NATIONAL BUREAU OF STANDARDS REPORT

8163

CHARACTERISTICS OF FOUR LIST-PROCESSING LANGUAGES

by

R. W. Hsu

To the

National Science Foundation
(Grant No. GN 107)

NBS

U. S. DEPARTMENT OF COMMERCE
NATIONAL BUREAU OF STANDARDS

NATIONAL BUREAU OF STANDARDS REPORT

NBS PROJECT
11-11-11404

NBS REPORT
8163

September 1, 1963

8163

CHARACTERISTICS OF FOUR LIST-PROCESSING LANGUAGES

by

R. W. Hsu

Applied Mathematics Division

To the

National Science Foundation
(Grant No. GN 107)

NBS

U. S. DEPARTMENT OF COMMERCE
NATIONAL BUREAU OF STANDARDS

Contents

CHARACTERISTICS OF FOUR LIST-PROCESSING LANGUAGES

by

R. W. Hsu

I. INTRODUCTION

This is a compilation of some characteristics of four list-processing languages, FLPL, IPL-V, LISP, and COMIT.

Generalizations concerning the suitability of particular languages to particular classes of programming situations tend to be subjective and dependent on the accidents of one's experience. Similarly unreliable are impressions as to the ease and difficulty of learning or using different languages. These judgments will be avoided as much as possible; instead, this report lists in parallel for the four languages certain of their properties that would be of particular interest to a user in choosing a language for a given job. Among the considerations excluded by this limitation of scope are theoretical aspects of the languages, such as completeness and minimality, and the methods of implementation, except insofar as they affect the user.

To prevent this paper from becoming a programming manual for four languages, not every feature referred to is explained in detail. Some familiarity with the languages is assumed, and key terms are given so that further details on them may be looked up in the respective manuals. Terms technical to a particular language are always underlined.

The author would be grateful for additions and comments, especially concerning any inaccurate statements made in this report.

The author's experience with these languages has been gained during the course of the last three years as an employee of the National Bureau of Standards, as an employee of the Computer Center at the University of California at Berkeley, and as an IBM-WDPC research assistant at the University of California.

FLPL (Fortran List Processing Language)

This is a set of about 50 list-processing subroutines in machine language, for use with Fortran. Thus the user has all of the power and facilities of Fortran, in addition to list-processing capability. The (binary) deck of subroutines is available for 704 and 709-90. The functions for 709-90 FLPL are slightly different from those of 704 FLPL. In this paper we will be talking about

704 FLPL, since this is the version described in the published descriptions of the language. However, since FLPL is only a set of list-processing functions, and not a complete language with control statements, input-output, etc., the actual language we will be considering will be Fortran augmented with the FLPL functions.

IPL-V (Information Processing Language, 5th, in a Series of IPL's).

This language is very much like a machine language, with a repertory of some 150 list-processing, arithmetic, input-output, and housekeeping instructions. It has extensive tracing facilities and subroutining (including recursion) facilities. The current implementations of the language are almost all interpretive, and they are available on some dozen different machines. IPL-V is probably the most widely used of the four list-processing languages considered here. Many new features, mainly in the form of new instructions added to the repertory, have been adjoined to individual implementations of the language since the original language was defined in the programmer's manual. The language we consider here is that of the manual, but mention will be made of some of the additional features, and some of the restrictions on the original system, that have come to the author's attention.

LISP (List Processor)

This language offers a compact functional notation, and can express manipulations on list structures represented in a certain symbolic fashion. Although the basic vocabulary of the language is very small, very powerful functions can quickly--say, in three or even two levels--be built up from them using the very powerful means of combining expressions available in the language. The language is basically interpretive, and the interpreter can itself be exhibited as a LISP function. Compilers have, however, also been built for compiling LISP functions into machine language routines. A 704 version of LISP is available, although it is said that it was never completely debugged. A 709-90 version, LISP 1.5, which includes several features not described in the LISP I manual was released recently, too late for detailed consideration in this report. Another system, MBLISP, is available. It also contains some features not described in the LISP I manual, as well as some restrictions. In particular, it does not have the compiler option and the program feature. A LISP system is also being implemented for the IBM 7070. The language we will be concerned with here will be that of the LISP I manual, and mention will be made, where relevant, of new features and restrictions that have come to our attention.

COMIT

This is, strictly speaking, not a general list-processing language, but a string-processing language. The method of addressing and manipulating data is perhaps the most novel and powerful feature of this language.

It was originally designed for natural language data processing and research, but has been found to be useful for a much wider variety of problems. The 709-90 version is available through SHARE; the 704 version was not completely finished. It is not available on any other machine. In the present implementation, the COMIT source program is first compiled into an intermediate language and packed into a table, and this table is then interpretively executed. The system has very extensive debugging aids.

II. THE SHAPE OF DATA

Symbols Inherent in the Language and Symbols Available for Use as Program Names and Data

FLPL

The fixed vocabulary of Fortran is rather small. The FLPL functions introduce about fifty new names. These names are meant to be mnemonic but certain pairs unfortunately lack visual distinctiveness, and are easily confused, e.g. XBSPSHF, XBTSCHF, XSTORTF, XSTORDF. In naming subroutines, one must of course avoid using any of the names of these functions. The rules for naming functions and variables which have fixed point values must also be adhered to, since the symbols that FLPL manipulates are treated as Fortran integers. Otherwise the user is free to choose his own names for use in programs. Arbitrary data may be handled, since due to context, there is no chance of collision with the fixed vocabulary of the language.

IPL

The IPL symbols in the J, H, and W regions, i.e., IPL symbols beginning with these letters, have special meanings within the system, that is, constitute the fixed vocabulary of the language, and may not be used to represent data or to name arbitrary routines. However, the programmer may invent and use any other symbols which adhere to the IPL symbol format. Data other than IPL symbols can be handled; but only in the restricted "data term" format.

LISP

The basic vocabulary of LISP consists of some half dozen words or "atomic symbols". The fixed vocabulary of the current LISP systems also includes some 40 or 50 names of commonly used functions which come with the system "package". Apart from these names the user is completely free to invent his own data symbols and function names (except that MBLISP restricts atomic symbols to 36 characters in length). Data symbols and function names are kept apart by context.

COMIT

Except for the "punctuation" symbols -, /, //, =, etc., the thirty or so "routing instructions" are the only symbols in the fixed vocabulary of COMIT. Names for program locations, or "rule" names, may be arbitrarily made up (except for a very few conventions such as not starting with a period, and being 12 characters or less in length.) There are also very few restrictions on data. Furthermore, an entire second font of letters is available for use by the program. These are represented by "double characters" *A, *B, ..., *Z. Symbols with double characters may be inserted into the data by the program only and cannot be read in as data. Thus when, for instance, a marker is to be inserted into the data at some point by the program, in order to insure that the marker will be distinct from any possible symbol in the data, one or two of these "double characters" may be used in the marker symbol.

DATA STRUCTURES

The kinds and shapes of data elements and data structures that can be manipulated vary greatly from language to language.

FLPL

Data elements in Fortran can be numeric (fixed or floating point), or arbitrary alphanumeric strings, but appropriate FORMAT statements must be provided by the programmer and adhered to by the data to be read in or out. For symbolic data it is most convenient to handle strings of six or fewer characters, since the language requires the user to manipulate contents of actual machine cells. Longer strings can of course be handled by creating list structures for them. List structures to be read in cannot be read in as list structures; the data must be input using ordinary Fortran facilities (READ and FORMAT statements) and then explicitly assembled into the desired list structures. Similarly, data from a list structure for output must be assembled by the program and written out using ordinary Fortran facilities. (See section on input-output). Within the machine, however, list structures of arbitrary shape and complexity may be generated and manipulated. The user has complete freedom to read and write in the decrement and address parts of cells.

IPL-V

The basic unit of data, which has no internal structure as far as the language is concerned, is the "IPL symbol", consisting of a letter (or the digit 9) followed by a number, e.g. A25, 93. Individual cells, lists, and list structures containing IPL symbols, are the normal kinds of data structures handled by IPL. A list is represented on paper by a vertical column of the symbols in the list, together with a symbol representing the "name" of the list, and a "termination symbol" 0 at the end of the list. When symbols on a list are names of other lists, we have a list structure. Simple "trees", (no re-entries, no loops) are the only list structures that are safe to handle in IPL. It is possible to construct and manipulate other kinds of structures in IPL, but the system assumes that only trees are being handled.

Numeric and arbitrary alphanumeric data can also be handled, but must be referred to indirectly by means of regular IPL symbols. The IPL symbols serve as names of the "data terms", which may be numbers (fixed or floating point) or strings of five or fewer alphanumeric characters. Input format for data terms is also columnar, as for lists, with one data term to a line.

Besides the normal output format for lists and data terms, that is, vertical lists, there is a horizontal "print line" format available. The user may "compose" the print line from symbols and data terms and print out the line. Thus, although the input format for data is very restricted, the output format is potentially completely general. The corresponding input capability, for reading in horizontal lines of data, exists in at least one implementation of IPL-V, but is not generally available.

LISP

The basic unit of data in LISP is the "atomic symbol"--an arbitrary string of alphanumeric characters (in the current MBLISP there is a length limit of 36 characters). Although the atomic symbol was the smallest piece of data distinguishable from another atomic symbol in LISP I, MBLISP has and LISP 1.5 will have the ability to break up an atomic symbol into a list of its constituent characters, as well as reassembling a list of characters into an atomic symbol. Thus completely arbitrary data can be handled in these versions of LISP. Floating point numbers were also permitted as atomic symbols in LISP I, and both floating and fixed point numbers will be permitted in LISP 1.5. The data structures that LISP handles are essentially simple trees. Depending on the history of a particular program, trees with common subtrees may in actual fact share those substructures in the machine, and hence would be called re-entrant list structures, but the user has no explicit control over this situation, and need think only in terms of simple trees. A LISP list structure is represented on paper horizontally, with parentheses grouping the subtrees. Input-output format of data is also in this form. But since blanks and card boundaries are ignored by the input routine, input data may be arranged so as to be very convenient to read. Output format is normally out of the control of the user. The system prints out list structures in horizontal unspaced and unindented lines. The capability of reading in an arbitrary record of data and assembling the characters into a list is available with MBLISP.

COMIT

Data in COMIT are of three types or levels: 1) symbols, which are arbitrary strings of characters; 2) subscripts, arbitrary strings up to twelve characters in length, which can be associated as a kind of "tag", with particular occurrences of symbols; and 3) values, which are arbitrary strings, associated, in turn, with each occurrence of a subscript. A special kind of subscript is the numerical subscript, whose value is an integer. This is the only place in COMIT where numerical data is handled. An occurrence of a symbol may have any number of subscripts associated with it, and each subscript may have up to 36 values. "New" subscripts or values, which are not mentioned in the program, cannot be read in as data during the running of the program. That is, the range of different subscripts and of their values (except for numerical subscripts) that a program can handle is set once and for all when the program is compiled, and other subscripts and values will not be recognized. A further characteristic of these three levels of data in COMIT is that data on one level can never become data on another level. Values can never be made into subscripts by a program nor subscripts into symbols, and reversely. (However, such a feature may become available in a future version of COMIT.)

Data structures in COMIT are restricted to linear strings of constitutents. A constitutent is a symbol occurrence together with subscripts and values, if any.

Constituents and strings of constituents are represented horizontally on paper. The symbol SYM, for instance, with numerical subscript 10 and another subscript SUB with values T1 and T2, would be represented as SYM/.10, SUB T1 T2. Strings of such constituents are written horizontally with + signs joining the constituents, e.g. A/N + B + C/G N, H L.

Data may be read in and written out in this form (format S, or "workspace notation"), complete with + signs and slashes. Data in arbitrary format may also be handled: The format A instructions read in data character by character and string them out as a string of one-character symbols. Format A instructions also can write out the symbol parts of specified constituents, with no spaces or + signs between them. Instructions which expand symbols into a string of their constituent characters, and which compress specified consecutive symbols into one symbol, are also available. Thus COMIT is able to handle completely arbitrary input data, and write out data in any desired format.

General Remarks on Heirarchy in Data Structures.

It is frequently convenient to organize data not as simple lists but as lists of lists, to arbitrary depth, and to be able to refer to the next element on a list at a certain level without having to pass through any intervening sublevels. Thus it may be useful to refer directly to the next sentence without having to count out the words of the current sentence, or to find the next right parenthesis at the current level without having to count out all intervening left and right parentheses. Such trees are naturally handled by FLPL, IPL, and LISP, (although there is no built-in facility in FLPL for the input and output of trees.). COMIT, however, stores only linear strings of symbols. To store trees in COMIT, special schemes have to be devised to represent them linearly. One natural way is to introduce special constituents as markers, like the parentheses in algebraic expressions. It would be convenient, though not logically necessary, to attach numerical subscripts to these markers to indicate their respective depths or scopes.

III. THE SHAPE OF ROUTINES

The "Sequential Units" of the Languages.

Programs in most procedural languages consist of a top-level routine and a set of subroutines. Each routine in turn consists of a sequence of units which all have the same structure, or at least have structures from a limited set of structures. In looking for and discussing these "sequential units" in the different languages, we may use this criterion as a rough guide: They are those recurrent structures in a program, generally of a limited number of types, which are interpreted, or executed, sequentially in the order in which they are written, except where jump instructions break the sequence.

It is interesting to consider units that have this property because, very generally speaking, a language with relatively powerful "sequential units" requires less breaking down of a procedure into sequential steps by the user. Thus the complexity of a process may be reflected more in the depth rather than in the length of its description.

One of the standard ways of making a "sequential unit" powerful is to enable it to execute (i.e., call and return from) an entire subroutine, as in IPL. The subroutine must still be coded, of course, but considerable flexibility can be achieved in a language which allows this. Some brevity of expression can be gained by allowing the "sequential unit" also to specify the transmission of variables to and from the subroutine, as in a functional notation such as that of Fortran and LISP. Nesting of functions further adds to the power of a sequential unit with a functional notation. Yet another extension of this device is provided in the LISP conditional expression notation (where we shall consider the pair, predicate and function, as the unit which satisfies the criterion above) which, in addition, expresses a condition (which itself may involve functions to arbitrary depth) to be evaluated to determine whether or not the associated function is to be evaluated. Here we may note that recursion, a very powerful means for constructing functions in LISP, is not a property of the "sequential unit", but of the next higher unit, that on the "subroutine" level.

Completely novel devices for increasing the power of "sequential units" are used in COMIT. The highest level of "sequential units" have: 1) The ability to specify a complicated linear pattern of symbols, which actually combines the operation of a predicate evaluation (determining whether the given pattern exists in the data) and that of data addressing (assigning <u>temporary numbers</u> to the <u>constituents</u> of the pattern); and 2) the ability to specify an arbitrary rearrangement of the pattern including insertions and deletions. This power, however, is partially counter-balanced by the absence of the powerful device which the other languages have, namely the ability to execute entire subroutines from within one such unit. Thus while each COMIT rule may be quite "deep", the total COMIT program is comparatively "shallow" for its length.

We now consider these "sequential units" in greater detail in each of the languages.

<u>FLPL</u>

The "sequential units" are the Fortran <u>statements</u> which are written one after the other in a column, and are executed in that order. Probably the most commonly used types are the "assignment statement" and the "IF" statement. An assignment statement can specify the evaluation of functions, nested to arbitrary depth, on given arguments, and the assignment of the value of the top level function to a specified variable. With the FLPL functions, such a statement might be used to specify a search down a list for a given piece of data, insertion or deletion or replacement of data, retrieval of data, etc., or combinations of these different types of operations. By invoking names of subroutines, rather than just the FLPL functions, subroutines of any depth may be executed within such a statement.

The "IF" statement, for the purposes of symbol manipulation, may be said to send control to one of two specified points in the routine depending on whether or not two variables currently have the same value. These variables can also be specified in terms of functions, nested to any depth. Hence, although the information finally extracted from the evaluation of these functions is very small (one bit, represented by the two-way branch), it may represent the value of a very complex predicate.

<u>IPL-V</u>

The "sequential units" are the <u>IPL instructions</u>. An IPL instruction consists of only one symbol, together with a prefix indicating how the symbol is to be interpreted e.g., as the name of a routine to be executed, as a piece of data to be copied, as the location of data to be copied, as an address to which control is to be conditionally transferred, etc. Although the one symbol may represent a subroutine or a list structure of great complexity, the programmer must still deal with only one symbol at a time. To apply an operation on, say, two arguments and to store the result of the operation requires in general four separate sequential instructions in IPL-V (but would need only one "sequential unit" in FLPL). If furthermore each argument must be computed as functions of other arguments, correspondingly more IPL instructions would be needed, whereas in FLPL the subfunctions may, in most cases, simply be nested within the one assignment statement.

LISP

The predicates in a function definition are evaluated one after the other in the order in which they are written, and the subfunction associated with the first predicate which has the value "true" is evaluated. Thus we may consider the predicate-and-function pair in a function definition as satisfying the criterion for a "sequential unit". As was mentioned in the introduction to this section, this unit is even more powerful than the assignment statement in Fortran, since it further contains a branching facility which decides, on the value of a predicate of arbitrary complexity, whether to evaluate the associated function or to proceed to the next "sequential unit".

COMIT

In COMIT we have two levels of "sequential units". One level is represented by the COMIT "rule", which is executed one after the other in a program. Within the rule, however, are other "sequential units", the "left-half", the "right-half" (which is optional), the individual "routing" instructions (which are also optional), and the "go-to", which are also executed one after the other in the order written. As mentioned in the introduction to this section, the left-half and right-half operations are quite powerful. Also, the different kinds of routing instructions express in very compact form some useful and essential operations that cannot be described in the left-half-right-half notation, such as input-output, expansion of symbols into their constituent characters, etc.

Physiognomy of the Code.

We are interested here in the physical format of programs and instructions written in the languages and their "readability", and, in particular, whether the physical appearance of the program reflects the flow of control and the hierarchy of routines, and distinguishes among the functionally different types of objects (e.g., operations, data, branch commands, etc.).

FLPL

A Fortran routine consists of a list of "assignment statements" and "control statements" (and input-output statements). The statements that may be approached by jump are clearly labeled in the margin with statement numbers. The jump statements themselves, however, are not distinguished by any such format feature, but merge in with the rest of the statements.

Within the assignment statement there is no difference between the shape of a function computed for its value and one computed for its effect on a list structure, nor are names of subroutines distinguishable from the names of the basic FLPL functions except simply by the reader's knowing their names. The dependence between routines is similarly not immediately obvious, since calls to subroutines, that is, mentions of subroutine names, are buried within the general mass of other computation in the calling routine.

Lines of explanatory comments may be inserted among the lines of code.

IPL-V

An IPL routine consists of a vertical column of instructions, each composed of a two-digit prefix and an IPL symbol. Instructions approachable by jump are labeled to the left of the column (in the NAME section), and unconditional transfers protrude to the right (LINK section) of the column. The uniform format of the instructions, however, does not provide simple visual cues to the reader for distinguishing between functionally very different operations, e.g. pushing down a cell, transferring a symbol from one cell to another, executing a subroutine, or a conditional branch. The logic at conditional branch points is especially obscure to a reader: He must carefully trace through several steps of logic in order to determine the relation between the directions of the branch and the condition which set the test cell H5 before the branch. (H5 is a standard cell in which may be placed one of two symbols and to which a conditional branch instruction always appeals in order to make the appropriate jump.) Since calls on subroutines are buried in the same format as ordinary housekeeping or logic instructions, the dependence of a routine on other routines is also not immediately evident.

Short comments may be written in parallel with the column of instructions. This format allows a clearer visual separation of comments from program, while preserving the advantage of a step-by-step parallel running commentary, than do the formats of comments in Fortran or Comit, which are interspersed among the instructions themselves.

LISP

Conditional branching and the dependence between functions are both more clearly brought out in the LISP notation, due to the distinctiveness of the arrow in the conditional expression and to the general absence of housekeeping matter that tends to clutter up the codes in the other languages.

Due to the free format, cascaded conditions can be written one under the other so that the arrows fall under each other, presenting a picture very suggestive of the logical structure of the function.

Comments may not be interspersed within or between LISP functions in the actual running deck, but may be placed after the STOP card.

COMIT

A COMIT program consists of a list of "rules", each of which has a name beginning in column one, and continues to the right of the name in a free format. The left and right halves of a COMIT rule provide convenient visualizations of the shape of the data before and after the operations to be performed. The following inconvenience is, however, frequently met: All the symbols (constituents) that the COMIT programmer writes in his code are treated with equal weight, whereas one of them may refer to a symbol used as a marker or punctuation, another may refer to actual data, and still another may be a dummy constituent used only for carrying a logical or numerical subscript, etc. These differences in function are not brought out in the shape or format of the symbols, and increases the difficulty of reading and understanding a program.

A more serious fault with the notation is this: The flow of control in a COMIT program is not generally evident from the shape of the code. It is not even evident from an inspection of the individual rules, since the reader has no immediate way of telling whether the left half of a given rule is meant to be always satisfied, or whether it is being used as a test for a conditional branch. To understand the function of a left half depends intimately on a knowledge of the expected shape of the intermediate data on which it is to operate. Similarly, other control operations such as dispatcher entries and the dollar sign in the go-to, are imbedded in the general text-like code, and their effect cannot be readily appreciated by the reader of a program.

IV. BASIC OPERATIONS

Basic Operations Provided by the System

List Processing Operations

All the languages, of course, provide list processing operations. FLPL and IPL-V provide large batteries of operations for inserting, deleting, and locating given symbols on lists, and for erasing, creating, and concatenating lists. These operations can, of course, all be performed in LISP, with functions that can be easily built up from the six elementary LISP functions. Some of the commoner list processing functions, such as appending and deleting, are provided in the "basic functions" package. In COMIT all these operations can be expressed within the framework of the left-half-right-half notation.

Arithmetic Capability.

With FLPL, all the arithmetic capabilities of Fortran are, of course, available. This includes integer and floating point arithmetic, the arithmetic comparisons, and some of the commonest functions such as exponentiation, square root and the trigonometric functions. IPL also provides the full range of arithmetic operations (but none of the other functions) on integers and floating point numbers. The arguments of arithmetic operations may be of mixed types (e.g. a floating point number may be added to a fixed point number). LISP I provided addition, multiplication, and exponentiation of (positive and negative) floating point numbers only, and no "less than" or "greater than" comparison between numbers. LISP 1.5 has both floating and fixed point numbers. MBLISP has no arithmetic capability, and whatever arithmetic is needed must be programmed as symbol-manipulation. In COMIT, only integer arithmetic is allowed. The numbers are manipulated as numerical subscripts on constituents. Addition, subtraction, multiplication, and a kind of division, as well as the comparison for equality and the inequalities, are available.

Manipulation of Internal Structure of Symbols.

Data in FLPL can be examined character by character and even bit by bit by using the masking and shifting operations provided.

The internal structure of IPL _symbols_ cannot be computed upon, nor can that of alphanumeric _data terms_. Once they are in the machine they can be treated only at atomic units. Only the relation of equality can hold between two _symbols_ or between two alphanumeric _data terms_.

The internal structure of atomic symbols is accessible to computation in MBLISP and LISP 1.5 via the expansion and compression functions.

In COMIT, _symbols_ can be _expanded_ character by character, each character forming a new _symbol_; and any sequence of _symbols_ can be combined, or _compressed_ into a single _symbol_. Thus the character structure of _symbols_ is completely accessible to the programmer. _Subscripts_ and _values_, however, cannot be so treated. They are fixed once and for all at the beginning of a run, and their internal structure is not accessible to the programmer.

Tagging of Symbols on List.

Sometimes it is convenient to be able to attach to a symbol extra information _about_ the symbol, and not to have to do this by adding additional symbols on the same list, or by means of a sublist. Means for doing this would then be, strictly speaking, not "list-processing" facilities.

FLPL provides such a facility by making available the six extra bits in the 704 (7090) word which are not used by address and decrement, the sign-prefix and the tag. There are functions available to retrieve and alter these portions of a given word, and functions to search linearly down a list for the first word which contains a given bit-configuration in the sign-prefix or tag, etc.

In IPL, these extra bits are not available for the user, but the comparable function can be easily performed by using _description lists._ While these are sublists in the ordinary sense, and can be treated by the regular list-processing operations, there are available special _description list processes_ which treat a _description list_ as a list of pairs--pairs of "attribute" and "value". Given a _symbol_ and an _attribute_, the value can be directly retrieved or altered by the _description list processes_, and the user need not be aware that the _description list_ exists as a regular list. Note that the _description list_ belongs to a _symbol_, rather than to a symbol occurrence or to a position in a list, whereas in FLPL, the flagging bits belong to a cell.

In LISP no such facility is provided in the system, but has to be programmed using sublists.

COMIT provides extensive tagging facilities on _symbols_. The tagging is of an occurrence of a _symbol_ rather than the _symbol_ itself. A _symbol_ occurrence may be tagged with an integer and with any number of _logical subscripts_, each of which may have up to 36 _values_. These items may be retrieved, altered or moved from one _symbol_ to another very simply. There is also a sophisticated way for computing with _logical subscripts_ and _values_, called _dispatcher logic_. As was previously mentioned, the total set of available subscripts and values is fixed for each program, and cannot be modified during a run.

V. BRANCHING AND OTHER CONTROL OPERATIONS

Branching.

FLPL

Branching is most commonly done with the IF statement. The three-way branch which it offers is usually, in symbol-manipulation situations, reduced to a two-way branch, since two symbols are either equal to or not equal to, and usually not greater or less than, each other. A complicated nest of expressions may occur in the IF statement, so that the branch can be made to depend on a complicated condition. Other branching methods are available, such as computed and present switches, etc.

IPL-5

All branching in IPL is binary and is contingent on the current state (+ or -) of a particular cell, the *test cell* H5. The existence of only one such *test cell* is sometimes inconvenient, requiring considerable housekeeping.

LISP

The only method of branching is through the *conditional expression*, which can actually look like a large switch, when the *predicates* are cascaded one after the other. Furthermore the *predicates* themselves can be of any complexity, either expressed fully in the *conditional expression* or defined explicitly elsewhere and only referred to by name.

COMIT has perhaps the most extensive and the most sophisticated and idiosyncratic set of branching abilities of all the languages, and someone who is very familiar with them can make very elegant use of them. The most common branching mechanism is the success or failure of *left half* match. This branching is binary. The condition tested for by this branching may be very complicated logically, although simple to express as a string configuration.

A similar branching occurs when an input instruction in the *routing section* fails (reads an EOF), in which case control also falls to the next *rule*.

Another device is the *rule* with multiple *subrules* where the choice of the particular subrule to be executed can be set previously in the program by a "dispatcher entry". If the switch is not already set before the multiple subrule *rule* is reached, or if the switch is set so as to still leave more than one choice among the *subrules*, the choice is made by a built-in pseudo-random number generator.

Finally, a very useful indirect transfer of control device is the "*dollar sign*" in the "*Go-To*". The programmer can find a *constituent* and send control to the *rule* whose *name* appears as a *subscript* on that *constituent* (without of course knowing in advance which *rule* it is). Thus he need only know what the *symbol* part of the *constituent* is, or know at least some way of locating that *constituent* (say the first *constituent* on a certain *shelf*). The *constituent* may have been put there by any other part of the program. Subroutining frequently uses this device. Before transferring to the subroutine, a dummy *symbol* having the return address as a *subscript* is put onto some standard *shelf*. At the end of the subroutine, the subroutine, as standard procedure, takes the first *constituent* off that *shelf* and transfers control to the *rule* whose name appears as *subscript* on that *constituent*.

A final remark concerning the use of the left-half match as a branching method. Frequently a left-half expresses more than one condition, whose satisfaction or failure is of interest. However, the left-half match discriminates only between a complete satisfaction of all the conditions and a failure of any one of them, that is, when the match fails, any one of the conditions might have caused it, and further left-half searches, in the succeeding rules, are needed to find out exactly which condition (or even conditions) was the cause. This reflects the difference between the great power of a left-half search and the extremely limited binary branching facility with which it is associated.

Multiple Branching and the Addressability of Program by Data.

We are interested here in whether it is possible for a program address or the name of a routine to be stored as data and to be used for transfer of control purposes. This is useful for constructing a many-way branch depending directly on a given piece of data, and is much more efficient and convenient than writing a tree of tests which test the given piece of data for all the possibilities.

A multi-way branch may be accomplished in three kinds of ways: 1) The most primitive is to use a tree of binary tests to determine exactly what the piece of data is, among a list of predetermined possibilities, and to transfer control according to the result. This is, of course, possible in all of the languages. 2) Some languages provide facilities for setting up a multi-way switch, which essentially compresses the test tree into something in a more convenient linear form. The user must still, at the point of the switch, provide the list of possible data items and the list of corresponding transfer addresses. Thus the list rule in COMIT may be used as such a device where the left halves of the subrules of the list rule provide the possibilities and the corresponding go-to's provide the addresses. IPL does not provide such a device within the system, but it may easily be coded using description lists. In LISP such a device exists naturally in the conditional expression, when used in a cascaded fashion (nesting always to the right) rather than in a fully nested fashion. The free coding format of LISP allows such a switch to be written in a particularly clear way, that is, with the arrows lined up underneath each other.

In all these schemes a left-over case can always be provided for, that is, the case where the input data does not match any of the anticipated cases: In the COMIT list rule, by the failure of any of the left halves to match, resulting in a skip to the subsequent rule; in the description list switch by a special provision when the given symbol is not found as an attribute; and in LISP by the use of the "T" as the final condition.

Fortran provides a multiway switch in the form of the computed and assigned go-to's. The input data to the switch is, however, restricted to integers and cannot be arbitrary symbolic data.

3) In all these schemes, however, the programmer is required to explicitly set up the switch, with its list of anticipated inputs (except for Fortran where this list is always a set of consecutive integers) and list of corresponding transfer addresses. A more convenient method would be the use of a statement such as "execute x" or "go to x" where x is a variable which can be assigned as value to the name (address) of some instruction or subroutine in the program. This can be done in LISP, COMIT, and IPL but not in FLPL.

In IPL this can be done by using the prefix codes 01 or 02. Thus 01A5 means execute the routine names in cell A5, and 02A5 means execute the routine named in the cell named in A5. In either case the actual name of the routine need not be explicitly stated at this point. Another way to do this is to use J1 "execute the routine named in H0". Notice that an indirect go-to is not possible, but an indirect execute is.

In LISP an indirect execute can be done by using the APPLY function. The first argument of the APPLY function is evaluated to get the name of a function (which is in a data structure as if it were data), and the APPLY function evaluates it with the argument provided by the second argument.

COMIT provides an indirect go-to in the ∅ in the Go-To facility. The address (name) of the rule to which control is to be transferred is a subscript on a constituent located by the rule from which control is to be transferred. Note that while the program address is storable as data, it is stored as a subscript, which can never be interpreted, by the same program, as a symbol or a value.

To summarize, we have:

	Binary Test Tree	Multiway Switch	Indirect Go-To	Indirect Execute
FLPL	IF statements	many way Go-To's	no	no
LISP	fully nested conditional expressions	cascaded to the right conditional expressions	no	APPLY function
IPL	ordinary branchings on H5	programmable	no	Prefixes 01 or 02 or J1
COMIT	ordinary left half test	list rule or simple sequence of ordinary rules	∅ in go-to	no

VI. SUBROUTINING AND RECURSION

Subroutining.

At least three factors are associated with the ease of writing and using subroutines: 1) the housekeeping necessary to transmit variables to and from subroutines; 2) the method of transferring control to and from the subroutines; and 3) the availability of local names, and other means of protecting the private property of individual subroutines.

FLPL

There is essentially no housekeeping associated with transmission of arguments and the storage of results. The arguments are simply mentioned by name in the right order in the calling statement.

The user, also, has nothing to do about transferring control to and from subroutines. All such housekeeping is automatic.

All names of variables and statement locations in subroutines are local except for variables declared "COMMON" at the beginning of the routine so that the user need not worry about collision of names while writing routines.

IPL

Transfer of control to and from subroutines is fully automatic but the conventions regarding the transmission of variables and regarding the use of temporary storage demand a considerable amount of careful housekeeping in both the using and the writing of subroutines: Inputs to subroutines have to be placed in the cell HO in the proper order before the subroutine is called, and the outputs left in HO by subroutine have to be saved or otherwise immediately dealt with upon return from the subroutine. Within the subroutine itself, upon entry the inputs must be saved, usually in the working cells WO through W9; and just before termination, the W's must be restored to their original state and the outputs, if any, have to be put into HO in the correct order.

Actually, these are only convention, adhered to by most IPL programmers, and not demanded, although greatly facilitated, by the structure of the language. They need not be observed, but only at great expense in flexibility and independence of the routines.

In IPL, local symbols are available for labeling private program and data addresses, and even for naming private subroutines.

LISP

Transfer of variables and transfer of control to and from subroutines are fully automatic as in FLPL. In fact, all variables appear as arguments. All variable names are local, while names of functions are always "common", in the Fortran sense. The lack of local subroutine (function) names is rarely an inconvenience.

COMIT

There is no transmission of data to and from subroutines, since all data is accessible to all parts of the entire program, although a programmer may wish to establish and observe certain conventions in this regard, if he finds it more convenient to do so. However, some housekeeping is necessary to transfer control correctly into and out of subroutines, and the absence of "local" facilities is frequently inconvenient.

There are several ways to approach and leave subroutines in COMIT. Using the dispatcher entry requires that the names of all the return points from the subroutine be known, and a special multi-way branch rule be set up with subrules corresponding to the return points. Another way is by shelving a constituent having as subscript the return point name, before going into the subroutine. The subroutine terminates by finding this constituent and exits indirectly by a $ in the GO-TO. This does not have the drawback of the dispatcher method and therefore is more flexible, and also permits recursive subroutining without difficulty (the return addresses are merely pushed down on the shelf) but has the disadvantage that housekeeping has to be done at three points; before the jump to the subroutine, at the exit to the subroutine, and upon return from a subroutine to delete the return-address-bearing constituent from the workspace.

The lack of local names is frequently an inconvenience, especially when a large program is being written and debugged in several segments. Rule names may inadvertently be repeated. Also, local shelves which would be useful for temporary storage are not available. In the present COMIT system the programmer in writing a subroutine must know which shelves are not in use by other portions of the program, unless he adopts a time consuming "push-down" convention.

Recursion

A routine is recursive if it calls upon itself as a subroutine. It is frequently useful or natural to write a subroutine as a recursive subroutine, for instance in processing a list structure whose elements are considered all of the same order, or in generating permutations. The particular difficulties presented by a recursive routine over and above those of a routine which simply loops back upon itself are that for each call upon itself, the routine must remember to return after the completion of the subroutine and it must keep safe the temporary storage registers which it is using, so that the subroutine does not overwrite them.

In FLPL no special facilities are provided for doing these things, although temporary storage can easily be arranged into push-down lists and the returns can be counted out either by using a push-down list or by numerical counting.

In IPL recursive subroutining presents no more difficulty than ordinary subroutining as long as the user observes certain conventions regarding the use of public working storage to assure the safety of temporary storage during any kind of subroutining. If he uses private storage he must remember to observe the same push-down and pop-up conventions.

In LISP, recursion is the normal way to write loops, and iteration cannot be done in any other way (except by using the program feature). This is, however, frequently an encumbrance since it is not always easy to see how to write a loop as a recursive function. However, as in other areas in the use of LISP, no housekeeping on the part of the programmer is necessary nor even possible. Such things as pushing down and saving results are taken care of automatically.

Since COMIT is essentially a one-level language and not even subroutining is particularly simple to do, recursion is not particularly natural to do in it either, although due to the natural way in which the <u>shelves</u> can be used as push-down stores, a recursive subroutine is not much more difficult to manage than an ordinary subroutine. Both data and return addresses can be pushed onto <u>shelves</u> and taken off in the opposite order.

VII. INPUT-OUTPUT

<u>Loading of Programs and Data, Output of Results</u>.

<u>FLPL</u>

The source and/or binary programs are loaded automatically in ways differing from installation to installation. Data is not loaded by the system but must be read in by the FLPL program itself. Since there are no FLPL functions to read in list structures, input of list structures has to be programmed using the ordinary Fortran input facilities in conjunction with FLPL list-generation functions. The card format for lists is thus also up to the programmer. Similarly, output routines must be written.

The program may read in data at any time during the run, but it cannot load more program and transfer control to it. Programs which take up more than one memory load may be run as "chain jobs" within some monitor systems.

IPL

During "initial loading", the system loads both routines and data. The program can, during a run, read in further data, as well as additional routines to be executed.

There are primitive operations which print out IPL symbols and data, lists and list structures (J150-153). Elaborate formatting may be done, with more programming effort, by using the line-printing primitives (J154-161). Input and output tape units are specified by data terms in fixed system cells, viz. W16 and W17.

LISP

The system initially loads both program and data. Further data and programs may be read in by the program during a run. Output may be either explicitly programmed, by using the PRINT and associated functions, or may be left to the system, which automatically prints out the value of the highest level function evaluated. Output is always on a fixed system output tape.

Before punching up a program, a transliteration and a small amount of translation must be done by the programmer to convert his functions, normally written in the m-expression notation, to s-expressions acceptable by the LISP interpreter. Parentheses must be counted out carefully.

COMIT

Program is loaded automatically from a fixed tape unit by the system. Data must be read in by the program, the choice of input unit being up to the programmer. Additional programs cannot be loaded in the middle of a run. The programmer has complete control over the output of data, including formatting and choice of tape unit.

Composition of the Running Deck.

FLPL

The actual deck submitted for a run consists at least of the source and/or already compiled routines plus the data. Depending on what routines are available on the library tape that a particular installation maintains, the program deck may also contain the FLPL binary deck and other library subroutines used. A number of control cards also appear in the deck, depending on the conventions at the installation.

IPL

The deck submitted for running must be headed by region reservation cards, a housekeeping detail which could logically be left to the system, but which, in the interest of loading efficiency, has been relegated to the user. These cards instruct the system as to what regional IPL symbols to expect in the routines and data that follow. Any "unexpected" symbol in the following deck not accounted for in the reservation cards will result in a bad assembly. Blocks of routines and data, each block initiated by a header card, constitute the body of the deck. A transfer card ("final type 5 card") terminates loading and directs the interpreter to the beginning of a specified routine.

LISP

The deck begins with a small binary deck which calls the system from tape. The body of the deck contains S-expressions representing the definitions and arguments of functions. Each job is headed by a special header card and terminated by a "STOP" card. A series of jobs is terminated by a "FIN" card.

COMIT

The running deck consists of the program itself, with the first rule to be executed on the first card, and the logically last rule on the last card. A control card is placed before and another after the program. After this may be placed the data.

Temporary Storage of Data and Program on Tape.

FLPL

By using the regular tape operations available in Fortran, temporary data can be stored on, and read back from, tape. Depending on the installation there will also be methods for saving all of memory at any point during a run for continuation of the program at a future time.

IPL

Facilities for temporarily storing data as well as routines on slow or fast back-up storage (e.g., drum, tapes) are described in the manual, but are not actually available on all of the systems presently in use. The normal input and output facilities may not be used for temporary storage on tape because of format discrepancies between normal input and output, and the lack of tape manipulation operations. The facility for saving memory for restart (J166) does, however, seem to be generally available.

LISP

There are no facilities for temporary storage of data and program on tape. By using the "SET" header card, core may be saved on tape at the end of a run, to be used in a later job.

COMIT

The normal input and output facilities of Format S and Format A can be used for temporary storage of data on tape. Format B, however, is particularly fast and suitable for this purpose since it reads and writes in binary. The user, however, is still responsible for such tape operations as writing and testing for end-of-file, rewinding tape, etc. By preceding the deck with a special COMSET card the user may cause core to be saved on tape at the end of a run, for re-run from the beginning at a future time.

On-Line Input and Output.

FLPL

The Fortran manuals describe statements which read information in from console switches and which print on the on-line printer, but at most installations the use of these facilities is discouraged or prohibited.

IPL

The manual mentions a primitive, J141, which reads in one IPL symbol from the console, but this is not available in most actual versions of the system. No on-line output facilities are generally available.

LISP

The standard system has no facilities for reading console switches. The LISP-flexo system, as described in the LISP I (704) manual, however, allows input-output of data via the on-line flexowriter if there is one.

COMIT

Both sense-switches and on-line flexo-writer may be read by a COMIT program. A COMIT program may also print out on the one-line printer.

VIII. DEBUGGING AIDS

Debugging.

We are concerned here with diagnostic features of the various systems, program tracing and dumping capabilities, and program patching facilities.

FLPL

The Fortran compiler checks for errors of format, spelling, punctuation, logic (e.g., unreachable points in the program), etc., in the source code and prints out explicit error comments. There is no special facility for tracing of the program during execution; any tracing must be programmed. However, depending on the installation, various kinds of memory dumps are available and may be quite easily requested at any point in the program or may be triggered by various error conditions. For FLPL specifically there is a list dump function, XDUMPF, which prints out a specified list or list structure, the absolute address and contents of each cell being represented in octal notation.

Patching of programs is done simply by replacing, deleting, and inserting statements in the source program. It is rarely justified to patch in the compiled binary deck.

IPL

The assembler, which loads the running deck into the computer, does not check for errors in format (e.g., symbols punched in wrong columns) or logical errors (e.g. unreachable points in routines).

Debugging of runs is aided by _tracing_, _snapshots_, and the _post-mortem dump_. An elaborate tracing facility is provided whereby the programmer may request a printout of the contents of a number of key cells of the system (H0, H5, H4, etc.) for each instruction executed. He may request that only the instructions in a certain routine be traced (and not those in its subroutines), or he may request a _full trace_ of a routine (in which case every subroutine which the routine calls is traced). In the hands of an experienced IPL programmer, the first type of tracing, the _selective trace_, can be a very powerful debugging aid. The traces can be turned on and off by a signal (Q = 3 or 4) written in the routine to be traced, or by execution of an instruction (J 47-9) in some other routine external to the routine being traced or by manual intervention through console switches. Breakpoints may also be planted at crucial points in routines by the programmer to trigger _snapshot_ routines that he himself provides. _Snapshot_ dumps and tracing can be controlled in very elaborate ways by a combination of internal (programmed) and external (console switches) means. At the end of a run the system always prints either a normal termination message or an error comment, followed by a post mortem dump of the final contents of all of the system cells (the H's and W's).

Correction of errors involves repunching of the offending instructions or routines. If routines are kept short and flexible, check-out and patching and other modifications are simplified. Experienced IPL programmers often consider it worthwhile to set up even two-instruction sequences as subroutines if it is convenient to think of them as operational units.

LISP

Essentially no checking of format, spelling, etc., is or can be done on input functions, since there is no format to speak of, and every spelling can represent a legal atomic symbol. Even missing subroutines can not theoretically be checked for during loading since an entire function might be generated during the execution of the program. About the only real input error that can be and is checked for is that of unbalanced left and right parentheses.

For debugging programs, the user has at his disposal two tracing functions. One, called TRACE, prints out the arguments and value of EVAL (the principal function in the interpreter) each time it is used by the interpreter (in IPL terminology this is a trace of each interpretation cycle). The other, TRACKLIST, prints out the arguments and value of any specified function each time it is evaluated. The interpreter also provides error comments whenever a computation cannot continue due to a logical error. However, these comments, as well as the print-outs from the trace functions tend to be rather cryptic to someone not intimately acquainted with the operation of the interpreter.

COMIT

The compiler checks for errors in format, syntax, and logic and prints out very explicit error diagnostics. A bad compilation does not halt a run, however. The system compiles some kind of program and proceeds to run it as far as possible, in order to discover further errors, if any.

COMIT does not provide special debugging aids within the language. Tracing and dumping must be programmed into the program being checked out. On-line debugging aids, however, are available. By manipulating sense switches, a programmer at the console may follow the progress of a program on the on-line printer, rule by rule as they are executed. At the end of each run the system prints out a normal termination message or an error comment.

IX. BIBLIOGRAPHY

Selected Introductory Papers:

FLPL
Gelernter, H., J. R. Hansen, C. L. Gerberich. "A Fortran-Compiled List-Processing Language." JOURNAL OF THE ASSOCIATION FOR COMPUTING MACHINERY, 7:2, April 1960, p. 87-101.

IPL V
Newell, A., and F. Tonge. "An Introduction to Information Processing Language V." COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY, 3:4, April 1960, p. 205-211.

LISP
McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY, 3:4, April 1960, p. 184-195.

Woodward, P. M., and D. P. Jenkins. "Atoms and Lists." THE COMPUTER JOURNAL, 4:1, April 1961, p. 47-53.

COMIT
Yngve, Victor. "The COMIT System for Mechanical Translation." PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON INFORMATION PROCESSING, PARIS, 1959, p. 183-187.

Reference Manuals:

FLPL
Hansen, J. R. "The Use of the Fortran-Compiled List-Processing Language." INTERNATIONAL BUSINESS MACHINES RESEARCH REPORT, RC-282, 1960.

IPL V
Newell, A., et al. INFORMATION PROCESSING LANGUAGE V MANUAL, Prentice-Hall, Inc., Englewoods Cliffs, New Jersey, 1961.

LISP
McCarthy, John, et al. LISP I PROGRAMMER'S MANUAL. Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1960.

McCarthy, John, et al. LISP 1.5 PROGRAMMER'S MANUAL. Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1960.

COMIT

Yngve, Victor. COMIT PROGRAMMERS' REFERENCE MANUAL. Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1961.

Yngve, Victor. INTRODUCTION TO COMIT PROGRAMMING. Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1961.

DEPARTMENT OF COMMERCE ★ NATIONAL BUREAU OF STANDARDS ★