Reprinted from the COMMUNICATIONS OF THE ASSOCIATION FOR COMPUTING MACHINERY
Volume 3, Number 4, April 1960
Made in U.S.A.

An Introduction to Information Processing Language V*

A. NEWELL AND F. M. TONGE, *The RAND Corp., Santa Monica, California*

Introduction

This paper is an informal introduction to Information Processing Language V (IPL-V), a symbol and list-structure manipulating language presently implemented on the IBM 650, 704 and 709. It contains a discussion of the problem context in which a series of Information Processing Languages has developed and of the basic concepts incorporated in IPL-V.[1] A complete description of the language can be found in the IPL-V Programmer's Manual [4, 5].

Development of the IPL Series

There exist many tasks that men can perform reasonably well without knowing in detail how they perform them. Playing chess, making a business decision, or proving theorems are examples. At some level, the computer can behave only in a manner that its users have specified. Getting the computer to play chess or prove theorems, using the same problem-solving techniques as humans, poses communication problems with the machine far beyond those of expressing formal algebraic manipulations. The user must somehow communicate to the machine his incomplete knowledge of how to behave in these complex situations. The IPL series of programming languages has been developed as an aid in constructing problem-solving programs using the adaptive, cut-and-try methods ("heuristics") characteristic of human behavior—as a research tool in the study of heuristic problem-solving.

IPL-I originated as a language for expressing a theorem-proving program in the sentential calculus [6] and was never implemented on a computer. IPL-II and IPL-III were coded for The RAND Corporation's JOHNNIAC and used for the Logic Theorist [7].

Next, a group at Carnegie Institute of Technology prepared an IPL for the IBM 650 [8], a project that has developed into IPL-V. At the same time a similar system, IPL-IV, was coded for the JOHNNIAC and is being used for a chess program [9] and a heuristic program to balance production assembly lines [10]. Major programs are being run or debugged in IPL-V in the simulation of human cognitive processes. These include work in the fields of discrimination learning [11], binary choice [12], and theorem-proving in certain formal areas [13].

The last IPL to date, IPL-VI [14], was written as an order code proposal for a computer that would realize an information processing language directly and hence achieve far more rapid execution than the current interpretive realization on conventional machines.

Problem Interests

We summarize below the characteristics of problems for which the IPL's were developed. This also indicates the type of problems for which IPL-V is a sensible programming system.

(1) The problem basically involves manipulating symbols that have other than numerical meaning and in other than algebraic systems.

(2) The particular storage requirements of the problem-solving program cannot be specified in advance; complex data structures are developed as the program proceeds. For example, a program [11] for memorizing lists of nonsense syllables builds up a net of discriminations for recognizing the different syllables. The size, shape and elaborateness of this net depend entirely on the particular list of syllables presented to the program.

(3) The relationships between elements of data are restructured during the program's operation. New associations must be represented and old ones deleted.

(4) The problem-solving process is naturally expressed at several levels of discourse, each built upon the lower levels. Thus, in the chess program there is a language for talking about the board and the pieces, a higher language for talking about particular pieces as a consequence of their position (for example, bearing on a particular square), and a still higher language for talking about desirable situations (as, control of the center).

(5) The problem-solving procedure will be modified frequently as the program is developed and tested. This change reflects the use of the computer as a means of studying and learning about the problem. Consequently, the program must permit easy modification at various levels and with a minimum of interaction with the rest of the program.

The IPL Computer

IPL-V is a formal language in terms of which information can be symbolized and processes specified for manipulating the information. IPL-V allows two kinds of expressions: *data list structures*, which contain the information

* Presented at the meeting of the Association, Boston, Mass., 1959.

[1] The name "Information Processing Language" was given to the series in its early days, and seems appropriate. But certainly LISP [1], FORTRAN List Processing Language [2], COMIT [3], and others yet to come are just as truly information processing languages as the IPL series.

to be processed, and *routines*, which define information processes. We use the term "IPL Computer" to refer to the IPL-V system as implemented on one of our object machines—650, 704, or 709.

The IPL Computer consists of:

(1) a set of *cells* that hold IPL words—known as the total *available space*;

(2) a stock of *symbols* used to form IPL expressions (within the computer all symbols are addresses, and thus name cells);

(3) a set of *primitive processes* which the computer can carry out without further IPL interpretation;

(4) an *interpreter* that interprets routines and performs the processes they define.

Representation of Data

Symbols. Two types of symbols are available to the programmer—*regional* and *local*. Regional symbols consist of an alphabetic character followed by a relative number —as, A27, C5, G1000. These are the relative symbols of normal programming usage. Local symbols are expressed as a regional character 9 followed by an arbitrary number—as, 9-7, 9-100. Local symbols are treated as pure symbolics, with their meaning constant within a particular IPL expression. The same local symbols are used with different meanings in different routines or data list structures.

All symbols not explicitly used by the programmer, and the cells they name, are available to the program during processing and are called *internal* symbols.

Lists. Generally, a larger unit of data than a single symbol is needed. In IPL, the list is this unit of data, and basic processes for manipulating lists exist. Normally, each cell in use holds an IPL word, consisting of two prefixes, P and Q, and two symbols, SYMB and LINK. Symbols are linked together in lists in the manner indicated in figure 1, which shows a list of the symbols S1, S2, S3. L0 is the name of the list and of the cell called the *head* of the list. The names of the *list cells* are internal symbols. The LINK of each cell holds the name of the cell holding the next symbol on the list. The final list cell has the *termination symbol*, 0, as its LINK. By convention, the first symbol on a list is stored in the first list cell, the SYMB part of the head being reserved for another use. (The internal symbols linking cells of a list are normally omitted, since they are supplied by the IPL system and need not concern the programmer.) Thus, several symbols can be associated into a unit of data by placing them on a list. These symbols may be the names of other lists or of more complicated structures.

Description Lists. A list can have associated with it certain descriptive information that can be added to, altered or deleted at will. This is accomplished through the *description list* mechanism. The symbol stored in the head of a list is the name of the list's description list. The symbols on a description list are considered in pairs, the first member of the pair being the attribute and the second member being its value. Each attribute corresponds to a function, with a value for the particular argument (unit of data) being described. Thus, for the unit of data "grass" the value of attribute "color" would be "green." Figure 2 illustrates a list, L1, with symbols S4 and S5, which is described by the two attributes A1 and A2.

The IPL-V primitive process "find the value of attribute A1 of L1" would produce the symbol V1. Additional descriptive information can be associated with a list during processing by performing the primitive process that assigns an attribute and its value to a symbol. Similarly, new values can replace the present ones, or an attribute and its value can be deleted entirely. The programmer needs no knowledge of the actual structure of the description list. All necessary processing is done by the appropriate primitive processes, which search the list for the desired attribute and take appropriate action.

Data Terms. Thus, symbols are given meaning by the list that they name and by descriptive information associated with them. Symbols can also name information beyond the scope of the Information Processing Language itself—such as integer or floating point numbers, binary fields, or alphanumeric information. Such information is encoded into the cell named by the symbol being defined and is manipulated by IPL processes operating on the symbol. The symbol and the associated encoded information are known as a *data term*. Primitive processes in IPL-V perform arithmetic operations on numerical data terms and print all types of data terms just mentioned. Other new types of data terms can be defined and appropriate primitive processes introduced into the system easily.

List Structures. More complicated units of data can be defined through the use of local names. A *list structure* consists of a *main list*, having a regional or internal name, and all those structures named on the main list having local names. Figure 3 illustrates a data list structure consisting of the main list, L2, description list 9-1 with data

Name	*PQ*	*Symb*	*Link*
L0		0	36
36		S1	508
508		S2	13
13		S3	0

FIG. 1. Simple List

Name	*PQ*	*Symb*	*Link*
L1		9-1	
		S4	
		S5	0
9-1		0	
		A1	
		V1	
		A2	
		V2	0

FIG. 2. Description List

term 9-10 (the integer 15) as the value of attribute A5 and sublists 9-7 and 9-5.

Primitive processes in IPL create, copy, erase and move to auxiliary storage a list structure as a single entity. Also the necessary processes exist so that a program can scan and process list structures in other ways.

Push-Down Lists for Storage Cells

The programmer can also use cells as working storage; that is, he can store symbols in their SYMB part. In this case the LINK of the storage cell holds the termination symbol.

Often it is desirable to store information in a storage cell without destroying the information already in the cell. For example, as is developed in more detail later, the interpreter always holds the name of the cell containing the current IPL instruction in a particular storage cell, named H1. If that instruction designates a subprocess to be interpreted, the interpreter must keep its location in the subprocess, but without losing its place in the higher routine. Indeed, since the subprocess may itself execute a subprocess, and so forth, an indefinite number of locations in various routines must be saved.

This problem is resolved through the *preserve* and *restore* operations. To preserve a cell is to take an unused cell from available space and copy into it the total contents of the cell being preserved. The name of this copy cell is then stored in the LINK of the preserved cell. Other symbols can then be stored in the cell without destroying its original contents. The original state of the cell is returned by the inverse operation, restore. The list of preserved symbols associated with a cell is called its *push down* list, and the operations preserve and restore are also called *push down* and *pop up*.

Figure 4 shows the status of cell H1, initially holding

Name	*PQ*	*Symb*	*Link*	
L2		9-1		
		9-7		
		G4	0	
9-1		0		
		A1		
		V1		
		A5		
		9-10	0	
9-7		0		
		S4		
		9-5	0	
9-10	1			15
9-5			0	
		Z1		
		L2	0	

FIG. 3. List Structure

Name	*PQ*	*Symb*	*Link*
H1		Q5	387
387		K3	0

FIG. 4. Push Down List

K3, immediately after it has been preserved and the symbol Q5 stored in it.

Thus, the interpreter, in beginning interpretation of a subprocess, pushes down H1 before recording the name of the subprocess as the new current instruction address. And, upon completing a subprocess, the interpreter pops up H1 to obtain the last current instruction address of the higher routine.

Available Space List

As lists in storage are built up and altered, cells are continually brought into use and discarded—as in push down and pop up operations. Some system is needed to keep track of which cells in storage are unused. In IPL all currently unused cells are linked together on a list, the *available space list*, named H2. Any process, or the interpreter, desiring a cell takes the first one on this list. Likewise, cells no longer needed are returned to the available space list. This device frees the programmer from problems of memory assignment, and allows him to apply at will various processes that modify the structure of memory.

Interpretation

Routines. An IPL routine is a list of instructions. (The format of instructions is explained later.) During interpretation the IPL interpreter examines each instruction word in sequence and carries out the process it designates. This process may be execution of some other routine. The rules for forming routines in IPL and the manner in which interpretation is mechanized insure that every routine is a closed subroutine usable by any routine, including itself. All routines are forced into a subroutine format, and all programs into a hierarchical organization, through a particular mechanization of the linkage between routines and conventions about specification of parameters and use of working storage.

Linkage—the Current Instruction Address List. As was mentioned above, the address of the cell holding the current instruction is stored in a particular cell, H1. If this instruction designates a subprocess to be interpreted, H1 is pushed down before interpretation of the subprocess begins and is popped up after that interpretation is completed. Thus, the return linkage for a routine is held in the push down list associated with H1, called the Current Instruction Address list. The programmer simply designates the subprocess to be executed by name; linkage is handled automatically by the interpreter.

Specification of Inputs and Outputs—the Communication Cell. The inputs to any process are specified by storing them in the Communication Cell, named H0. H0 is preserved before each input is entered, so that the set of inputs to a process are the top symbols in H0's push down list. By convention, each process removes its inputs from H0. Likewise, each process leaves any outputs it produces in H0.

Q = 0 S = SYMB
Q = 1 S = Symbol in cell named SYMB
Q = 2 S = Symbol in cell named by symbol in cell named SYMB.

For example, given the following two cells:

Name	*PQ Symb*	*Link*
C0	B0	
B0	K0	

we have as the designated symbol, S:

0C0 = C0
1C0 = B0
2C0 = K0

FIG. 5. Designation Operation

P = 0 EXECUTE S. S is assumed to name a routine or a primitive. The process it specifies is carried out.
P = 1 INPUT S. The Communication Cell H0 is preserved; then a copy of S is put in H0.
P = 2 OUTPUT TO S. A copy of the symbol in H0 (hereafter abbreviated as (0)) is put in cell S; then H0 is restored.
P = 3 RESTORE S. The symbol most recently placed in the push down list of cell S is moved into S; the current symbol in S is lost.
P = 4 PRESERVE S. A copy of the symbol in cell S is placed in the push down list of S; the symbol remains in S.
P = 5 REPLACE (0) BY S. A copy of S is put in H0; the current (0) is lost. (This is analogous to the normal "load accumulator.")
P = 6 COPY (0) IN S. A copy of (0) is put in cell S; the current symbol in S is lost and (0) is unaffected. (This is analogous to the normal "store accumulator.")
P = 7 BRANCH TO S IF H5−. If H5 is +, LINK names the cell containing the next instruction to be performed. (This is the normal sequence of instructions.) If H5 is −, then S names the cell containing the next instruction to be performed.

FIG. 6. Operation Code

Working Storage. A set of ten cells, W0-W9, are reserved for Public Working Storage (through a process may use any available cell for working storage if it so desires.) If routines using a public working storage cell first preserve the cell, thus adding the information in the cell to the push down list associated with the cell, and when through restore the cell, any routine can execute any routine, including itself, as a subprocess without the danger that its information in working storage will be violated.

By convention, the Communication Cell and the Public Working Storage are *safe* cells. That is, any process using them is morally bound to first preserve them and when finished restore them. This explicit handling of the context in which a routine operates offers flexibility in several ways: outputs of a process can be left in the Communication Cell as inputs of a later one; each routine is an independent subroutine with respect to working storage. It has the drawback of requiring explicit handling of each safe cell used.

Test Cell. Many processes, in addition to producing other outputs, result in the information "yes" or "no": as, "yes, I have found the location of that symbol on this list," or "no, these two symbols are not identical." The results of such binary decisions are symbolized in the Test Cell, H5 (+ for "yes" and − for "no").

Instruction Format. Each instruction of a routine is expressed as an IPL word. The process to be carried out is designated by the prefixes P and Q and by SYMB. LINK is the name of the next cell on the routine list.

The Q prefix specifies a designation operation to be performed upon SYMB. The result of this operation is the *designated symbol*, S. This designation operation is a form of indirect addressing. The three degrees of designation available in IPL-V are illustrated in figure 5.

The P prefix specifies the *operation* to be performed upon the designated symbol. These operations accomplish the setup, execution and cleanup of routines. The eight P prefixes are explained in figure 6.

Interpretive Cycle. The interpreter takes a program and interprets it as a sequence of primitive processes, executing each of these in turn. This interpretive process consists of the cycle of operations illustrated by the flow diagram in figure 7.

Basic Processes

The IPL-V system includes approximately 150 basic processes. While clearly not a minimal set—indeed, some of the basic processes are coded in IPL-V itself—experience with earlier IPL's indicates that this is a useful one. The several classes of basic processes are described below. The GENERAL processes include such instructions as "no operation," "test if two symbols are identical," "set the signal in H5 plus," and "halt."

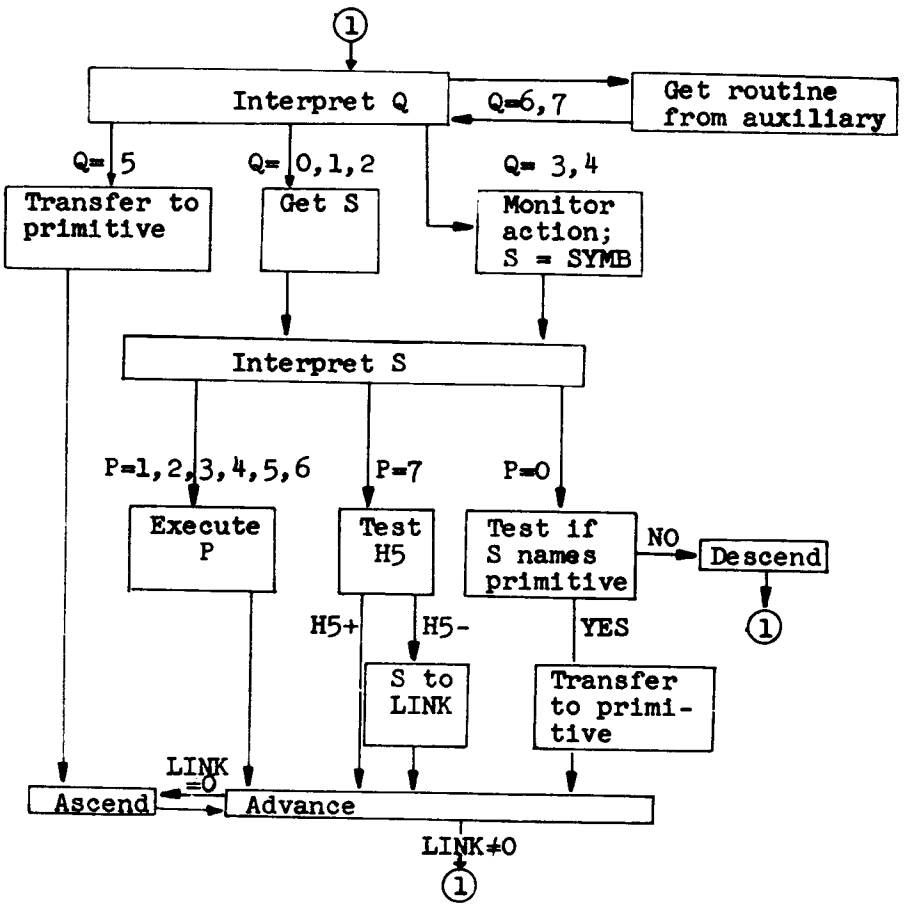

FIG. 7. Interpretive Cycle

Among the DESCRIPTION LIST processes are "find the value of an attribute of an object," "assign a new value to an attribute," and "erase an attribute and its value."

The PUBLIC WORKING STORAGE processes make it possible to preserve, restore, or move symbols from the Communication Cell into several of the W's with one operation.

The LIST processes include such operations as "locate the next symbol on a list," "insert a symbol on a list," "erase a list structure," and "copy a list structure."

The ARITHMETIC processes contain such operations as "add," "multiply," and "test if *a* greater than *b*." The system also includes a basic operation that generates random numbers within a specified range.

Through the DATA PREFIX processes the programmer can identify the various types of symbols and data terms present in the system and so construct other list structure processes. These processes include "test if a symbol names a data term," and "make a symbol local."

The AUXILIARY STORAGE processes enable the programmer to "file" data list structures in auxiliary storage and to "move" filed data into immediate storage.

The INPUT-OUTPUT processes permit reading or writing data list structures using any peripheral equipment present on the object computer. Data punched out on cards or written on external tapes is in the appropriate form for re-entry either at loading or by the read process. Full control of print column and line spacing is available within the IPL system.

Repetitive operations can be handled in IPL-V with loops, utilizing the conditional branch, or by a special class of processes, called *generators*. A generator is a process that produces a sequence of outputs and applies to each output a specified process. The process that the generator applies is an input to the generator and is called the subprocess. The generator is associated with the kind of sequence it produces, and will apply any subprocess to the elements of the sequence. (The subprocess must obey a system convention on how to signal the generator to continue or stop producing elements.) Thus, the generator, just like the "iteration" statements of algebraic compilers, accomplishes a separation of the "production of elements" part of a loop from the "processing" part.

The subprocess is executed for each element of the output sequence as though it were a continuation of the process firing the generator (the superprocess)—that is, as though the generator had made no use of the Communication Cell or Public Working Storage. Generators are different from all other IPL processes in that two contexts of information in working storage must coexist in the computer—that of the generator and that of the superprocess and subprocess. There is an alternation of both control and context between the generator and the subprocess. To produce an element of the sequence, the generator must be in control and its context should occupy the W's; to process the element, the subprocess must be in control and its context (the context of the super-routine) should occupy the W's. Hence the strict hierarchy of routines and subroutines is violated, and special pains have to be taken to see that information remains safe and that each process works in its appropriate context.

To handle this special housekeeping, the GENERATOR HOUSEKEEPING processes are provided. These processes insure that the generator's context is hidden away before the subprocess is executed, and returned to the W's after the subprocess is completed. The programmer uses these processes in coding generators. Some generally useful generators—"generate the symbols on a list," "generate the cells of a data list structure" and "generate the cells of a tree structure"—are included among the basic list processes.

It is possible to prepare additional machine language routines and append these to the basic system, entering them with other programs during loading. These machine language routines will generally be coded in the assembly system appropriate to the object machine and assembled prior to IPL loading.

Operating Aids

Debugging aids include selective tracing of any routines desired, snapshots of any data (including system cells) at the beginning and/or end of tracing, and a post mortem dump of any data. The system also includes provision for saving the program on tape or cards for later restart.

An Example of IPL Coding

As a simple example of coding in IPL, consider the problem of testing if a given symbol occurs in a given tree. A tree is a list structure in which no sublist occurs more than once. The list structure of figure 3 is a tree.

We shall code this problem in two ways—first using the basic process for moving down a list cell by cell (J60), then using the basic process for generating the cells of a tree structure (J102).

The basic processes required are given below. (Just as (0) stands for the symbol in H0, (1) indicates the symbol one down in H0's push down list, (2) the symbol two down, and so forth.)

J50: PRESERVE W0, THEN MOVE (0) INTO W0.

J60: LOCATE NEXT SYMBOL AFTER CELL (0). (0) is assumed to be the name of a cell. If the next cell exists (LINK of (0) not a termination symbol), the output (0) is the name of the next cell and H5 is set +. If LINK is a termination symbol, then the output (0) is the name of the last cell—i.e., input (0)—and H5 set −.

J132: TEST IF (0) IS A LOCAL SYMBOL. Set H5+ if (0) is local; set H5− if not.

J2: TEST IF SYMBOL (0) = SYMBOL (1). Set H5+ if equal; set H5− if not.

J30: RESTORE W0. (Same as 30W0.)

J131: TEST IF (0) NAMES A DATA TERM. Set H5+ if (0) is data term; set H5− if not.

J5: REVERSE THE SIGN OF H5.

J8: RESTORE H0. (Same as 30H0).

J102: GENERATE CELLS OF TREE (1) FOR SUBPROCESS (0). The subprocess named (0) is performed successively

with the names of each of the cells of the tree (1) as input. The order is that the cells of each sublist are generated before going on with the higher list. The subprocess signals the generator to continue by setting H5+; it signals the generator to stop by setting H5−. The generator terminates with H5+ if it was not stopped by the subprocess, and with H5− if it was stopped. Also, H5 is set + to the subprocess if the input cell is the head of a sublist, and is set − otherwise.

Formally, E0 is defined as:

E0: TEST IF SYMBOL (0) OCCURS IN TREE (1). Set H5+ if (0) occurs; set H5− is it does not.

First, E0 using J60 to move down the list examining each symbol:

Name	*PQ*	*Symb*	*Link*	*Comments*
E0		J50		Push down W0 and move the test symbol to W0.
9-3		J60		Locate the next cell of the tree
	70	9-1		If no more cells, exit with H5−
	12	H0		Input the symbol in the next list cell
	11	W0		Input the test symbol
		J2		Test if symbols are the same
	70	9-2		If same, exit with H5+
9-1	30	H0	J30	Discard list reference, pop up W0.
9-2	12	H0		Input list symbol again
		J132		Test if local
	70	9-3		If not local, continue down this list
	12	H0		Input list symbol again
		J131		Test if names data term
	70		9-3	If data term, continue down this list
	12	H0		If not data term, names sublist
	11	W0		Input the test symbol
		E0		Apply this process to sublist
	70	9-3	9-1	If found on sublist, exit with H5+

This same routine, using J102 to produce the cells of the list structure:

Name	*PQ*	*Symb*	*Link*	*Comments*
E0		J50		Push down W0 and move the test symbol to W0.
	10	9-10		Input the name of the subprocess
		J102		Generate cells of tree for subprocess 9-10.
		J5	J30	Reverse final sign, pop up W0
9-10	70		J8	If head, discard without examining
	52	H0		Input symbol on list, destroying cell reference
	11	W0		Input test symbol
		J2	J5	Test if identical; reverse sign

Note that the subprocess reverses the sign produced by J2 for its signal to the generator. If the two symbols were identical, the subprocess must stop the generator, and so changes the + to −. If the symbols were not identical, the generator must continue and so the appropriate signal from the subprocess is +. The superroutine E0 reverses the generator's signal since the subprocess would stop the generator (with H5−) only if it found the test symbol.

A Final Remark

While the value of this system can be adequately assessed only through its use, we feel that we have gained considerably by this approach to symbol manipulation. We have gained the flexibility to do many interesting tasks, tasks that could not be done in any straightforward way in more machine-oriented programming systems. Both complex structures and complex processes can be designated by a single symbol and manipulated as single units. We have shaped the system to do *easily* those information processing tasks in which we are interested and which we found difficult to specify in other commonly used programming languages.

We have paid in operating speed and storage utilization. This payment is quite severe for standard arithmetic manipulations, for which conventional computers were specifically designed. It becomes less severe as the programs and data manipulations become more complex, and elaborate housekeeping conventions of some sort are required, no matter what the programming system.

Acknowledgment

In addition to the authors of this paper, E. A. Feigenbaum (704 system), N. Saber of the University of Pittsburgh (650 system), G. H. Mealy of The RAND Corporation (formerly Bell Telephone Laboratories) (704 system), and B. F. Green, Jr., and A. K. Wolf of Lincoln Laboratories (709 system) have participated in developing IPL-V. The basic ideas stem from the work of A. Newell, J. C. Shaw and H. A. Simon. C. Hensley of IBM participated in the early design effort. The support of the Graduate School of Industrial Administration, Carnegie Institute of Technology, is gratefully acknowledged.

REFERENCES

1. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine (The LISP Programming System), "Quarterly Progress Report No. 53, Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 15, 1959.
2. Gelernter, H. The Fortran List Processing Language. IBM dittoed paper, 1959.
3. Yngve, V. A programming language for mechanical translation, *Mech. Translation 5* (July, 1958), 1.
4. Newell, A., F. M. Tonge, E. A. Feigenbaum, G. H. Mealy, N. Saber, B. F. Green, Jr., and A. K. Wolf. Information Processing Language V Manual, Section I: The Elements of IPL Programming. The RAND Corporation Paper P-1897, 1960.
5. Newell, A., F. M. Tonge, E. A. Feigenbaum, G. H. Mealy, N. Saber, B. F. Green, Jr., and A. K. Wolf. Information Processing Language V Manual, Section II: Programmers' Reference Manual, The RAND Corporation Paper P-1918, 1960.
6. Newell, A., and H. A. Simon. The Logic Theory Machine. *Transactions on Information Theory*, Vol. IT-2, No. 3, IRE, September, 1956.
7. Newell, A., and J. C. Shaw. Programming the Logic Theory Machine. *Proceedings of the 1957 Western Joint Computer Conference*, IRE, February, 1957.
8. Hensley, C. B., A. Newell, and F. M. Tonge. 650 IPL Information Processing Language. C.I.P. Working Paper No. 9, Carnegie Institute of Technology, April 30, 1958 (ditto).
9. Newell, A., J. C. Shaw, and H. A. Simon. Chess playing programs and the problem of complexity. *IBM J. Res. Develop. 2* (Oct. 1958), 4.

10. Tonge, F. M. Summary of a heuristic line balancing procedure. The RAND Corporation Paper P-1799, 1959.
11. Feigenbaum, E. A. An information processing theory of verbal learning. The RAND Corporation Paper P-1817, October, 1959.
12. Feldman, J. Analysis of predictive behavior in a two-choice situation. Unpublished doctoral dissertation, Carnegie Institute of Technology, 1959.
13. Newell, A., J. C. Shaw, and H. A. Simon. Report on a general problem-solving program. The RAND Corporation Paper P-1584, January, 1959.
14. Shaw, J. C., A. Newell, H. A. Simon, and T. O. Ellis. A command structure for complex information processing. *Proceedings of the 1958 Western Joint Computer Conference*, IRE, May, 1958.