembled program was used.

[c]Rough estimate for an experienced programmer.

It may be used as a rough guide, but should not be considered a valid basis for any general conclusions.

V. CONCLUSIONS

Table II, the comparison chart on the next two pages, summarizes the details presented in this Memorandum. In this section we present heuristics we feel are important in choosing the LP language best suited for a particular problem. Almost any problem can be coded in any of the four languages discussed, or for that matter, in almost any programming system, including binary machine language, provided the programmer is sufficiently clever and ambitious. In choosing a language, one should consider the characteristics of both the problem and the programmer. Intimate knowledge by the programmer of any one of the LP languages will probably be an overriding factor toward the choice of that language for any LP problem. With this in mind, let us look at problem characteristics which tend to make a solution more easily expressible in one of the languages than in any of the others.

If the data is in the form of strings of alphanumeric characters — e.g., natural language text — and the operations to be performed on this data are string manipulations, such as substitution, rearrangement, and duplication, then COMIT is a natural choice of a programming language. If extensive arithmetic or statistical operations are also to be performed, then COMIT's awkward arithmetic facilities would be a hindrance, and if these operations constitute the bulk of the task, then perhaps the superior FORTRAN arithmetic in SLIP would make it worthwhile to use SLIP, even for the string manipulations. (However, we understand that COMIT II, soon to be available for the 7090, will have facility for incorporating FORTRAN-compiled functions.)

Table II

COMPARISON CHART

ITEM \ LANGUAGE	COMIT	IPL	LISP	SLIP
What are data representations? Programmer:	Strings of items which may have subscripts.	Lists of elements which name data terms or other lists.	Symbolic expressions (parenthesized list structures and dotted pairs).	Parenthesized list structures.
Internal:	Linked two-word blocks.	Binary trees (i.e., two address fields per word) usually representing list structures.	Binary trees; i.e., two address fields per word.	Headed lists of two-word blocks linked both ways.
Are common sublists allowed?	No, but this is unimportant to the user.	Yes; by convention each is "local" to a particular structure for copying and erasing purposes.	All structures and substructures can be referenced freely.	Only headed lists can be used as sublists.
How are cells no longer needed made available?	Automatically as soon as they are available.	By the programmer through "erase" statements in the program.	Automatically by "garbage collection" when free storage is exhausted.	By the programmer through "erase" statements in the program.
What are the auxiliary storage facilities?	Data may be written and read from magnetic tape.	Data and blocks of programs may be written on and read from magnetic tape.	None available.	Data may be written and read from magnetic tape.
What is the form of a program?	A labeled sequence of rules for string transformations.	Lists of basic instructions and names of subroutines.	Definitions of LISP-functions, and functions applied to arguments.	FORTRAN program with special FORTRAN pseudo-functions.

Table II (Continued)

ITEM \ LANGUAGE	COMIT	IPL	LISP	SLIP
Are recursive operations definable and usable?	Awkwardly.	Yes.	Very easily, partly because temporary results are saved automatically during recursion.	With some difficulty.
Can a program modify itself?	Slightly.	Yes.	Yes.	Slightly.
Can mnemonic symbols be used?	Yes.	No.	Yes.	Only those allowed by FORTRAN.
How is the arithmetic facility?	Very poor; numbers must be integer subscripts of string symbols.	Poor; awkward representation and slow interpretive execution.	Fair; Polish notation for representation. All numbers indirectly addressed at execution.	Good; i.e., FORTRAN arithmetic.
Implementation.	IBM 709/7090 IBM 704/7040	IBM 709/7090 Control Data 1604 Bendix G-20 Philco 2000 IBM 650 UNIVAC 1105 AN/FSQ-32	IBM 709/7090	FORTRAN and hand-coded FORTRAN functions for IBM 709/7090 and Control Data 1604.
Documentation and Teaching Aids.	Excellent Introduction to COMIT Programming and Reference Manual, marking program for sample problems.	Elements of IPL Programming and a Reference Manual, marking program for sample problems.	LISP 1.5 Reference Manual and annotated FAP listing.	SLIP report containing not extensively tested programs.

If some string manipulation is to be done, but processing of complex list structures is also needed, LISP is probably a good language to use. There is documented and available a LISP function, "METEOR,"[12] which performs COMIT-type string manipulations and allows full use of other LISP facilities.

Very recursive processes are most easily expressed in the LISP formalism, and programmers with mathematical training will probably find it the most natural. Extensive experience at Hughes Aircraft has indicated that LISP is a good language for expressing solutions to LP problems and testing ideas for solutions. Then, if the ideas prove feasible, more efficient production programs or ones incorporating auxiliary storage may be constructed in another language using the LISP program essentially as a flowchart.

Programs working with complex list structures and needing auxiliary storage to handle large amounts of data should probably be written in IPL or SLIP. It is difficult to choose between these two languages, except that much more background information and documentation is available for IPL that for the newcomer, SLIP. For ease of insertion of an LP language into a monitor system and for programmers brought up on a FORTRAN diet, SLIP is an excellent starting language. A principal disadvantage of SLIP is that very little experience has been accrued with it, and, further, some care must be taken to fool a too clever FORTRAN compiler.

To reiterate, no one of these LP languages is distinctly superior over the entire range of problems for which LP techniques are needed. They all provide a sufficient number of built-in operations so that a programmer may directly express and deal with his own problem in the formalism of the chosen language without having to prepare many basic utility routines.

REFERENCES

1. Gelernter, H., et al., "A Fortran-Compiled List Processing Language," J. ACM, Vol. 7, No. 2, April 1960, pp. 87-101.

2. Introduction to COMIT Programming, Research Laboratory of Electronics and MIT Computation Center, MIT Press, Cambridge, Massachusetts, 1961.

3. COMIT Programmers Reference Manual, MIT Press, Cambridge, Massachusetts, 1961.

4. Yngve, Victor H., "COMIT," Comm. ACM, Vol. 6, No. 3, March 1963, pp. 83-84.

5. Newell, Allen, (ed.), Information Processing Language-V Manual, Prentice-Hall, Englewood Cliffs, New Jersey, 1961 (being revised).

6. Newell, Allen, "Documentation of IPL-V," Comm. ACM, Vol. 6, No. 3, March 1963, pp. 86-89.

7. McCarthy, J., et al., LISP 1.5 Programmer's Manual, MIT Computation Center and Research Laboratory of Electronics, 1962.

8. Weizenbaum, J., "Symmetric List Processor," Comm. ACM, Vol. 6, No. 9, September 1963.

9. Green, B.F., "Computer Languages for Symbol Manipulation," IRE Trans. on Human Factors in Elec., Vol. HFE-2, No. 1, March 1961, pp. 2-3.

10. Kleene, S.C., Introduction to Metamathematics, Van Nostrand, Princeton, New Jersey, 1952.

11. Stefferud, Einar, The Logic Theory Machine: A Model Heuristic Program, The RAND Corporation, RM-3731-CC, June 1963.

12. Bobrow, D.G., METEOR: A LISP Interpreter for String Transformations, Memo 51, Artificial Intelligence Project, Research Laboratory of Electronics and MIT Computation Center, Cambridge, Massachusetts, April 29, 1963.