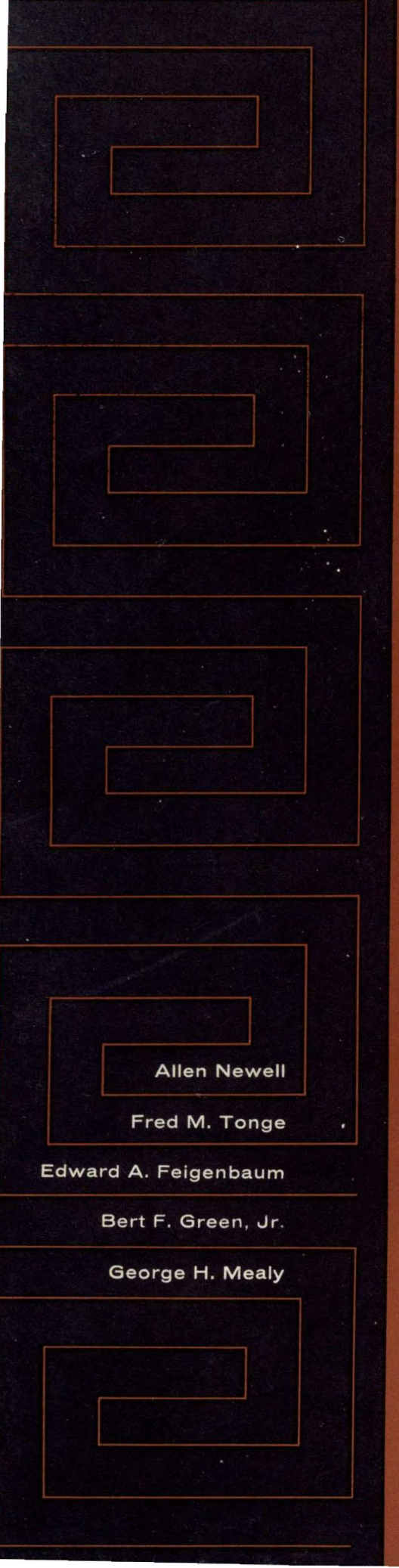


Allen Newell

Fred M. Tonge

Edward A. Feigenbaum

Bert F. Green, Jr.

George H. Mealy

The RAND Corporation

INFORMATION PROCESSING LANGUAGE-V MANUAL

Second Edition

INFORMATION PROCESSING LANGUAGE-V MANUAL

IMPLEMENTATIONS

Burroughs 220
Alan Olson, Gilbert Hansen, Ranan B. Banerji

Control Data 1604
Robert K. Lindsay, John Dauwalder

Control Data G-20
David Cooper, Richard G. Shoup

Ferranti Mercury
David Cooper

IBM 650
Nicholas K. Saber, Ted Van Wormer

IBM 704
Charles L. Baker, Edward A. Feigenbaum, Hugh S. Kelly, George H. Mealy

IBM 709-7090 (Lincoln System)
Bert F. Green, Jr., Alice K. Wolf, Michael Kahn

IBM 709-7090 (RAND System)
Charles L. Baker, Hugh S. Kelly

IBM 1620
Wendell T. Beyer, John D. MacDonald

Philco 2000
Stuart S. Shaffer, Clark Weissman, Julian Feldman

UNIVAC 1105
Bobbie F. Caviness, H. Lee Butler

UNIVAC 1107
Gilbert Hansen, Ranan B. Banerji

AN/FSQ-32
Michael Kahn, Clark Weissman

INFORMATION PROCESSING LANGUAGE-V MANUAL

Second Edition

Allen Newell
Fred M. Tonge
Edward A. Feigenbaum
Bert F. Green, Jr.
George H. Mealy

Second Edition prepared by
Hugh S. Kelly
Allen Newell

The RAND Corporation

PRENTICE-HALL, INC. ENGLEWOOD CLIFFS, N. J.

Printed in the United States of America

46440-C

PREFACE

The development of Information Processing Language-V (IPL-V) has been a cooperative effort of many people in numerous organizations over a period of almost five years. The basic ideas from which IPL-V was derived came from the work of Allen Newell, J. C. Shaw, and Herbert A. Simon on the earlier IPL's. These earlier languages were private to a small research group at The RAND Corporation and Carnegie Institute of Technology working with JOHNNIAC, a computer at RAND. IPL-V was the first IPL to be made available for public use. A preliminary version for the IBM-650, called 650-IPL, was developed by Fred M. Tonge and Carleton B. Hensley. Although the title page lists the people principally involved in the design and original implementations of IPL-V, many others have been concerned with the continued maintenance, revision, and updating of the various systems. In particular, the contributions of Gloria Goldberg and Einar Stefferud deserve mention.

Throughout the entire period, the primary support for the development of IPL-V has come from The RAND Corporation under U.S. Air Force Project RAND. However, many other organizations have, at one time or another, provided substantial amounts of support, both in manpower and computing time. The organizations include Carnegie Institute of Technology, Lincoln Laboratory, Bell Telephone Laboratories, System Development Corporation, University of North Carolina, University of Pittsburgh, University of Texas, Case Institute of Technology, MESA Scientific Corporation, Hughes Aircraft Company, and the Computing Unit of the University of London. In addition, grants and contract support for various parts of this development have been made by the Social Science Research Council, Carnegie Corporation of New York, and the Advanced Research Projects Agency. The revision of the Manual was supported by The RAND Corporation with its own funds.

Allen Newell
January 1964

CONTENTS

INTRODUCTION TO THE FIRST EDITION

The language described in this Manual is an addition to the techniques for using digital computers. It can be placed in perspective by describing briefly the developments in the computer and programming arts that preceded it, and current work that is closely allied to it.

We now recognize that the digital computer is a device for manipulating symbols of any kind, in any way. Its genesis, however, lies in the desk calculator, which is a device for automatically performing the four operations of addition, subtraction, multiplication, and division. The core of the computer, as initially envisioned, is an arithmetic unit which performs these same four operations extremely rapidly, in anywhere from a tenth of a second to a few millionths of a second. It is impossible to make good use of such a device in the manner of a desk calculator, since the entire time of calculation would be taken up with human decisions as to what numbers to process next, and with human actions to put the numbers into the machine. Several innovations were made to take advantage of the tremendous speeds in arithmetic. First, a way was found to remember the numbers associated with a computation. Second, a small set of "non-numerical" operations was invented to do the tasks normally done by the human working with a desk calculator. These were of three types: moving a number from one cell in the memory to another without modification; sensing simple conditions (such as whether a number

in a cell is negative) and taking differential action depending on the result; and actuating various reading and writing mechanisms, such as printers, card readers, and the like. Third, a language of imperatives was invented, which consisted of a sequence of instructions. Each instruction stated that one of the operations of the machine was to be performed on the numbers in specified memory locations. Finally, a way was found in which the machine could interpret each instruction, perform the actions designated, and then automatically pass on to the next instruction. Thus, a memory unit and a control unit were added to the arithmetic unit, and the automatic digital computer was produced in essentially its present form.

It was already known at the time these innovations were taking place that the resulting machines would have almost unlimited and universal capacities. The work of Turing is central in this connection, but the entire field of symbolic logic contributed to this knowledge. Turing showed in 1937 that a machine with certain relatively simple capacities could compute any number that could reasonably be considered computable. By an intuitive generalization, such a computer can be made to perform any process that can be specified. However, this abstract knowledge of great power is not equivalent to knowing what sorts of symbol manipulations are needed and are useful to achieve various practical results. This latter knowledge has grown up only slowly from the attempts to use the computer in ever widening contexts.

Problems in Writing Programs[†]

Writing programs of any sort is difficult enough. As delivered by the engineers, the machine makes some very specific and rigid demands on the user. The instructions require proper names for both operations and locations--that is, fixed codes, which correspond to the circuitry of the machine. Working with absolute addresses, as these proper names are called, has many drawbacks. Most important is having to select absolute locations in order to get on with the coding, without knowing whether later there will be good reason to want these cells for other purposes. These conflicts arise with great regularity, since computer routines are usually put in the memory in consecutive cells, so that starting a routine at a given location automatically places a claim on many following locations.

A solution to this difficulty was found in regional addressing and symbolic addressing. Both of these allow the coder to write down a symbol to refer to an address, but to delay the assignment of absolute addresses until all the requirements for memory space are available. Regional addressing assigns addresses according to a scheme whereby A5 becomes the fifth cell after the "origin cell," A. The origin cells, which control the relationship between regional symbols and absolute

---

[†]See Ref. 9 for a rather complete description of the programming techniques currently available. It also contains an extensive bibliography. (The References begin on p. xxxi.)

addresses, are assigned by the programmer after the entire routine has been written. Symbolic addressing assigns addresses according to an arbitrary dictionary, created after the entire routine is finished.

Regional and symbolic addressings are conventions that the programmer invents and adheres to, and they involve no change in the engineering characteristics of the computer. They can legitimately be considered an increase in the computer's symbolic capabilities because another program called an assembly program makes the translation from regional and symbolic addresses to absolute addresses. The "computer-plus-assembly-program" acts in every way like a machine with improved design. The programmer can now use the new conventions as if the machine understood them directly. Most of the improvements in computers have been achieved by constructing programming systems to augment their capabilities, rather than through hardware modifications.

The machine also requires that all numbers, constants, and operation codes be translated into the fixed form acceptable to its circuitry. This is especially vexing with machines that work internally with a binary representation. It is desirable to write numbers in ordinary decimal form and to refer to the operations by easily remembered mnemonic codes. These increases in capability have been rather universally achieved by adding the necessary translation facilities to the assembly program.

As constructed, the computer has a small repertoire of instructions it can execute, each producing a very small step forward in the total computation. However, it

is desirable to work with much larger units of processing and to preserve the effort spent in constructing such units. The subroutine has emerged as the standard solution to this problem. A subroutine is a block of coding written under standard conventions so that others at a later time can incorporate the coding into their program. The conventions govern the transfer of control to the subroutine and the return of control to the main program that uses the subroutine. They also govern how to give the subroutine its input information and how to get from it the products of its computation. Originally, the subroutine was a pure programming innovation, but its use has become universal, and by now almost all machines have special instructions that make subroutines easy to use. In connection with assembly programs, libraries of subroutines are built up that can be called in by name and incorporated into the program automatically. From a linguistic viewpoint, the use of subroutines is an abbreviation device, whereby a name can be assigned to a collection and used in place of it.

It is possible to go further than the isolated subroutine, and to build up whole systems of subroutines to be called into action by means of a "pseudo-code." A typical example is the extension of a machine to handle floating point arithmetic (arithmetic on numbers written as $.621 \times 10^2$ rather than 62.1). For most scientific calculations this is the preferable form, since the problems of keeping track of the decimal point can be avoided. None of the early machines had circuitry for performing floating point arithmetic. Rather than just

create a subroutine for each operation, an instruction format (the pseudo-code) was set up, corresponding directly to the regular instruction format of the machine, in which the arithmetic operations were understood to be floating point operations. The machine itself, of course, could not interpret these instructions. A program called an interpreter was used to decode these pseudo-instructions and to execute subroutines corresponding to them. The machine was made to behave as if it followed different instructions from the ones the engineers had built into it. This is achieved at a great cost in speed, since each pseudo-instruction requires several machine instructions to interpret it. Partly for this reason, pseudo-codes have never been very different from standard machine codes, since they would then have required elaborate interpretation. IPL-V, the language described in this Manual, is a pseudo-code.

The fundamental form of the machine language, although universal in its applicability, is rather far from the familiar and powerful language of algebra. As sophistication in programming has increased, computers have finally been given the capability of understanding algebraic notation. The relative complexity of algebra requires that this be a programming innovation, rather than a hardware one. A translation program accepts the language containing algebraic formulas and produces a code in machine language that will accomplish the same computation. These translation programs, like FORTRAN and IT, are functionally similar to assembly programs but the translations they accomplish are much more complex, and, instead, they are called compilers or translators.

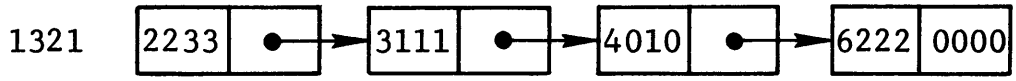
Heuristic Programming and the Simulation of Cognitive Processes

The language described in this Manual stems from recent attempts to program computers for problems that are sufficiently complex, ill-structured, and difficult to require intelligence for their solution by humans. The motivations behind these attempts range from the desire to extend the capabilities of computers, to the desire to understand how humans think, learn, and solve problems. Most of the work in this area has focused on rather formalized tasks, such as proving simple theorems, playing chess or checkers, and performing various symbolic calculations like differentiation and integration.

These programs have revealed some additional desirable symbolic capabilities for computers; most important is a need for a unit of data larger than the single number. Indeed, in these programs the entire pattern of data, both structure and content, evolves during the course of processing. Thus, it is necessary to have the program construct its own data structure dynamically. This implies some convenient way of creating, aggregating, modifying, and referring to units of data.

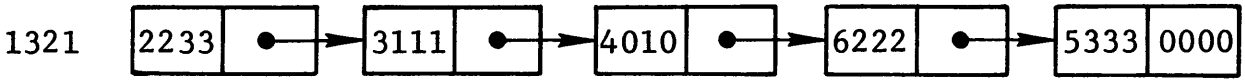
A solution to this problem has been found in the list. A list is a set of words tied together by having the address of each word in the list recorded in the word that occurs just prior to it in the list. Each word in the list contains two addresses: one gives the item of information at this location in the list; the other gives the link to the next word. This is shown below, where

numerical addresses have been written for items and arrows have been drawn for linking addresses:

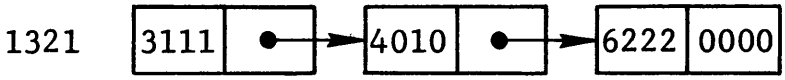


The list is a unit of data, just as a routine is a unit of processing. Its name is the address of the first cell (1321, above). It contains complete information about its structure, including a special link (0000) to indicate the end of the list.

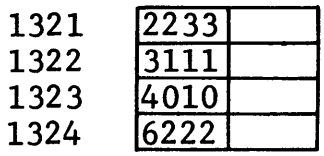
Operations are performed on it as a unit. For example, the operation of inserting an item at the end of a list would take as input the name of the list (1321) and an item (say, 5333) and produce:



Similarly, the operation of deleting the first item on list 1321 (as it was originally) would produce:

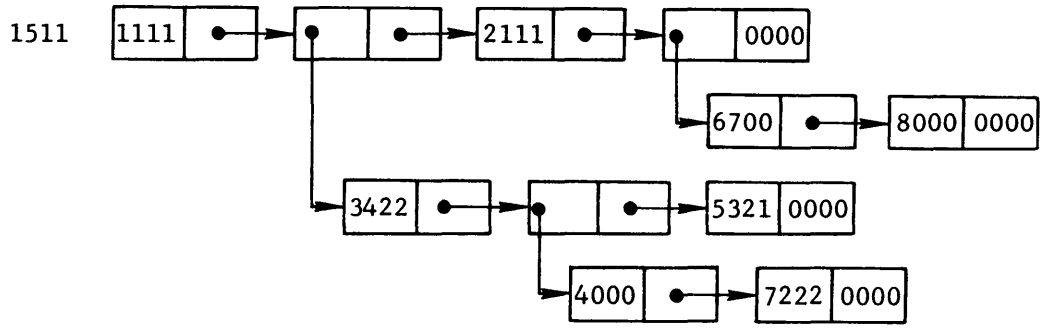


These operations change the structure of the data unit, not just its content. It is this ability that contrasts rather strongly with standard ways or organizing data on the computer. Normally, one would put items in consecutive cells in memory:



There is a simple rule for finding the next item on the list (add 1 to the address of the current cell), just as there is a simple rule for lists (look in the link of the current cell). With index registers, the address-incrementing rule is even easier and quicker to use. More important, no space has been taken to hold all the linking addresses. However, if we wish to insert or delete items, or otherwise modify the structure, then difficulties arise with the sequential scheme. For example, in order to insert after 1322 it is necessary to make cell 1323 available, and this means moving all the items after this into new addresses. No such difficulty arises with lists, since there is no need for the cells to bear any specified relation to each other.

It is desirable to have a unit of data larger than a list. This is called a list structure and is compounded from lists, by having the names of some lists occur as items of information on other lists. A simple list structure is shown below:



The example has a main list of four items, with one sublist of three items and another of two items. One sublist itself has a sublist of two more items. The ability to deal with a list structure as a unit--to copy it, print

it, or search it--is a great advantage.

The use of one address to link together a structure is not an altogether new idea. For example, many machines (e.g., the IBM 650) have a "plus-one" instruction format in which one address of each instruction gives the address of the next instruction. This is a list in the sense we are using the term. The main use of "plus-one" addressing is for efficient operation of machines which have cyclic storage systems (such as a magnetic drum), and which need minimal latency coding. Not until attempts were made to code very complex symbol manipulating programs did lists and list structures, together with the associated list operations for manipulating the structures dynamically, get adequately developed, both technically and conceptually.

There is a technical barrier to be overcome before lists can be used as the basis of a complete and flexible computing system. Imagine the data part of computer memory being initially completely unused, and imagine lists being built up and modified during processing. Then, just as planned, all apparent order in the memory disappears. New words are taken from the gradually diminishing block of unused space. There is no difficulty until, eventually, all the space is used up. By this time there will be lots of unused cells, since many of the lists are no longer needed, and many cells have been removed by delete operations. However, these cells are scattered throughout the memory in a completely haphazard fashion. Some scheme must be incorporated to make all of this space available again, so that processing can continue.

The solution to this is an extremely simple, but elegant, trick--to have a list of available space. All words not otherwise in use form the cells of a list, each cell linking to the next. This list has a known name (it is H2 in IPL-V). Any process that needs space can get the address of an available cell from H2. This cell in turn gives the address of another available cell, and so on. A general responsibility is imposed on all processes of the system to put any cells they make available back on the available space list. Thus, at all times all the free cells are linked together and available to whatever process needs them. This technical device clears the way for the programmer to become almost completely free from problems of memory assignment, and to apply at will various processes that modify the structure of memory.

The work on heuristic programs has also emphasized the need for good conventions for subroutines. The programs are sufficiently complex and hierarchical in nature that the power of abbreviation is extremely useful. Recursive definitions have also been used extensively, and ways for mechanizing these have been necessary. A recursively-defined subroutine is one in which the subroutine executes itself. These arise because list structures are defined recursively--a list structure is a list plus all the list structures whose names occur on the list--so that the most natural way to define processes on list structures is recursively. Recursive routines also arise because of the recursiveness of the problem-solving process. The general way to solve a problem, X,

is to set up some subproblems, say Y and Z, then to try to solve Y, and then to try to solve Z. The process of solving the subproblems is exactly the same routine as the original. The use of lists has allowed simple solutions to both of these problems. Since these are discussed in detail in the remainder of the Manual, there is no need to describe them here.

History of Work on List Languages and Heuristic Programming

Considerable work is going on in constructing programming languages using lists, most of it in connection with work on heuristic programs. This work illustrates both the applications of list-processing languages which have been made to date and the various ways in which list processing can be introduced into programming.

Triggered by the pioneering work of Selfridge[50] and Dinneen[12] on a program for recognizing visual patterns, the work of Newell, Shaw, and Simon began in late 1954. They first worked on chess and then switched to the task of proving theorems in the propositional calculus of Whitehead and Russell.[40] Their earlier languages were tied closely to subject matter--a "chess language" and a "logic language." These languages, collectively called IPL-I, although designed as pseudo-codes, never reached the coded stage.† By the time a coded version appeared, called IPL-II, for use with the symbolic logic program, the concepts of lists and list processing as a

†See Ref. 43 for what they looked like.

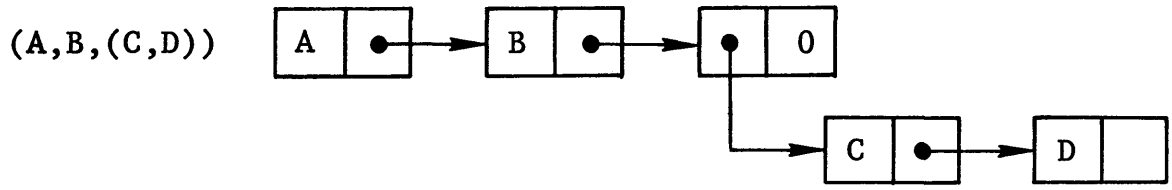
more general substratum had already developed. IPL-II was coded for JOHNNIAC, a RAND computer of the Princeton class.[39] IPL-III[7] was a version that attempted to reduce list processing to its ultimate simplicity, but it required too much space for JOHNNIAC, which has only 4096 words of core storage, and it was abandoned shortly after it became operational. IPL-IV is a list language, very similar to IPL-V, the language described in this Manual. It is coded for JOHNNIAC, and much of the subsequent work in heuristic programs has been done on it, including a chess program,[41] a program for a problem in management science of assigning tasks to work stations for an assembly line,[56] and a program called GPS, for General Problem Solver, that represents current efforts to simulate human behavior.[42,44] No documentation exists for IPL-IV. In a recent paper,[52] the design of a command structure for list processing is considered; this language is called IPL-VI, although it has never progressed beyond the design stage.

The work with IPL-V so far is mostly concerned with the simulation of human behavior. E. Feigenbaum has developed a program, EPAM, for memorizing nonsense syllables and making simple discriminations.[15] J. Feldman has developed a program to explain the behavior of humans in a binary choice situation, where the subject is required to guess which of two symbols (say + or -) will occur at each step in a long sequence.[18] Both of these tasks have received much attention in the psychological literature.

H. Gelernter at IBM has developed a program for proving theorems in plane geometry.[21,23] To do this he and

his colleagues developed a list-processing adjunct to FORTRAN for the IBM 704, called FLPL, for FORTRAN List Processing Language.(22) This was done by adding a series of subroutines to the FORTRAN system. FLPL uses the lists only for data, since it uses FORTRAN-produced machine code for routines.

J. McCarthy of MIT has developed another list language for the 704, called LISP, for List Processor.(34) LISP, like IPL-V, uses lists for both routines and data. Externally, LISP uses a horizontal notation in which list structures are represented with the aid of parentheses. The identity of parenthetical notation and list structures can be seen from the following figure:



Part of the reason for the development of LISP is McCarthy's own work, with M. Minsky, on heuristic programs.(35) There has also been some work done in LISP on analytic differentiation and integration of elementary functions.

It is also possible to add list-processing subroutines to standard assembly systems, so that coding is done in standard machine code, but the basic list operations are available as processing units. No such system is documented yet, but one for the 704 is under way.

The work of Newell, Shaw, and Simon exhibits a bias toward approaching complex programs by means of symbol-

manipulating languages, and this attitude is shared by most of the others mentioned so far. However, a number of ambitious programs have been written without benefit of this intermediate stage. Included here are the chess programs of Bernstein[5] and of Wells and others at Los Alamos;[29] the checker program of Samuel;[49] and several programs for proving theorems in symbolic logic--e.g., Dunham,[13] Gilmore,[24] and Wang.[57] The work of Kemeny and others at Dartmouth on analytic differentiation and integration[28] and of Barnett at MIT[3] belong in between those who have worked directly in machine code, and those who have developed rather complete list languages.

Additional Sources of Programming Innovation

The two sources of innovation for increasing the symbolic capabilities of computers that have been mentioned--the writing of programs, and the attempts to create intelligent programs--have had a particularly strong effect on IPL-V. There are other sources that have also contributed to the current state of the general programming technique. The attempts to use computers for business data processing have focused attention on the unit of data, and have led toward variable-word-length machines (like the IBM 702-705-7080) and toward variable-field operations, rather than toward systems that allow for the dynamic modification of the structure of information in the memory. These attempts have also focused on the operations of merging and sorting masses of data, processes which have received scant attention in heuristic programming.

The work on machine translation from one natural language to another is a field that is similar to data processing in those parts that concentrate on the dictionary problem, and similar to heuristic programming in those parts that concentrate on the use of multiple, complex rules for resolving ambiguities and analyzing sentences. A programming language, called COMIT, has been developed by V. Yngve of MIT for the IBM 704 in order to express translation algorithms easily.(60) There is a great deal of similarity in underlying structure between COMIT and the list languages.

The assembly programs and compilers, which have been the chief solution to most of the program-writing problems, are themselves complex symbol-manipulation programs. A great deal of work has gone into their development and a great deal of programming know-how has resulted. However, to date no innovations as specific as lists or variable-field operations have resulted from the process of coding these programs (as opposed to the programs themselves, which are the means for the major programming innovations).

INTRODUCTION TO THE SECOND EDITION

The continued use of IPL-V and the necessity of a reprinting of the Manual present an opportunity to incorporate some of the additions to the language and to thoroughly rewrite Part One. A few words here will serve to relate this second edition to the first, and to bring some of the comments in the original introduction up to date.

IPL-V has remained essentially constant since the first edition of the Manual. By deliberate design no attempt has been made to produce a general revision of the language. A language is to be used, and it must remain constant so that projects undertaken in its terms can cumulate. Thus, the conventions and definitions described in the initial Manual have been changed at only six minor points, all associated with loading and monitoring. W14 and W15 have had their monitoring function slightly redefined; J166 has been modified to set H5; an error in specifying the conventions on Block Reservation Cards has been corrected; loading routines into auxiliary storage has been generalized slightly; a first card (TYPE = 9) has been added to the loading sequence; and returning unused regionals to available space has been divorced from the loader and placed in a primitive (J171). The exact changes made are listed in Sec. 25 of Part Two of the revised Manual. This revision of the Manual implies no changes in the running of existing IPL-V programs.

Since the Manual was first published several additions to IPL-V have been made, providing facilities not available in the original. One is a set of routines (J180's) for reading lines of input, analogous to the existing set for printing lines. A second is a set of routines (J190's) for manipulating the subfields (P, Q, SYMB, and LINK) in an IPL word. A third is the ability to read and write blocks of contiguous cells onto tape. This facility is needed to handle programs that greatly exceed the size of core. The various parts of the existing system that formed blocks of storage, such as regions, auxiliary buffers, etc., have been fitted into the same framework, so that they can be manipulated with the block handling routines (the J170's). With this has come the ability to obtain the region (in the form of its block control word) given any regional symbol (J175). All of these extensions imply no modification of current programs; they are new facilities to be exploited if the programmer desires. They are also listed in Sec. 25 of Part Two.

More and more the programmer has wanted to get under his control all of the features of the running system; as a consequence, several changes have been made to accomplish this. For example, the post mortem has been made into a primitive (J202); also, the various constants and routines involved in compacting auxiliary have been made available in systems cells and primitive routines. Again, these do not affect current operation. They are listed in Sec. 25.

The final item about the language itself is really a suggestion. In order to avoid conflicts in name assignments, it is suggested (in § 4.1, SYSTEM REGIONS) that all installations use the $ region for installation-wide processes, cells, constants, etc.

Part Two, which is the Reference Manual, has been revised only to the extent of incorporating the changes and additions smoothly into the text and correcting various misprints. As mentioned above, all changes and extensions are listed in Sec. 25, which gives references to where in the Manual these changes are actually incorporated. There should be little trouble spotting the new material.

Part One, which is the introduction to IPL for those who are trying to learn the language, has been largely rewritten. The impetus was the need for exercises; once a set of these was on hand, extensive revision was inevitable. The resulting text still contains much of the same material as the original, with the notable exception of the use of Ackermann's function as an initial example (it can now be found as Problem 45, p. 47), and the program on the organism, which was used to illustrate the programming and coding of a complex problem. This latter was omitted because the new material claimed the space, and because it was felt that students rarely worked their way through it in enough detail to get much out of it. (At least, this was the experience in several courses which used the Manual.)

The problems themselves have been used in a Summer Training Institute on Simulation of Cognitive Processes

held at The RAND Corporation during the summer of 1963 and sponsored by RAND and the Social Science Research Council under a grant from the National Science Foundation. Solutions to a selected subset of the problems are given at the end of Part One.

Two additional teaching aids were used at the Summer Institute. One of these was an IPL-V version of the Logic Theorist (LT), a heuristic program for proving theorems in elementary symbolic logic. LT, which was originally coded in a different language in 1956, was recoded in IPL-V by Fred Tonge and then carefully worked into a demonstration problem by Einar Stefferud. The result is a large, complex program which has been documented from the point of view of someone trying to learn the ingredients of complex programs. Because of its size, the LT demonstration program could not be incorporated in the revised Manual, but information on how to get it can be obtained by writing The RAND Corporation.(54)

The second teaching aid used is a program called TIPL (Teach IPL). This program, developed by R. A. Dupchak, takes as input the IPL code proposed by the student as a solution to a problem and tests it against various sets of inputs to see if it gives the correct output. It then prints out appropriate information about whether the student's program was correct or incorrect (but not why). TIPL exists as an IPL routine, so it will run on any machine that will run IPL. It can be used with Problems 10 through 75 of Part One. Information on obtaining it is also available by writing The RAND Corporation.(14)

So much for the changes in the Manual and in the language. Looking at the use of IPL-V, we find that it continues to be used primarily for research purposes. Many of the efforts noted in the first edition have been continued[†] and a number of new ones reported.[‡] Generally, these lie in the behavioral sciences or in the areas of advanced programming research; e.g., artificial intelligence, information retrieval, natural language processing, etc.

The number of computers which have IPL-V systems is now quite large (IBM 704, 709, 7090, IBM 650, Control Data G-20, Control Data 1604, Univac 1105 and 1107, Burroughs 220, Philco 2000, and AN/FSQ-32), and systems are still being added as someone who wants to use IPL becomes interested (IBM 1620 and IBM 7040/44). An IPL Secretary has been established in order to provide a minimum amount of coordination among interested parties. Anyone wanting information about what IPL systems exist and how they may be obtained can write to the IPL Secretary, The RAND Corporation, 1700 Main Street, Santa Monica, California 90604. A few published articles about the IPL-V system also exist.[††] However, no users organization has been formed in the sense in which this notion is generally understood in the computing world.

Beyond IPL itself, work on list-processing languages has continued,[‡‡] several new systems becoming available.

[†]For example, see Refs. 17, 20, 37, 45.

[‡]For example, see Refs. 2, 10, 16, 26, 27, 30, 31, 32, 53, 55.

[††]For example, see Refs. 19, 38, 46, 51.

[‡‡]For example, see Refs. 36, 61, 62.

Some of these are direct proposals for list-processing languages, analogous to LISP and IPL;[†] others use list processing as a base within a different framework, analogous to COMIT.[33] This has led to the inevitable comparisons as to which language is best. The most recent comparison, by two knowledgeable programmers, ends where most such comparisons end--"it all depends ...."[6] An earlier, more general, discussion of list processing by Green is also worth consulting.[25]

Looking still more broadly, it is apparent that list-processing notions are by now an accepted part of the programming art, to be used where and when the technical devices pay off in comparison with alternative ways of organizing programs.[‡] The time is not too distant when these capabilities will be available as an integrated part of our main programming languages.

[†] See Refs. 11, 58, 59.

[‡] A scattering of references will give the flavor: Refs. 1, 4, 8, 47, 48.

REFERENCES

1. Baecker, H. D., "Mapped List Structures," Comm. ACM, Vol. 6, No. 8, August 1963, pp. 435-438.
2. Banerji, R. B., "The Description List of Concepts," Comm. ACM, Vol. 5, No. 8, August 1962, pp. 426-432.
3. Barnett, M. P., "A FORTRAN Encoded Symbol Pattern Locator," Unpublished, 1959.
4. Berlekamp, E. R., "Program for Double-Dummy Bridge Problems--A New Strategy for Mechanical Game Playing," J. ACM, Vol. 10, No. 3, July 1963, pp. 351-364.
5. Bernstein, A., M. De V. Roberts, T. Arbuckle, and M. A. Belsky, "A Chess-Playing Program for the IBM 704," Proceedings of the Western Joint Computer Conference (1958), Institute of Radio Engineers, New York, 1959, pp. 157-159.
6. Bobrow, D. G., and Bertram Raphael, A Comparison of List-Processing Computer Languages, RM-3842-PR, The RAND Corporation, October 1963.
7. Bottenbruch, H., et al., Application of Logic to Advanced Digital Computer Programming, Summer Session Notes, University of Michigan, 1957.
8. Bowlden, H. J., "A List-Type Storage Technique for Alphanumeric Information," Comm. ACM, Vol. 6, No. 8, August 1963, pp. 433-434.
9. Carr, J., "Programming and Coding," Handbook of Automation, Computation and Control, Vol. 2, E. Grabbe, S. Ramo, and D. Wooldridge (eds.), Wiley, New York, 1959.
10. Clarkson, G.P.E., Portfolio Selection: A Simulation of Trust Investment, Prentice-Hall, Englewood Cliffs, New Jersey, 1962.
11. Cooper, D. C., and H. Whitfield, "ALP: An Autocode List-Processing Language," Comp. J., Vol. 5, No. 1, April 1962, pp. 28-32.

12. Dinneen, G. P., "Programming Pattern Recognition," Proceedings of the Western Joint Computer Conference (1955), Institute of Radio Engineers, New York, 1955, pp. 94-100.

13. Dunham, B., R. Fridshal, and G. L. Sward, "A Non-Heuristic Program for Proving Elementary Logical Theorems," Information Processing, Proceedings of the International Conference on Information Processing, 1959, UNESCO, Paris, 1960, pp. 282-285.

14. Dupchak, Robert, TIPL: Teach Information Processing Language, The RAND Corporation, RM-3879-PR, October 1963.

15. Feigenbaum, E., "The Simulation of Verbal Learning Behavior," Proceedings of the Western Joint Computer Conference (1961), Institute of Radio Engineers, New York, 1961, pp. 121-132.

16. Feigenbaum, E. A., "An Experimental Course in Simulation of Cognitive Processes," Behavioral Science, Vol. 7, No. 2, April 1962, pp. 244-256 (including several articles by members of the class).

17. Feigenbaum, E. A., and H. A. Simon, "Performance of a Reading Task by an Elementary Perceiving and Memorizing Program," Behavioral Science, Vol. 8, No. 1, January 1963, pp. 72-76.

18. Feldman, J., "Simulation of Behavior in the Binary Choice Experiment," Proceedings of the Western Joint Computer Conference (1961), Institute of Radio Engineers, New York, 1961, pp. 133-144.

19. Feldman, Julian, "TALL--A List Processor for the Philco 2000 Computer," Comm. ACM, Vol. 5, No. 9, September 1962, pp. 484-485.

20. Feldman, J., F. Tonge, and H. Kanter, Empirical Explorations of a Hypothesis-Testing Model of Binary Choice Behavior, System Development Corporation, SP-546, December 1961.

21. Gelernter, H., "Realization of a Geometry Theorem Proving Machine," Information Processing, Proceedings of the International Conference on Information Processing, 1959, UNESCO, Paris, 1960, pp. 273-282.

22. Gelernter, H., J. R. Hansen, and C. L. Gerberich, "A FORTRAN-Compiled List-Processing Language," J. ACM, Vol. 7, No. 2, April 1960, pp. 87-101.

23. Gelernter, H., and N. Rochester, "Intelligent Behavior in Problem-Solving Machines," IBM J. Res. & Develop., Vol. 2, No. 4, October 1958, pp. 336-345.

24. Gilmore, P. C., "A Program for the Production from Axioms, of Proofs for Theorems Derivable Within the First Order Predicate Calculus," Information Processing, Proceedings of the International Conference on Information Processing, 1959, UNESCO, Paris, 1960, pp. 265-273.

25. Green, B. F., Jr., "Computer Languages for Symbol Manipulation," IRE Trans. on Human Factors in Electronics, Vol. HFE-2, No. 1, March 1961, pp. 3-8; Reprinted in IRE Trans. on Electronic Computers, Vol. EC-10, No. 4, December 1961, pp. 729-735.

26. Gullahorn, J. T., and J. E. Gullahorn, "A Computer Model of Elementary Social Behavior," Behavioral Science, Vol. 8, No. 4, October 1963.

27. Hormann, A. M., "Programs for Machine Learning," Info. & Control, Vol. 5, No. 4, December 1962, pp. 347-367.

28. Kemeny, J., et al., Symbolic Work on High Speed Computers, Dartmouth Mathematics Project Report No. 4, June 1959.

29. Kister, J., et al., "Experiments in Chess," J. ACM, Vol. 4, No. 2, April 1957, pp. 174-177.

30. Kochen, M., "Adaptive Mechanisms in Digital 'Concept' Processing," Discrete Adaptive Processes--Symposium and Panel Discussion, AIEE, New York, 1962, pp. 50-58.

31. Laughery, K. R., and L. W. Gregg, "Simulation of Human Problem-Solving Behavior," Psychometrika, Vol. 27, No. 3, September 1962, pp. 297-306.

32. Lindsay, R. K., "The Reading Machine Problem," Unpublished Ph.D. dissertation, Carnegie Institute of Technology, Pittsburgh, Pennsylvania, June 1961.

33. Markowitz, H. M., B. Hausner, and H. W. Karr, SIMSCRIPT: A Simulation Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

34. McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," Communications of the ACM, Vol. 3, No. 4, April 1960, pp. 184-185.

35. McCarthy, J., "Programs with Common Sense," Mechanisation of Thought Processes, Her Majesty's Stationery Office, London, 1959, pp. 77-84.

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

37. Newell, Allen, A Guide to the General Problem-Solver Program GPS-2-2, The RAND Corporation, RM-3337-PR, February 1963.

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

39. Newell, A., and J. C. Shaw, "Programming the Logic Theory Machine," Proceedings of the Western Joint Computer Conference (1957), Institute of Radio Engineers, New York, 1957, pp. 230-240.

40. Newell, A., J. C. Shaw, and H. A. Simon, "Empirical Explorations of the Logic Theory Machine: A Case Study in Heuristics," Proceedings of the Western Joint Computer Conference (1957), Institute of Radio Engineers, New York, 1957, pp. 218-230.

41. Newell, A., J. C. Shaw, and H. A. Simon, "Chess Playing Programs and the Problem of Complexity," IBM J. Res. & Develop., Vol 2, No. 4, October 1958, pp. 320-335.

42. Newell, A., J. C. Shaw, and H. A. Simon, "Report on a General Problem-Solving Program," Information Processing, Proceedings of the International Conference on Information Processing, 1959, UNESCO, Paris, 1960, pp. 256-264.

43. Newell, A., and H. A. Simon, "The Logic Theory Machine: A Complex Information Processing System," IRE Trans. Info. Theory, Vol. IT-2, No. 3, September 1956, pp. 61-79.

44. Newell, A., and H. A. Simon, "The Simulation of Human Thought," Current Trends in Psychological Theory, University of Pittsburgh, 1961, pp. 152-179.

45. Newell, A., and H. A. Simon, "GPS, A Program that Simulates Human Thought," Lernende Automaten, H. Billings (ed.), Oldenbourg, Munich, 1961.

46. Newell, Allen, and F. Tonge, "An Introduction to Information Processing Language-V," Comm. ACM, Vol. 3, No. 4, April 1960, pp. 205-211.

47. Perlis, A. J., and R. Iturriaga, "An Extension to ALGOL for Manipulating Formulae," Proceedings of ACM Working Conference on Mechanical Language Structures, August 1963 (to be published in Comm. ACM).

48. Reiss, R. F., "The Digital Simulation of Neuro-Musclar Organisms," Behavioral Science, Vol. 5, No. 4, October 1960, pp. 343-358.

49. Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers," IBM J. Res. & Develop., Vol. 3, No. 3, July 1959, pp. 210-229.

50. Selfridge, O. G., "Pattern Recognition and Modern Computers," Proceedings of the Western Joint Computer Conference (1955), Institute of Radio Engineers, New York, 1955, pp. 91-93.

51. Shaffer, S. S., "Current Status of IPL-V for the Philco 2000 Computer (June 1962)," Comm. ACM, Vol. 5, No. 9, September 1962, p. 479.

52. Shaw, J. C., A. Newell, H. A. Simon, and T. O. Ellis, "A Command Structure for Complex Information Processing," Proceedings of the Western Joint Computer Conference (1958), Institute of Radio Engineers, New York, 1959, pp. 119-128.

53. Simon, H. A., "Experiments with a Heuristic Compiler," J. ACM, Vol. 10, No. 4, October 1963, pp. 493-506.

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

55. Stone, P. J., and E. B. Hunt, "A Computer Approach to Content Analysis: Studies Using the General Inquirer System," 1963 Spring Joint Computer Conference, Spartan, Baltimore, Maryland, 1963, pp. 241-256.

56. Tonge, F. M., "Summary of a Heuristic Line Balancing Procedure," Management Science, Vol. 7, No. 1, October 1960, pp. 21-42.

57. Wang, H., "Toward Mechanical Mathematics," IBM J. Res. & Develop., Vol. 4, No. 1, January 1960, pp. 2-22.

58. Weizenbaum, J., "Knotted List Structures," Comm. ACM, Vol. 5, No. 3, March 1962, pp. 161-165.

59. Weizenbaum, J., "Symmetric List Processor," Comm. ACM, Vol. 6, No. 9, September 1963, pp. 524-544.

60. Yngve, V. H., "A Programming Language for Mechanical Translation," Mechanical Translation, Vol. 5, No. 1, July 1958, pp. 25-41.

61. Yngve, V., COMIT Reference Manual, MIT Press, Cambridge, Massachusetts, 1962.

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

Part One

THE ELEMENTS OF IPL PROGRAMMING

1.0 LISTS AND ROUTINES

In IPL, everything is written in the form of lists: programs are lists of instructions; data are lists of symbols. Lists are written vertically on the coding sheet, as shown below. The name of the list is put in the NAME field and the symbols on the list are written in sequence down the sheet in the SYMB field. For example, if we wish to put the symbols, S1, S3, S4, on a data list, and to name the list L1, we write:

NAME	PQ	SYMB	LINK	COMMENTS
L1		0		
		S1		
		S3		
		S4	0	

Symbols in IPL-V are written as a character followed by up to four digits.

Notice that the first symbol of the list is written on the line below the line where the name is written; also, that a termination symbol, 0, is written in the LINK field of the last line of the list. The list, L1, when it is stored inside the IPL computer will consist of four cells, corresponding to the four lines on the coding sheet; the address of the first cell will be L1. (More precisely, since L1 is a relative address, the first cell will be at the address into which the symbol L1 is translated when the list is loaded into the computer.) The addresses of the other cells are not specified by the coder, but after loading, the LINK of each cell in the list will contain the address of the following cell in the list. The contents of each cell of the list, therefore, consists of four parts: P and Q, the prefixes; SYMB, the symbol in the cell; and LINK, the name of the next cell on the list. On the coding sheet, the NAME, SYMB, and LINK fields all hold IPL symbols which become addresses inside the computer.

After loading, data lists can be processed by means of routines. A routine has a name, say R1, and has inputs and outputs. Suppose that R1 is a routine that will find the last symbol on a list. This means that R1 takes as input the name of a list (say L1, the list we just discussed), and provides as output the last symbol on the list (in this case, S4). Suppose, further, that we wanted to store the last symbol of L1 in a cell, W0. Then on the coding sheet we would write:

NAME	PQ	SYMB	LINK	COMMENTS
	10	L1		Input name of list, L1.
		R1		Find last symbol on list.
	20	W0		Output last symbol to W0.

Each line on this coding sheet is an instruction. The first instruction, 10L1, makes the symbol L1 the input to the process. The code, P = 1, indicates that the symbol is an input, and the code, Q = 0, indicates that the symbol we are designating is L1. The second instruction is really 00R1, but we do not need to write the zeros in the PQ field. P = 0 indicates that a routine is to be executed, and again, Q = 0 shows that the name of the routine is R1. Finally, the instruction 20W0 puts the output of R1--the symbol that R1 has found at the end of list L1--into cell W0. The P = 2 indicates that the output symbol is to be put in a specified cell, and the Q = 0 shows that the name of the cell is W0.

In many cases we do not know the name of the list directly, but only know what cell the name is in. In fact, indirectness of reference is the rule rather than the exception, since the lists we wish to process usually will themselves have been found as the result of prior processing. Suppose, for example, we know that the last symbol on list L1 is itself the name of another list (that is, the list S4), and that we wish to put the last symbol on

this list in W1. To do this, we do not need to know the name, S4, as long as we know L1. For, after executing the program described above, we will have the symbol S4 in cell W0 (although we do not know what this symbol is; only where it is) and can then proceed as follows:

```
NAME   PQ SYMB   LINK   COMMENTS
       11 W0           Input 1W0.
          R1           Find last symbol on list.
       20 W1           Output to W1.
```

The first instruction, 11W0, again provides an input symbol, but the Q = 1 indicates that the input is the symbol contained in W0, and not the symbol W0 itself, as before. Once we have provided the input symbol (it would be S4 in our example), we can proceed by executing R1 (00R1) and putting the result in W1 (20W1).

In this example, the symbol in W0 is only an intermediate product, and we might have no need for it except to obtain the final result in W1. Since inputs and outputs of all routines are held in the same cell, this symbol does not have to be moved out and back in again. Instead, we simply execute the second R1 on the output of the first:

```
NAME   PQ SYMB   LINK   H0     COMMENTS
       10 L1           0      Input L1.
          R1           L1     Find last symbol of L1.
          R1           S4     Find last symbol of
                              sublist (say, B2).
       20 W1           B2     Output to W1.
                       0
```

This is an essential point: Each routine finds its inputs and places its outputs in the same fixed place. This fixed cell is called the communication cell, and its name is H0. (The symbols naming IPL-V system cells all have the character H or W.) In the coding above, we have shown in the column labeled H0 the symbol in H0 just before execution of the instruction. The 0 means that

HO is empty. Thus, the second R1 was able to take as input the output of the first R1--i.e., S4--without having to move or adapt output to input. In terms of the communication cell, HO, we can state more precisely what the first three P codes mean:

If P = 0, Execute routine S (the meaning of S will be explained below);

P = 1, Put S in HO;

P = 2, Put the symbol now in HO in the cell S.

We use S to stand for the symbol that P operates on. We have seen from the examples already given that S depends on the code in Q. S is called the designated symbol of the instruction. It is obtained by the operation of Q on the symbol in the SYMB field of the instruction:

If Q = 0, Take SYMB as S;

Q = 1, Take the symbol in the cell named SYMB as S;

Q = 2, Take the symbol in the cell whose name is in the cell named SYMB as S.

We will often indicate a designated symbol in the text of this manual by writing the Q code, followed by a symbol; e.g., 1WO to indicate the symbol in WO. Thus, to illustrate the range of designation possible, in the two cells shown below, OWO is WO, 1WO is S5, and 2WO is S6. Since 1S5 is also S6, 2WO is identical with 1S5.

NAME	PQ	SYMB	LINK	COMMENTS
WO		S5	0	
S5		S6	0	

A routine can have more than one input or output--in fact, it can have any number of either. The routine J64, for instance, inserts a symbol on a list after another symbol. (J64 is a basic IPL-V process. All basic processes are named by symbols beginning with J.) J64 has two inputs: the symbol to be inserted, and the location in the

list after which it is to be inserted. We need names for talking about multiple inputs and outputs, and shall use (0) to stand for the first input, (1) to stand for the second, and, generally, (n) to stand for the n+1 input. Thus, we would define J64 more formally as:

J64: Insert (0) after the list cell named (1).

To illustrate, suppose we want to insert K5 in L1, changing the original L1 into:

NAME	PQ	SYMB	LINK	COMMENTS
L1		0		
		K5		
		S1		
		S3		
		S4	0	

A comparison with the original L1 shows that we can make this change by executing J64 with (0) = K5 and (1) = L1. On the coding sheet we write:

NAME	PQ	SYMB	LINK	HO	COMMENTS
	10	L1		0	Input L1.
	10	K5		L1	Input K5.
		J64		K5	Insert K5 on L1.
				0	

Two symbols are put successively into HO; hence, HO must be capable of holding not just a single symbol, but a whole stack of symbols (since it is permissible to define routines having any number of inputs). In IPL, cells capable of holding stacks of symbols are called storage cells, or push down cells. A storage cell is constructed like a cafeteria well for holding plates--as each new plate is put in the top of the well, all the others are pushed down, so that the only apparent change is that a new plate sits on top. When this top plate is removed, the one just below it "pops up" and becomes the new plate on top. The input and output operations treat HO as a storage cell. In the program written just above, 10L1 put L1 into HO

and pushed down the symbol that had been in HO. Then 10K5 put K5 in HO after pushing down L1 (so that only K5 is seen in HO, above). If we refer back to our notation for inputs, we see that (0) is the symbol in HO itself (at the top of the stack), (1) is the symbol beneath (0), (2) is the symbol just beneath (1), and so on.

Suppose we do not yet have the routine R1, and need to create it. The code for R1 will be the list of instructions whose name is R1. We construct a list of instructions, with name R1, that will carry out the required processing, provided that the inputs are in HO at the outset, and that will put the outputs in HO when it is complete. To code R1 we need a basic operation, J60, that allows us to move down a list in order to examine the contents of each cell in sequence:

J60: Locate the next symbol after cell (0).

The output of J60 (placed as (0)) will be the name of a cell. Suppose we write L1 again, giving names to all the cells (these are assigned automatically on loading) to show the links explicitly:

NAME	PQ	SYMB	LINK	COMMENTS
L1		0	9-1	(The symbols 9-1, 9-2,
9-1		S1	9-2	and 9-3 are called
9-2		S3	9-3	local symbols.)
9-3		S4	0	

If the input to J60 is L1, then the output will be 9-1, the name of the next cell on the list. If the input is 9-1, then the output will be 9-2. Thus, repeated applications of J60 provide access to the successive symbols on a list. But what if 9-3 is the input to J60? A standard convention is needed to recognize the end of a list, and to communicate this information to the coder. IPL uses a special cell, H5, called the test cell, for this purpose. The cell H5 can contain either of two symbols, plus (+) or minus (-). In executing J60, a + would be placed in

H5 if the next cell existed, and a - would be placed in H5 if the next cell didn't exist (i.e., if the LINK of the input to J60 were 0). Thus, the modified definition of J60 is:

> J60: Locate the next symbol after cell (0). Set H5+ if it exists; if not, set H5- and leave (0) unchanged.

Now we can write the routine, R1, for finding the last symbol on a list:

NAME	PQ	SYMB	LINK	COMMENTS
R1		J60	9-5	Locate next symbol.
9-5	70	9-7	R1	Repeat unless end of list.
9-7	52	HO	0	Input last symbol on list.

Let us trace this routine through, using L1 as the input to J60:

H5	NAME	PQ	SYMB	LINK	HO	COMMENTS
+	R1		J60	9-5	L1	Input (0) = L1.
+	9-5	70	9-7	R1	9-1	The first list cell is 9-1.
+	R1		J60	9-5	9-1	Since H5+, recycles.
+	9-5	70	9-7	R1	9-2	The second list cell is 9-2.
+	R1		J60	9-5	9-2	Since H5+, recycles.
+	9-5	70	9-7	R1	9-3	The third list cell is 9-3.
+	R1		J60	9-5	9-3	Since H5+, recycles.
-	9-5	70	9-7	R1	9-3	H5 set - , because no more cells on list.
-	9-7	52	HO	0	9-3	Branches to 9-7, because H5-.
					S4	Produces output.

This trace shows all the steps in carrying out R1. J60 replaces the name of a cell on the list with the name of the next cell. It sets H5+ if the next cell exists, which is the case until (0) becomes 9-3. The next instruction, 70 9-7, has a P-prefix, 7. This prefix transfers control conditionally upon the symbol in H5: If H5 is + , then the name of the next instruction to be performed is in LINK (as usual). If H5 is - , however, the name of the next instruction--in this case 9-7--is the designated symbol, S. In our example, R1 is in LINK of the transfer

instruction, so the routine loops back as long as H5 is + . When H5 does become - , the end of the list has been reached, and the last location is stored in HO. Then the instruction named by the designated symbol, 9-7, is executed. To complete R1, this final instruction must replace the name of the final list cell (9-3) by the name of the symbol it contains (here, S4). P = 5 is a prefix that does just this: It replaces the symbol in HO by the designated symbol, S. It is like P = 1, the input prefix, except that it does not push down HO, but destroys the symbol previously in HO. In the present example, 2HO designates the symbol we want (the symbol S4 in the cell 9-3, whose name is in the cell named HO), and 52HO produces the required output. We terminate the routine like any list, with the termination symbol for LINK.

Consider a similar routine:

J77: TEST if (0) is on list (1). Set H5+ if it is; set H5- if it isn't.

Here the result is simply a yes or no, and the test cell is used to communicate this result. The P = 7 prefix makes it easy to convert this signal into a conditional transfer of control. To accomplish J77 we need J2, the test of identity between two symbols:

J2: TEST if (0) $\equiv$ (1).

(We can omit the statements about H5 in the definition of this routine, since all TEST routines set H5+ if they are satisfied, and set H5- otherwise. Likewise, it is understood from the definition of TEST that J2 will remove both (0) and (1) from HO.)

To do J77, we shall iterate through the list, testing if each symbol on the list is the desired symbol:

```
NAME    PQ  SYMB    LINK    COMMENTS
J77     40  WO              Push down WO.
        20  WO              Output test symbol.
9-2         J60             Locate next list symbol.
        70  9-1             Transfer at end of list.
        12  HO              Input list symbol.
        11  WO              Input test symbol.
            J2              Test for identity.
        70  9-2             Iterate if unequal.
9-1     30  WO              Pop up WO.
        30  HO      0       Discard cell name in HO.
```

The first problem we face in coding J77 is the need for a working cell to hold the symbol (0). We choose the cell WO, but some other routine may already be using WO for a different purpose, and have a symbol in it. If we simply perform 20WO, we will destroy this information. WO is to be used as a public working cell, and to avoid loss of information it is declared safe--that is, any routine that uses WO must take care not to destroy the information already there. Our interpretive system is so arranged that the only routines that could have symbols in WO when J77 goes to use it are the routines "above" J77--that is, the routines that are using J77 as a subroutine (as J77 is using J60 as a subroutine). Hence, it is all right to use WO within J77 as long as the symbols that are found in WO and its push down list at the beginning of execution of J77 are left unchanged at the end. WO and the other public working cells are push down cells, so that we can save the symbol in WO simply by pushing it down prior to doing 20WO. P = 4 is the operation that does this: It pushes down the cell whose name is the designated symbol (here, OWO). This operation leaves the same symbol in WO, but also puts a copy right beneath it on WO's push down list. Now, although the symbol in WO is destroyed when 20WO occurs, the copy right beneath it is still there, and when a pop up occurs--the 30WO at the end of the routine--this original symbol is returned to the working

cell (W0). The rest of the routine J77 is straightforward: Having put (0) in W0, the name of the list, formerly (1), is now (0) and we begin a loop, using J60 to advance down the list and J2 to test if the symbol in the list cell is the desired symbol, 1W0. At each passage around the loop, the input (1) to J2 is the symbol in the list cell whose name has been placed in H0 by J60. This symbol is 2H0, since H0 holds the name of the cell. The input (0) to J2 is the symbol we wish to test for, which is designated 1W0. The latter is brought into H0 by the operation 11W0. Notice that after the test has been performed, both of these inputs have been consumed and the name of the list cell is again in H0, ready for the next operation of J60. Finally, we do a 30H0 to discard the list cell name left as (0) from our last passage around the loop.

By now we have introduced a number of P and Q operations. The only P prefix we have not yet introduced is P = 6, which, like P = 2, stores (0) in cell S, but does not pop up H0. We have also defined several processes. All processes whose names begin with J form a basic set from which all other processes are constructed. The J's and their complete definitions are given in Part Two. For the first few examples in Part One, we will explain each of the new J's we use; but later, as we proceed to more complex examples, we will assume an acquaintance with Part Two.

1.1 PROBLEMS

PROBLEM 1

Code the list named X1 which represents the days of the week in the order in which they occur, with Sunday being the first on the list. Use the symbol D1 for Sunday, D2 for Monday, etc.

PROBLEM 2

Code the list named X2 which represents the phrase, THE MORE THE MERRIER. Use the symbol T1 to represent the word THE, the symbol M1 for MORE, and the symbol M2 for MERRIER. The first word of the phrase should be first on the list.

PROBLEM 3

Code the list named X3 which represents stops on the route of a commercial airline flight originating at New York, stopping at Chicago and Denver, and terminating at Los Angeles. Invent symbols to represent the four cities, and show the correspondence between cities and symbols in the COMMENTS field of the coding sheet. The list should be ordered, with New York appearing as the first city on the list, Los Angeles as the last.

PROBLEM 4

Code the list named X4 which represents the string A+B=A+C. Represent the character A by the IPL-V symbol A0 (A-zero), the character + by the symbol +0, the character = by the symbol =0, etc.

PROBLEM 5

The examples below show the effect of P = 1 through P = 6 for Q = 0, Q = 1, and Q = 2. The lists and cells involved in the instruction are shown both before and after the execution and the symbol designated by the Q code is displayed as S. There is an obvious error in one of the examples after 5F. You should be able to spot it.

5A

Instruction P	Q	SYMB	Before Execution NAME	SYMB	LINK	S	After Execution NAME	SYMB	LINK
1	0	X1	HO	X2	0	X1	HO	X1	
								X2	0
Input		X1	X1	X3	0		X1	X3	0

5B

1	1	X1	HO	X2	0	X3	HO	X3	
								X2	0
Input		1X1	X1	X3	0		X1	X3	0
			X3	Y1	0		X3	Y1	0

5C

Instruction P Q SYMB	Before Execution NAME	SYMB	LINK	S	After Execution NAME	SYMB	LINK
1 2 X1	HO	X2	0	Y1	HO	Y1	
						X2	0
Input 2X1	X1	X3	0		X1	X3	0
	X3	Y1	0		X3	Y1	0
	Y1	Y2	0		Y1	Y2	0

5D

2 0 X1	HO	Y1		X1	HO	X2	0
		X2	0				
Output to X1	X1	X3	0		X1	Y1	0

5E

2 1 X1	HO	Y1		X3	HO	X2	0
		X2	0				
Output	X1	X3	0		X1	X3	0
to 1X1	X3	A1	0		X3	Y1	0

5F

2 2 X1	HO	Y1	0	A1	HO	0	0
	X1	X3	0		X1	X3	0
Output	X3	A1	0		X3	A1	0
to 2X1	A1	B4	0		A1	Y1	0

5G

3 0 X1	X1	X3		X1	X1	X4	0
		X4	0				
Restore X1							

5H

3 1 X1	X1	X3		X3	X1	X3	
		X4	0			X4	0
Restore 1X1	X3	X5			X3	X6	0
		X6	0				

5I

3 2 X1	X1	X3		X5	X1	X3	
		X4	0			X4	0
Restore 2X1	X3	X5			X3	X5	
		X6	0			X6	0
	X5	X7			X5	X8	0
		X8	0				

	Instruction P Q SYMB	Before Execution NAME	SYMB	LINK	S	After Execution NAME	SYMB	LINK
5J	4 0 X1 Preserve X1	X1	X3 X4	 0	X1	X1	X3 X3 X4	 0
5K	4 1 X1 Preserve 1X1	X1 X3	X3 X4 X5 X6	 0 0	X3	X1 X3	X3 X4 X5 X5 X6	 0 0
5L	4 2 X1 Preserve 2X1	X1 X3 X5	X3 X5 X7	0 0 0	X5	X1 X3 X5	X3 X5 X7 X7	0 0 0
5M	5 0 X1 Replace with X1	HO X1	Y4 Y2 X3	 0 0	X1	HO X1	X1 Y2 X3	 0 0
5N	5 1 X1 Replace with 1X1	HO X1 X3	Y4 Y2 X3 Y1	 0 0 0	X3	HO X1 X3	X3 Y2 X3 Y1	 0 0 0
5P	5 2 X1 Replace with 2X1	HO X1 X3	X1 X3 X5	0 0 0	X3	HO X1 X3	X3 X3 X5	0 0 0
5Q	6 0 X1 Store in X1 or Copy to X1	HO X1	Y4 Y5 X3	 0 0	X1	HO X1	Y4 Y5 Y4	 0 0

5R

Instruction P Q SYMB	Before Execution NAME	SYMB	LINK	S	After Execution NAME	SYMB	LINK
6 1 X1	HO	Y4		X3	HO	Y4	
		Y5	0			Y5	0
Store in 1X1	X1	X3	0		X1	X3	0
	X3	X5	0		X3	Y4	0

5S

6 2 X1	HO	Y4		X5	HO	Y4	
		Y5	0			Y5	0
Store in 2X1	X1	X3	0		X1	X3	0
	X3	X5	0		X3	X5	0
	X5	X6	0		X5	Y4	0

Show the state of the lists in Problems 5A through 5S after the second execution of each instruction. You need show only those lists which are changed by the second execution of the instruction in each example. Label the solutions 5A, 5B, etc.

PROBLEM 6

For each of the problems, 5A through 5S, code the instruction or instructions needed to return the lists from the "After Execution" state shown in the problem, to the "Before Execution" state. Label the solutions 6A, 6B, etc.

2.0 A COMPLETE PROGRAM

We can now take a simple problem and carry it all the way through to assembly and execution on the computer. The routines and data are shown on a facsimile of the coding sheet to illustrate exactly how a deck suitable for running would look. An assembly listing of the complete program and a trace of its execution is also included.

Suppose we have a number of lists like K1 below. Each list has an unknown number of distinct symbols on it, each distinct symbol occurring on the list any number of times and in any order. The list K1, for example, has only three distinct symbols: A1, B1, and C1. There is one occurrence of the symbol C1, three occurrences of B1, and two of A1.

COMMENTS	TYPE	NAME	SIGN	PQ	SYMB	LINK
0000011111111112222222222333333333344445678901234567890123456789012345678901	4	4444444	4	45	5555555	5555666
						6789012
DATA HEADER. TYPE=5, Q=1. THE LIST K1.	5	K1		01	0 B1 C1 B1	
					A1 A1 B1	0

We wish to write a routine named F1 which will accept the name of a list like K1 as input and will produce as output a new list that contains only one occurrence of each distinct symbol found on the input list. For example, if F1 were to operate on the list K1 above, it would produce a new list with an internal name, say 777, that looked like this:

NAME	PQ	SYMB	LINK
777		0	
		B1	
		C1	
		A1	0

The routine F1 needs to be able to create a new list each time it is executed. The process named J90 will do this, by removing a cell from a special list of cells reserved by the system for this purpose.

> J90: Get a cell from the available space list, H2, and leave its name in H0.

The cell named in H0 is empty, containing 0 in both SYMB and LINK. The name of this cell is an internal symbol, which is simply the decimal address of the cell. This cell will be the head of the new list which F1 will build; additional list cells will be added to it as necessary by the process named J66:

> J66: Insert the symbol (0) at the end of list (1) if not already on it. If the symbol (0) already exists on list (1), J66 does nothing.

Given J66, the task of coding F1 becomes quite simple: We create an empty output list with J90 and attempt to add to it all of the symbols on the input list, using J66. J66 will refuse to put more than one occurrence of each distinct symbol onto the output list. The code for F1 is shown below; it is preceded by a header card with TYPE = 5 and Q = 0, which signals the system that the cards following the header card are to be loaded into the computer as routines. A different header must precede data lists, since the P and Q codes of data and routines are treated differently by the loading processes. Cards with TYPE = 1 are comments; these cards are printed on the assembly listing, but are otherwise ignored by the initial loader. (The comments can extend across the entire card, even though we have not done so in the example below.)

COMMENTS	TYPE	NAME	SIGN	PQ	SYMB	LINK
000000011111111112222222222333333333344 345678901234567890123456789012345678901	44 12	44444 34567	4 8	45 90	55555 12345	55566 67890	66 12
RØUTINE HEADER, TYPE=5, Q=0.	5			00			
F1---PRØDUCE AN ØUTPUT LIST (0)	1						
CØNTAINING ØNLY ØNE ØCCURRENCE ØF	1						
EACH DISTINCT SYMBØL ØN THE INPUT	1						
LIST (0). THE RØUTINE F1 IS	1						
MARKED TØ TRACE CØNDITIØNALLY WITH	1						
Q=4 IN ITS FIRST INSTRUCTIØN.	1						
CREATE AN EMPTY ØUTPUT LIST.		F1		04	J90		
PRESERVE W0.				40	W0		
W0 HØLDS NAME ØF ØUTPUT LIST.				20	W0		
LØCATE NEXT CELL OF INPUT LIST.		9-1			J60		
GØ TØ 9-2 IF NØ NEXT CELL.				70	9-2		
INPUT THE SYMBØL IN THE CELL.				12	H0		
INPUT THE NAME ØF THE ØUTPUT LIST,				11	W0		
REVERSE THEIR PØSITION IN H0, AND					J6		
ADD THE SYMBØL TØ THE ØUTPUT LIST					J66	9-1	
IF IT IS NØT ALREADY ØN IT---	1						
RETURN TØ 9-1 IN EITHER CASE.	1						
PUT THE NAME ØF THE ØUTPUT LIST		9-2		51	W0		
IN H0 AND RESTØRE W0 BEFØRE				30	W0	0	
QUITTING.	1						

The routine E1 below is a simple executive routine whose main function is to place the name of the input list K1 into H0, execute the routine F1, and print the output list produced by F1. We have marked E1 to trace unconditionally by making Q = 3 in its first instruction. As each instruction of E1 is executed, the monitor system prints out pertinent information, such as the location of the instruction, the contents of H0, the sign of H5, etc. Since F1 is marked to trace conditionally, with Q = 4, it will also trace because the routine which executes it (E1) is tracing.

COMMENTS	TYPE	NAME	SIGN	PQ	SYMB	LINK
0000000011111111112222222222333333333344	44	444444	4	45	555555	555666
3456789012345678901234567890123456789012	12	345678	9	01	234567	890123
RØUTINE HEADER. TYPE=5, Q = 0	5			00		
E1---EXECUTIVE RØUTINE.	1					
EXECUTE F1 WITH THE LIST K1	1					
AS INPUT AND PRINT THE	1					
RESULTING ØUTPUT LIST.	1					
Q=3 SAYS TRACE THIS RØUTINE.		E1		13	K1	
EXECUTE F1 AND					F1	
PRINT THE ØUTPUT LIST.					J151	0

Figure 1 is an assembly listing of the complete input deck, produced by the IPL-V system as the deck was being loaded for execution. First comes a Type-9 card, which signals the start of the program. (The various Type cards control the loading process.) This is followed by some Type-1 cards (comment cards) which describe the program but have no effect on the loading. Type-1 cards may appear anywhere in a deck. Next come several Type-2 cards which declare what regional symbols will be used: the first Type-2 card, for example, states that ten symbols beginning with the letter A may be used, A0 through A9. The decimal integers to the left show that cell 64 represents A0, 65 represents A1, etc. Next comes a Type-5 card with Q = 0, indicating that a set of routines will follow immediately. The data are preceded by a similar Type-5 card, except with Q = 1 to signal data. The assembly listing shows each card of the input deck and displays the cell NAME, P, Q, SYMB, and LINK corresponding to it. The final card of the deck is the start card, a Type-5 card with a regional SYMB, indicating, in this case, that interpretation is to begin at the routine named E1.

Notice that E1 has Q = 3 in its first instruction and that F1 has Q = 4 in its first instruction. Q = 3

```
TYPE=9, FIRST CARD.                          9
EXAMPLE RUN OF F1 TO ILLUSTRATE              1
ASSEMBLY LISTING AND TRACE.                  1
THE A-REGION=10 CELLS, A0-A9.                2 A                10      64                  73
THE B-REGION=10 CELLS, B0-B9.                2 B                10      74                  83
THE C-REGION=10 CELLS, C0-C9.                2 C                10      84                  93
THE E-REGION=10 CELLS, E0-E9.                2 E                10      94                  103
THE F-REGION=10 CELLS, F0-F9.                2 F                10      104                 113
THE K-REGION=10 CELLS, K0-K9.                2 K                10      114                 123
ROUTINE HEADER.  TYPE=5, Q=0.                5          0
E1---EXECUTIVE ROUTINE.                      1
     EXECUTE F1 WITH THE LIST K1             1
     AS INPUT AND PRINT THE                  1
     RESULTING OUTPUT LIST.                  1
Q=3 SAYS TRACE THIS ROUTINE.                   E1     13K1                95      1 3 115     124
EXECUTE F1 AND                                          F1                124     0 0 105     125
PRINT THE OUTPUT LIST.                                  J151   0          125     0 0 24911   0
ROUTINE HEADER, TYPE=5, Q=0.                 5          0
F1---PRODUCE AN OUTPUT LIST (0)              1
CONTAINING ONLY ONE OCCURRENCE OF            1
EACH DISTINCT SYMBOL ON THE INPUT            1
LIST (0).  THE ROUTINE F1 IS                 1
MARKED TO TRACE CONDITIONALLY WITH           1
Q=4 IN ITS FIRST INSTRUCTION.                1
CREATE AN EMPTY OUTPUT LIST.                   F1     04J90               105     0 4 24850   126
PRESERVE W0.                                           40W0                126     4 0 24587   127
W0 HOLDS NAME OF OUTPUT LIST.                          20W0                127     2 0 24587   128
LOCATE NEXT CELL OF INPUT LIST.                9-1       J60              128     0 0 24820   129
GO TO 9-2 IF NO NEXT CELL.                             709-2               129     7 0 131     130
INPUT THE SYMBOL IN THE CELL.                          12H0                130     1 2 24574   132
INPUT THE NAME OF THE OUTPUT LIST,                     11W0                132     1 1 24587   133
REVERSE THEIR POSITION IN H0, AND                        J6                133     0 0 24766   134
ADD THE SYMBOL TO THE OUTPUT LIST                        J66    9-1        134     0 0 24826   128
IF IT IS NOT ALREADY ON IT---                1
RETURN TO 9-1 IN EITHER CASE.                1
PUT THE NAME OF THE OUTPUT LIST                9-2     51W0                131     5 1 24587   135
IN H0 AND RESTORE W0 BEFORE                            30W0     0          135     3 0 24587   0
QUITTING.                                    1
DATA HEADER.  TYPE=5, Q=1.                   5          1
THE LIST K1.                                   K1        0                115     0 4 0       136
                                                         B1                136     0 0 75      137
                                                         C1                137     0 0 85      138
                                                         B1                138     0 0 75      139
                                                         A1                139     0 0 65      140
                                                         A1                140     0 0 65      141
                                                         B1     0          141     0 0 75      0
START CARD.  EXECUTE E1.                     5          E1

END OF LOADING.  PROGRAM STARTS AT             E1     13K1     124
NUMBER OF CELLS ON AVAILABLE SPACE=21858
```

Fig. 1--Assembly Listing of F1

and Q = 4 mean the same as Q = 0 except they also cause tracing, as explained above. The trace of E1 and F1 is shown in Fig. 2. The information provided in the trace is that shown in the previous examples: H5 (before execution of the instruction), the instruction, its address (known as the CIA or Current Instruction Address), the contents of H0 (before execution of the instruction), and the contents of the cell named in H0. The trace also prints: an integer indicating the level at which the subroutine operates, indented for reading ease; the designated symbol, S; and H3, a data term that keeps a count of the number of interpretive cycles that have been performed. The computation of F1 is short enough so that every instruction executed appears in Fig. 2.

2.1 PROBLEMS

PROBLEM 7

Code the Type-2 cards defining all the regional symbols that would be needed if you were to load the lists X1, X2, and X4 at the same time. (See Problems 1, 2, and 4, pp. 12, 13, for a description of all the symbols used in these lists.)

PROBLEM 8

Code a set of Type-2 cards to define 100 symbols for each of the letters and punctuation marks. Omit H, J, W, and $, since they are specially defined by the system. Use unspecified origins for the regions. Keypunch and keep this deck of cards for use in running your future codes on the computer. It will be adequate for any problem you will be asked to code in these exercises.

PROBLEM 9

Code a new executive routine named E5 for the complete program described in this section (§ 2.0). E5 should first print the list K0, execute the routine F1 with K0 as the input list, then print the output list produced by F1. Code the list K0 also, using only symbols defined by the Type-2 cards in the sample assembly listing, and precede K0 with a Type-5 data header. Provide a Type-5 start card to start the program at the new execution routine, E5. Mark E5 for trace.

```
LEVEL   CIA                                H5 P Q SYMB  LINK    S     (0)     CONTENTS OF (0)      H3
  1     E1                                  + 1 3 K1    124           0                           5
  1     124                                 + 0 0 F1    125           K1    0 4 0       136       6
    2    F1                                 + 0 4 J90   126           K1    0 4 0       136       7
    2    126                                + 4 0 W0    127           148   0 4 0       0         9
    2    127                                + 2 0 W0    128           148   0 4 0       0         10
    2    128                                + 0 0 J60   129           K1    0 4 0       136       11
    2    129                                + 7 0 131   130           136   0 0 B1      137       13
    2    130                                + 1 2 H0    132   B1      136   0 0 B1      137       14
    2    132                                + 1 1 W0    133   148     B1    0 4 0       0         15
    2    133                                + 0 0 J6    134           148   0 4 0       0         16
    2    134                                + 0 0 J66   128           B1    0 4 0       0         18
    2    128                                + 0 0 J60   129           136   0 0 B1      137       20
    2    129                                + 7 0 131   130           137   0 0 C1      138       22
    2    130                                + 1 2 H0    132   C1      137   0 0 C1      138       23
    2    132                                + 1 1 W0    133   148     C1    0 4 0       0         24
    2    133                                + 0 0 J6    134           148   0 4 0       149       25
    2    134                                + 0 0 J66   128           C1    0 4 0       0         27
    2    128                                + 0 0 J60   129           137   0 0 C1      138       29
    2    129                                + 7 0 131   130           138   0 0 B1      139       31
    2    130                                + 1 2 H0    132   B1      138   0 0 B1      139       32
    2    132                                + 1 1 W0    133   148     B1    0 4 0       0         33
    2    133                                + 0 0 J6    134           148   0 4 0       149       34
    2    134                                + 0 0 J66   128           B1    0 4 0       0         36
    2    128                                + 0 0 J60   129           138   0 0 B1      139       38
    2    129                                + 7 0 131   130           139   0 0 A1      140       40
    2    130                                + 1 2 H0    132   A1      139   0 0 A1      140       41
    2    132                                + 1 1 W0    133   148     A1    0 4 0       0         42
    2    133                                + 0 0 J6    134           148   0 4 0       149       43
    2    134                                + 0 0 J66   128           A1    0 4 0       0         45
    2    128                                + 0 0 J60   129           139   0 0 A1      140       47
    2    129                                + 7 0 131   130           140   0 0 A1      141       49
    2    130                                + 1 2 H0    132   A1      140   0 0 A1      141       50
    2    132                                + 1 1 W0    133   148     A1    0 4 0       0         51
    2    133                                + 0 0 J6    134           148   0 4 0       149       52
    2    134                                + 0 0 J66   128           A1    0 4 0       0         54
    2    128                                + 0 0 J60   129           140   0 0 A1      141       56
    2    129                                + 7 0 131   130           141   0 0 B1      0         58
    2    130                                + 1 2 H0    132   B1      141   0 0 B1      0         59
    2    132                                + 1 1 W0    133   148     B1    0 4 0       0         60
    2    133                                + 0 0 J6    134           148   0 4 0       149       61
    2    134                                + 0 0 J66   128           B1    0 4 0       0         63
    2    128                                + 0 0 J60   129           141   0 0 B1      0         65
    2    129                                - 7 0 131   130           141   0 0 B1      0         67
    2    131                                - 5 1 W0    135   148     141   0 0 B1      0         68
    2    135                                - 3 0 W0    0             148   0 4 0       149       69
  1     125                                 - 0 0 J151  0             148   0 4 0       149       70
        148     0
                B1
                C1
                A1

PROGRAM RAN TO COMPLETION.
```

Fig. 2--Trace of E1 and F1

3.0 SIMPLE LIST PROCESSING

The heart of IPL coding is in the manipulation of lists. This section is devoted to learning how to perform these basic manipulations. It consists almost entirely of a set of problems, since the techniques introduced so far are sufficient.

A large number of basic processes which can be used in composing routines is given in Part Two. All their names begin with J, and, in fact, the letter J is reserved for officially defined processes. The routines J60 to J99 are all devoted to processing lists, and a few of these (J60, J64, J66, J77) we have already used. Although some of these J's could be coded as IPL routines (e.g., J77), they may be looked at as a complete complement of primitive list processes. From now on, free use should be made of processes listed in Part Two, except as they are specifically prohibited in some of the problems for learning purposes.

Already in coding J77 we faced the need for working storage and introduced pushing and popping as appropriate basic operations. In general, several working cells will be needed. This will require a mass push down, a mass transfer of symbols stacked in H0 to the working cell, and a mass pop up at the end. To make these housekeeping operations easy, a set of cells, W0, W1, ..., W9, are set aside as working cells, and basic J-processes are provided for pushing, popping and storing. Thus, if three working cells are needed one has available:

J22: Move (0), (1), and (2) to W0, W1, and W2, respectively.
J32: Pop up W0, W1, and W2.
J42: Push down W0, W1, and W2.
J52: Push down W0, W1, and W2; then move (0), (1), and (2) to W0, W1, and W2, respectively.

For any other number of cells, similar processes are available--e.g., for W0 through W6 there is J26, J36, J46, and J56. Collectively there are forty such routines, which may seem a great many. However, they only take fifty cells to code, and since this is an interesting coding device in IPL, it is worth illustrating. We give below the codes for J22, J32, J42, and J52:

NAME	PQ	SYMB	LINK
J22		J21	
	20	W2	0
J21		J20	
	20	W1	0
J20	20	W0	0

NAME	PQ	SYMB	LINK
J32	30	W2	J31
J31	30	W1	J30
J30	30	W0	0

NAME	PQ	SYMB	LINK
J42	40	W2	J41
J41	40	W1	J40
J40	40	W0	0

NAME	PQ	SYMB	LINK
J52		J42	J22
J51		J41	J21
J50		J40	J20

Each of these J's uses members of the same class to do part of the task. They also show that one can exit from a routine by executing a routine "from the right"--that is, by writing the name of a routine in LINK. Thus, J52 is a one-word routine: J42 J22, which reads, "Do J42, then transfer to J22." Ultimately, of course, there comes a routine which does not quit by transferring to another one, but by having a 0 in the LINK, the standard termination signal.

A number of terms have a special meaning in IPL parlance. In Sec. 10. the term TEST was introduced with a special meaning. Others, such as FIND, INSERT, DELETE, are defined in Part Two (§§ 5.0, and 9.2 through 9.7). For example, to locate a symbol on a list means to produce the name of the cell which contains the symbol. R62 is a routine which locates the first occurrence of symbol (0) on the list named by (1), sets H5+, and outputs the cell name. Otherwise, H5 is set - and the output (0) names

the location of the last symbol on the list. The code for R62, below, uses the working cell processes just introduced. One of the Q codes below is wrong; can you spot it?

NAME	PQ	SYMB	LINK	COMMENTS
R62		J51		Preserve WO and W1 before using them as working cells, and output the symbol to WO, the list to W1.
9-1	10	W1		Start (or resume) search.
		J60		Locate the cell holding next symbol.
	70	J31		Pop WO and W1 when no more symbols. Quit with H5-.
	20	W1		Save location to resume search.
	12	W1		Input the new symbol.
	11	WO		Input symbol being sought.
		J2		Test if symbols match.
	70	9-1		Resume search at 9-1 if no match.
	11	W1	J31	Location of matching symbol to HO, restore working cells, and quit with H5+.

The erroneous Q-code is at 9-1. The instruction should be 11W1, since we do not want to search the list W1 but the list whose name is in the top cell of W1.

3.1 PROBLEMS

PROBLEM 10

Recode R62 above using only one working cell. Let WO hold the input character (0) and manipulate the HO list so that it provides the storage formerly provided by W1.

PROBLEM 11

Code the routine R66--"Insert the symbol (0) at the end of list (1) if (0) is not already on list (1). Set H5+ if (0) was added to the list, set H5- if (0) already existed on the list (1)." J5, J51, J65, and J77 may be useful in coding R66. (R66 differs from the primitive J66 in that R66 sets H5; J66 does not.)

PROBLEM 12

The list named X6 consists of one occurrence each of only three symbols, A1, S1, and T1, in an unknown order. Code P32--"Print X6 as a list, then substitute the symbol B1 for the symbol A1, then delete the symbol T1 and print X6 again."

PROBLEM 13

The list named X7 contains three occurrences of the symbol A1, and three occurrences of the symbol B1, all in unknown order. Code P33--"Print X7 as a list, replace the first and second occurrences of A1 with the symbol C1, and delete the third occurrence of the symbol B1. Print X7 again."

PROBLEM 14

The list X8 contains one occurrence each of four symbols in unknown order: S1, S2, T1, T2. Code P34--"Delete S1 if S1 occurs before T1. If T1 occurs before S1, add another T1 at the end of list X8. Print X8 as a list, before and after altering it."

PROBLEM 15

Code P65--"Substitute the symbol (0) for each occurrence of symbol (1) found on the list named by the symbol (2)." Use J62.

PROBLEM 16

Code P66--"Replace the final occurrence of the symbol (1) on the list named (2) with the symbol (0), and set H5+. If the symbol (1) does not occur on the list, insert symbol (0) at the end of the list and set H5-."

PROBLEM 17

Code P67--"Test if the last symbol on the list named (1) is the symbol (0)." Remember that "test" has a technical meaning in IPL-V. A test routine's only output is the sign of H5.

PROBLEM 18

Code P68--"Insert the symbol (0) ahead of all other symbols on the list named (1)." P68 inserts symbol (0) so that it is the first symbol on the list (1), not in the head of the list (1). Only one insertion is requested.

PROBLEM 19

Code P69--"Insert the symbol (0) immediately ahead of each occurrence of the symbol (1) on the list named (2)."

PROBLEM 20

Code P70--"Insert the symbol (0) immediately after each occurrence of symbol (1) on the list named (2)."

PROBLEM 21

Code P71--"Delete all occurrences of symbol (0) from the list named (1). Set H5- if the symbol did not occur on the list, otherwise set H5+." Do not use J69.

PROBLEM 22

Code P72--"Copy the list named (1). Divide the new copy of the list after the first occurrence of the symbol (0), which is guaranteed to occur on the list. Output the name of the new copy in (0) and the name of the remainder of the copy in (1)." Use J75.

PROBLEM 23

Code P73--"Join the list named (0) to the end of the list named (1) to make one long list. Print the combined list." (The head of list (0) should not become part of the combined list; only its list cells are added to list (1).)

PROBLEM 24

Code P74--"Reverse the order of the symbols on the list named (0)." P74 should erase any lists it creates. After P74, the list may be composed of the same or different list cells, depending on the method used.

PROBLEM 25

Code P75--"Delete from the list named (1) all occurrences of each symbol on the list named (0). H5 is safe over P75." P75 must preserve H5 on input and restore it before quitting, in order to make H5 safe over the routine. Assume that a given symbol may occur only once on list (0) but may occur several times on list (1).

4.0 LIST STRUCTURES

The basic process J77 tests if a symbol is on a simple list. Suppose we complicate matters a little, and have a list of lists, such as L5, below:

```
NAME   PQ  SYMB   LINK
L5         0
           L10
           L11
           L12    0
L10        0
           S1
           S2
           S3     0
L11        0
           S4
           S5
           S6     0
L12        0
           S7
           S8
           S9     0
```

We want to find if a given symbol is on any of the three sublists, L10, L11, or L12. We can define a routine, R2, as follows:

```
R2:  Test if (0) is on any of the sublists of
     list (1).
```

Given J77, the code is almost immediate.

```
NAME   PQ  SYMB   LINK   COMMENTS
R2         J50           Output the test symbol to
                         WO, pushing down WO.
9-2        J60           Locate the next sublist.
       70  9-1           If no more sublists (H5-),
                         exit from R2 with H5-.
       12  HO            Input name of sublist.
       11  WO            Input the test symbol.
           J77           Test if symbol on sublist.
       70  9-2           If so (H5+), exit from R2
                         with H5+.
9-1    30  HO    J30     Common clean-up for both
                         exits.
```

We can create a routine similar to R2 for any structure of lists and sublists, providing that we know the exact structure--in this case, that all the symbols of L5 name sublists, and that each sublist consists entirely of symbols to be tested.

We may not always have such definite information. Suppose we have another list, say L6, which has other information on it besides the names of sublists, and some of whose sublists themselves have further sublists that require exploring. To handle this more general situation, IPL uses the list structure. L6 might appear as follows:

NAME	PQ	SYMB	LINK
L6		0	
		S1	
		9-1	
		S2	
		9-2	0
9-1		0	
		S3	
		9-3	
		9-3	
		S4	0
9-2		0	
		S5	
		S6	0
9-3		0	
		S7	
		S8	
		S9	
		S10	0

L6 is a list structure consisting of four lists, L6, 9-1, 9-2, and 9-3. The symbols beginning with 9 are local symbols (we have been using them right along), and they indicate which lists belong to the structure. Thus, although the symbol S1 may name a list, it is not part of list structure L6, because it does not occur with a local name. Local names can be detected by J132 (and also manufactured, but we do not need to do this yet):

J132: Test if (0) is a local symbol.

Hence, in L6 we can detect which of the symbols are local, and therefore require further search.

We can now define a routine that works for all list structures:

> R3: Test if (0) occurs anywhere in the list structure named (1).

The problem is that list structures are defined recursively--they permit sublists to occur on sublists ad infinitum. This means that our routine must likewise be prepared to continue its search for the symbol (0) down as many sublists as actually occur. The natural way to write such a routine is recursively, as is shown if we build a simple plan for how to code R3:

> R3: Test if (0) occurs on the main list; if it does, exit + . Otherwise, find the sublists of the main list. For each one, test if (0) occurs anywhere in the substructure.

The last line of the plan is equivalent to doing R3 on the sublist. Thus, R3 will occur as a subroutine in the code for R3--which is the formal way of defining a recursive process. There is nothing circular about using a routine to define itself, as long as there comes a point where the recursion stops. For R3, this will occur when, finally, a sublist is reached which either contains the symbol (0), or contains no further sublists. Since any list structure is finite in size (although perhaps of unknown structure), this will eventually occur. (Actually, for our code, we must assume that the sublists 9-1 and 9-2 do not mutually reference each other. This we will do, although we will soon see how to remove even this restriction.) The plan divides the task into two separate parts, but there is no reason why we should not accomplish them simultaneously as we iterate down the list. A code for this is the following:

```
NAME   PQ SYMB   LINK   COMMENTS
R3        J50    9-2    Output the test symbol to
                        WO, pushing down WO.
9-2       J60            Locate the next cell.
       70 9-1            If no more cells (H5-),
                         exit from R3 with H5-.
       12 HO
       11 WO
          J2             Test if symbol is test
                         symbol.
       70        9-1     If so (H5+), exit from R3
                         with H5+.
       12 HO
          J132           Test if symbol in cell is
                         local (the name of a sublist).
       70 9-2            If not (H5-), recycle.
       12 HO
       11 WO
          R3             If so, test if symbol is in
                         substructure (recurse).
       70 9-2            If R3 gives H5+, exit this
                         R3 with H5+; if not, recycle.
9-1    30 HO     J30     Common clean-up for all
                         exits.
```

The only new feature in this program besides the recursion is in the instruction following J2, where a blank SYMB means the name of the next cell, just as it does for LINK when LINK is left blank.

Let us trace schematically the operation of R3 finding S4 on L6 to see how a recursion is actually carried out. To help keep track of exactly which instructions are being executed and which list cells are being referenced, we arbitrarily load the routine as shown below. Starting at cell 100 is a segment of a routine preparing to use R3; R3 is loaded into cells starting at 200, and L6 is loaded into cells starting at 500, with the various sublists loading into cells at 600, 700, and 800, respectively.

NAME	PQ	SYMB	LINK
100	10	L6	101
101	10	S4	102
102		R3	103
103	70	110	104
R3		J50	200
200		J60	201
201	70	213	202
202	12	HO	203
203	11	WO	204
204		J2	205
205	70	206	213
206	12	HO	207
207		J132	208
208	70	200	209
209	12	HO	210
210	11	WO	211
211		R3	212
212	70	200	213
213	30	HO	J30

NAME	PQ	SYMB	LINK
L6		0	500
500		S1	501
501		9-1	502
502		S2	503
503		9-2	0
9-1		0	600
600		S3	601
601		9-3	602
602		9-3	603
603		S4	0
9-2		0	700
700		S5	701
701		S6	0
9-3		0	800
800		S7	801
801		S8	802
802		S9	803
803		S10	0

H1, the Current Instruction Address cell, always holds the address of the instruction being executed. It is a push down cell, just like HO and the W's. When a subroutine is to be executed, H1 is pushed down and the name of the subroutine is put in H1. The address originally in H1, which is needed to tell where to resume interpretation after the subroutine is finished, is then one-down in H1.

Figure 3 shows the course of processing. It starts with instruction 100 to be executed. By step 4 we see that subroutine R3 is to be executed on its inputs, S4 and L6, and that H1 has been pushed down. With the next step we start to execute the instructions of R3 in sequence, and J50 has pushed down WO and moved S4 to it.

The next twenty instructions, which are not shown (but which can be easily traced by following R3), involve seeing that S1 is neither S4 nor local, advancing down the list, seeing that 9-1 is not S4 but is a local sublist, and inputting S4 and 9-1 into HO. At step 25, we are ready to

Level	Step	H1	Instruction	H5	HO	WO
1	1	100	10L6	+	0	
1	2	101	10L4	+	L6	
1	3	102	R3	+	S4,L6	
2	4	R3,102	J50	+	S4,L6	
2	5	200,102	J60	+	L6	S4
		...	...		...	...
2	25	211,102	R3	+	S4,9-1,501	S4
3	26	R3,211,102	J50	+	S4,9-1,501	S4
3	27	200,211,102	J60	+	9-1,501	S4,S4
		...	...		...	...
3	47	211,211,102	R3	+	S4,9-3,601,501	S4,S4
4	48	R3,211,211,102	J50	+	S4,9-3,601,501	S4,S4
4	49	200,211,211,102	J60	+	9-3,601,501	S4,S4,S4
		...	...		...	...
4	85	200,211,211,102	J60	-	803,601,501	S4,S4,S4
4	86	201,211,211,102	70213	-	803,601,501	S4,S4,S4
4	87	213,211,211,102	30HO J30	-	803,601,501	S4,S4,S4
3	88	212,211,102	70200	-	601,501	S4,S4
3	89	200,211,102	J60	-	601,501	S4,S4
		...	...		...	...
3	100	211,211,102	R3	+	S4,9-3,602,501	S4,S4
		...	...		...	...
3	142	200,211,102	J60	-	602,501	S4,S4
3	143	201,211,102	70213	+	603,501	S4,S4
3	144	202,211,102	12HO	+	603,501	S4,S4
3	145	203,211,102	11WO	+	S4,603,501	S4,S4
3	146	204,211,102	J2	+	S4,S4,603,501	S4,S4
3	147	205,211,102	70206 213	+	603,501	S4,S4
3	148	213,211,102	30HO J30	+	603,501	S4,S4
2	149	212,102	70200 213	+	501	S4
2	150	213,102	30HO J30	+	501	S4
1	151	103	70110	+		

Fig. 3--Trace of R3

execute instruction 211, and to recurse on R3. Step 26 corresponds to step 4, except that H1 has been pushed down again, leaving 211 as the instruction from which to continue when the latest performance of R3 is over. Notice that H0 contains 501, the place in list L6 from which to continue when the processing of 9-1 is done.

By steps 47, 48, and 49, testing has progressed past S3 in sublist 9-1 to the first occurrence of sublist 9-3. Another recursion occurs, resulting in H1 being pushed down, etc.

By step 85 the last cell on list 9-3 has been processed and J60 attempts to find the next cell. Since J60 sets H5-, a transfer to instruction 213 occurs, leading to step 87 which pops H0 and W0. This is the end of the routine, so in step 88 we see that H1 has been popped, which means that the program continues in the superroutine of R3. Since we are in the midst of a recursion, the superroutine is R3 and the program continues at the next instruction after 211 (which was held in H1), namely, 212. At step 100, the system has advanced to the second occurrence of 9-3 in sublist 9-1 and another recursion of R3 has commenced. This follows the same course as the previous one, so that by step 142 instruction 200 is prepared to do J60 on 602 and process the cell which holds S4.

The final steps of the program, 144-151, start with the test J2 setting H5+. This result rapidly terminates the two occurrences of R3 that are still represented in H1. At step 151 the original R3 has been completed, H5 is + , and instruction 103 is about to be executed.

Although somewhat tedious, tracing out a recursive routine clearly demonstrates the way push down lists keep the place in the routines to be executed and in the data to be processed.

We also note that at all levels of the recursion, W0 contains the same symbol, the input (0), so that much of

the symbol shuffling is superfluous. We want a way to hold the test symbol in common for all levels of the recursion. Thus, a more elegant R3 should follow a slightly different plan:

> R3: Set up (0) in WO. Test if the symbol in WO occurs in the list structure (0) (this is a recursive routine). Clean up WO.

The coding for this is shown below. The recursive subroutine is given a local name, since its use is strictly internal to R3.

```
NAME   PQ  SYMB   LINK   COMMENTS
R3         J50           Set up (0) in WO.
           9-10   J30    Execute subroutine, then
                         clean up.
9-10       J60           9-10 tests if symbol in WO
                         occurs in list structure.
       70  J8
       12  HO
       11  WO
           J2
       70          J8
       12  HO
           J132
       70  9-10
       12  HO
           9-10          9-10 can be executed with-
                         out input from WO.
       70  9-10   J8
```

We have indeed avoided the superfluous shuffling of the test symbol.

This example illustrates rather well several aspects of subroutines, iterations, and recursions. First, the last two instructions show clearly the difference between executing a subroutine (P = 0 on 9-10) and transferring to a routine (P = 7 on 9-10). The processing of 9-10 is accomplished, after whith the instruction 70 9-10 J8 is executed; thus, the processing always returns to the point at which the subroutine was executed. In the transfer, no return ever occurs; if H5 is - so that 70 9-10 transfers

to 9-10, it is just as if the symbol 9-10 had been written in the link. Second, we have already noted that since the execution of 9-10 (P = 0) occurs as part of the subroutine 9-10, we have a recursion, whereas the transfer to 9-10 simply causes an iteration. In both cases, the routine is repeated, and so a recursion differs from an iteration simply in that the processing returns to the point of execution in a recursion. Lastly, this routine should provide an appreciation of how easily subroutine hierarchies are built up when the mechanization of subroutines is sufficiently smooth, as it is in IPL.

Notice that R3 tested the sublist 9-3 twice, because it appears on sublist 9-1 twice. This could have been anticipated, since J132 only lets us detect all the sublists, and gives no information as to whether a list has already been processed. To avoid such duplication, we must keep track somehow of those sublists that have been processed at any point. Indeed, if two lists hold each others' names, the recursive routines we have been writing may never terminate. We could restrict list structures to simple trees which allow a sublist to appear only once. We have preferred a more general form; however, it is far from the most general list structure possible: We require lists to terminate, and we don't allow one list to link into another.

It is always possible to keep a list of the local names and test against it, say with J77. A more efficient method is also provided in IPL. A list can be marked "processed" with a special mark, which can then be detected directly without a search. It is still necessary to keep a list of the local names, since the marks must be removed at the end of processing. Two J's provide the tools:

J133: Test if (0) is marked processed--that is, if P = 1 in the cell (0).

J137: Mark list (0) processed. This pushes (0) down and places a P = 1 in the head cell. The head cell also is put blank; i.e., now contains 0 for SYMB. (0) is left as the output.

Marking a list processed pushes the list down and puts a special signal (P = 1) in the top word that can be recognized by J133. The processing mark is "undone" whenever the list is popped up. Pictorially:

	NAME	PQ	SYMB	LINK
L8 at the beginning:	L8		C3	
			S4	
			S5	0
L8 after J137, now marked processed:	L8	1	0	
			C3	
			S4	
			S5	0
L8 after pop up operation, 30L8:	L8		C3	
			S4	
			S5	0

We used a list with a symbol in the head (top word), so that the effect of J137 could be clearly seen. (It can be seen from this example that the P and Q of data lists are used differently than the P and Q of routines.)

The modification in the R3 coding consists of marking each list of the list structure processed with J137, and then searching down only those local lists which are not yet marked. R3 must also create a list, put all the local names on it, and then clear it up at the end. A plan for this is shown in Fig. 4, together with the code.

R3: Set up (0) and a list for local names in the W's.

Execute recursive subroutine on main list (set H5+ if found, H5- if not).

Clean up the local names' list.

Pop each local name to unmark it.
Erase local names' list.

Clean up W's and exit with H5 as set by the recursive subroutine.

Recursive subroutine:

Mark list processed and add to local names' list.

Locate next symbol on list.
If none, exit - .

Test if it is the test symbol.
If yes, exit + .

Test if it is local (hence, names a sublist).
If not, loop to locate next symbol.

Test if marked processed.
If yes, loop to locate next symbol on sublist.

Execute recursive subroutine on sublist (set H5+ if found, H5- if not).
If test symbol found, exit +).

Loop to locate next symbol.

```
NAME   PQ   SYMB   LINK   COMMENTS

R3          J90           Create local name list.
            J51           Put local name list in W0, test symbol in W1.
            9-10          Execute recursive subroutine, result is + or - .
       40   H5            Push down H5 to save result.
       11   W0
9-2         J60           Locate next symbol on local name list.
       70   9-1
       32   H0     9-2    Pop it up, thus unmarking it (doesn't pop up H0).
9-1    51   W0            Input local name list (replacing unwanted symbol).
            J71           Erase list (see definitions of J's).
       30   H5     J31    Pop up H5, bringing back result; clean up W's.
9-10        J137          Recursive subroutine; mark list processed.
       61   W0            Put in top cell of local name list (in cell named in W0).
       41   W0            Push down local name list (push down cell named in W0).
9-11        J60           Locate next symbol on list being tested.
       70   J8            If at end, exit--(not find test symbol).
       12   H0
       11   W1
            J2            Test if symbol is test symbol (which is in W1).
       70          J8     If so, exit + .
       12   H0
            J132          Test if local.
       70   9-11          If not, proceed to next symbol.
       12   H0
            J133          Test if symbol already marked processed.
       70          9-11   If so, proceed to next symbol.
       12   H0
            9-10          Execute recursive subroutine on unsearched sublist.
       70   9-11   J8     If H5+, found symbol, quit; if not, go to next.
```

Fig. 4--Plan and Code for R3

4.1 PROBLEMS

PROBLEM 26

L9 below is a list structure which represents an algebraic expression in a hierarchical form. In this form, an operator is the first symbol on a list, followed immediately by its operands. If an operand is a sub-expression instead of a simple variable, the operand is represented by a local sublist of the same form as the main list. L9 is a tree because each local sublist name occurs only once in the structure.

TYPE	NAME	PQ	SYMB	LINK	COMMENTS
5		01			DATA HEADER FOR L9.
	L9		0		A+B=C in a hierarchical form.
			=		The main operator is EQUALS.
			9-1		One operand is the sub-expression (A+B).
			C	0	The other operand is the variable C.
	9-1		0		The subexpression A+B.
			+		The operator is PLUS.
			A		The operands are the simple
			B	0	variables A and B.

Code the list structure named X9, which will represent the algebraic expression "A+B=C-D" in a tree like L9 above. Then code the routine Y9 which prints X9 as a list structure.

PROBLEM 27

Represent the algebraic expression "A=B+(C*D)" by a list structure named X10. (The asterisk signifies multiplication and should appear in the structure as an operator. The parentheses indicate grouping and should not appear explicitly in the structure. Use L9 as a model.) Then code the routine Y10 which prints X10 as a structure.

PROBLEM 28

Code P77--"Test if variable (0) occurs in the expression named (1)." P77 assumes the expression is represented by a tree like L9, arbitrarily deep; thus, P77 should be recursive. Note that since L9 is a tree, P77 need not mark sublists processed.

PROBLEM 29

Code P78--"Create a list of the locations of all of the occurrences of variable (0) in the expression named (1)." P78 assumes that (1) names a tree similar to L9.

Set H5- and produce no output if the variable does not occur; otherwise, set H5+ and output the list of locations. The location of a variable means the name of the cell which holds it, not the name of the sublist on which it occurs.

PROBLEM 30

Code P79--"Create a list of the sublists of tree (1) on which the operator (0) occurs. Set H5- and produce no output if operator (0) does not occur; otherwise, set H5+ and output the list of sublists. The names of the sublists on the output list should be internal symbols." Use J138. (If we did not make the sublist names non-local with J138, we would erase the sublists if we ever erased the output list with J72. In practice, we would not need to make the sublist names internal if we were careful to use J71 to erase the output list.)

PROBLEM 31

Code P80--"Substitute the variable (0) for the variable (1) wherever (1) occurs in the expression named (2)." Use P78 of Problem 29, above.

PROBLEM 32

Code P81--"At every occurrence of the variable (1) in the expression named (2), substitute the name of a copy of the subexpression named (0). Make the name of the copy a local symbol with J136, since we assume (0) to be an internal symbol." Do not use P78. Will your routine take care of the case where subexpression (0) contains an occurrence of the variable (1)?

PROBLEM 33

A slightly different representation of algebraic expressions may give some insight into the practical differences between trees and more general list structures. Both L10 and L11 below represent the expression "(A+B)-(A-B)=(A-B)+(A+B)." L10 is a tree, since each sublist name occurs only once in the structure, where a given local sublist name may occur several places in the structure. L11 occupies less space than L10, but, as will be seen, must be processed with care.

TYPE	NAME	PQ	SYMB	LINK	COMMENTS
5		01			DATA HEADER FOR L10.
	L10		0		L10 is a tree, representing (A+B)-(A-B)=(A-B)+(A+B).
			=		The operator is EQUALS.
			9-1		9-1 represents the left half and
			9-2	0	9-2 the right half of expression

TYPE	NAME	PQ	SYMB	LINK	COMMENTS
	9-1		0		(A+B)-(A-B).
			-		The operator is MINUS.
			9-10		9-10 represents (A+B).
			9-11	0	9-11 represents (A-B).
	9-10		0		(A+B).
			+		The operator is PLUS.
			A		The left operand is A.
			B	0	The right operand is B.
	9-11		0		(A-B).
			-		The operator is MINUS.
			A		Left operand.
			B	0	Right operand.
	9-2		0		(A-B)+(A+B).
			+		The operator is PLUS.
			9-20		The left operand is (A-B).
			9-21	0	The right operand is (A+B).
	9-20		0		9-20 represents (A-B).
			-		
			A		
			B	0	
	9-21		0		9-21 represents (A+B).
			+		
			A		
			B	0	
5		01			DATA HEADER FOR L11.
	L11		0		[(A+B)-(A-B)]=[(A-B)+(A+B)].
			=		The main operator is EQUALS.
			9-1		The left operand is 9-1.
			9-2	0	The right operand is 9-2.
	9-1		0		(A+B)-(A-B).
			-		The operator is MINUS.
			9-10		(A+B).
			9-11	0	(A-B).
	9-10		0		9-10 represents (A+B).
			+		
			A		
			B	0	
	9-11		0		9-11 represents (A-B).
			-		
			A		
			B	0	
					9-2 represents the entire right half of the expression quite simply now, since it merely points to existing sublists.
	9-2		0		9-2 represents (A-B)+(A+B).
			+		
			9-11		
			9-10	0	

Code the expression "[(A+B)+(A+B)]=(A+B)" as a general list structure named X11, in which identical subexpressions are all represented by the same single sublist. Use L11 as a model. Then code the routine Y11 to print X11 as a structure.

NOTE: Problems 34 through 37 all assume that the algebraic expressions are represented by a general list structure like L11. Hence, sublists should be marked processed with J137 when first encountered, and tested for the process mark with J133 to avoid multiple processing. Correct solutions must remove the process marks from the sublists, and erase any temporary lists that were needed.

PROBLEM 34

Code P87, whose definition is the same as that of P77 (Problem 28, p. 40), with allowances for the fact that P87 must work on a general structure instead of a tree. P87 must mark local sublists processed and pop up the process marks before quitting.

PROBLEM 35

Code P88, whose definition is the same as that of P78 (Problem 29, p. 40), except that P88 operates on general structures like L11. P88 marks sublists processed and removes the process marks before terminating.

PROBLEM 36

Code P89, whose definition is the same as that of P79 (Problem 30, p. 41), except that P89 does not assume the structure to be a tree, but marks local sublists processed and cleans them up afterward.

PROBLEM 37

Code P90, defined like P80 (Problem 31, p. 41), except it works on general list structures. P90 should use P88 as a subroutine.

PROBLEM 38

Code P91--"Erase the tree named (0), including all its local sublists, without using J72." P91 also erases any lists it may create. J71 may be used.

PROBLEM 39

Code P92--"Erase the general list structure named (0), taking care not to attempt to erase a given local sublist more than once." Do not use J72. J71 and J137 should be used. P92 also erases any temporary lists it creates for its own use.

5.0 DATA TERMS

IPL works with symbols. These are not numbers, even though they are represented by numerical addresses in the computer. To work with information other than IPL symbols--integers, floating point numbers, alphabetic information, etc.--IPL uses data terms. For example, the integer data terms MO and NO are written as follows:

```
NAME  PQ SYMB  LINK  COMMENTS

MO    01        1   This is integer 1.
NO    01        2   This is integer 2.
```

Q = 1, when used with data, indicates that the remainder of the word is in a special form. The P codes tell what kind of data: P = 0 is used for integers, P = 1 is used for floating point numbers, and so on. Each data term has a name, which is an IPL symbol--the MO and NO in the example above. IPL routines manipulate data terms through their names. For example, one of the basic routines is J125:

J125: Add 1 to the numerical data term whose name is (0). Leave the name in HO.

If we use J125 on MO, we get:

```
H5  NAME  PQ SYMB  LINK  HO  PQ DATA COMMENTS
          10 MO          0
             J125        MO  01    1
                         MO  01    2
```

Now we have added a DATA column to the trace, which shows the data term named by the symbol in HO.

The four basic arithmetic operations in IPL require three inputs: (1) and (2) are the names of the data term operands, and (0) is the name of the data term which will hold the result. Thus, J110, the primitive process for addition, is defined as follows:

J110: (1) + (2)⟶(0). The data term named (0) is set equal to the algebraic sum of the data terms named (1) and (2). The output (0) is the input (0); i.e., the result.

Let us introduce a few of the data term processes by coding a simple routine, S1:

S1: Test if the sum of the integer data terms named on list (1) exceeds (0). (0) names a floating point data term. List (1) names data terms only, but some of them are not integers. All numeric data terms are positive.

The code for S1 follows below. It introduces three new data term processes, J115, J124, and J127, which are defined in § 11.0 of Part Two.

```
NAME   PQ SYMB   LINK   COMMENTS
S1        J90           Create cell to hold the sum,
          J124          make cell into integer zero.
          J51           WO holds name of sum; W1
                        holds name of limit to test
                        against.
9-1       J60           Locate next cell of the list.
       70 9-5           To 9-5 when done, with H5
                        set correctly.
       12 HO            Name of next data term to HO.
       11 WO            Name of data term known to be
          J127          integer.
       70 9-1           To 9-1 if not an integer data
                        term.
       12 HO            Name of next data term to HO.
       11 WO            Name of sum to HO twice,
       40 HO            once as operand, once as result.
          J110          ADD next to sum.
       11 W1            Name of limit to HO.
          J6            Reverse positions of sum
                        and limit.
          J115          Test if sum exceeds limit.
       70 9-1           Continue summing if no.
9-5    51 WO            Name of sum to WO.
          J9    J31     Erase sum and pop W's.
```

A few points about this code should be explained. Even though the limit (0) that we will test against is a floating point data term, we have chosen to accumulate the

sum as an integer data term. We are free to do this since J115, like most of the arithmetic processes, will operate successfully with numeric operands of mixed types. (J114, the equality test, is the only exception to this rule.) Alternatively, we could have chosen to accumulate the sum as a floating point data term; in this case, J110 would have converted the integer operand before adding it to the floating point sum, without any more coding on our part. Since it was immaterial whether the sum be integer or floating for the addition and comparison processes, but imperative that we have an integer data term around to detect integer data terms on the list (with J127), we chose to create a cell (J90) and make the cell into an integer data term equal to zero (J124). This satisfied both our requirements. S1 properly erases this data term (J9) before popping the W's, thus avoiding a common error encountered when working with data term processes.

Note that S1 would not function properly if the input list contained some symbols that did not name data terms; in this case, we would first have to detect (with J131) which symbols were the names of data terms before the J127 test would be valid.

5.1 PROBLEMS

PROBLEM 40

Code P93--"Test if the number of cells on H2, the available space list, exceeds (0). (0) is an integer data term." Use J126 but not J200. P93 erases any data terms it creates for its own use.

PROBLEM 41

Code P94--"Test if the sum of the integer data terms named on list (1) is equal to (0). (0) names an integer data term. List (1) contains the names of some cells which are not data terms and the names of some data terms which are not integers." (Consider the consequences of allowing input (0) to be a floating point data term.)

PROBLEM 42

The simple lists (0) and (1) are considered to represent unordered sets of symbols (by disregarding the inherent ordering of the lists). Each set may have only one occurrence of any given symbol. Hence, the sets are identical if each symbol of set (0) also occurs on set (1) and the sets contain an equal number of symbols. Using J77 and J126, code P95--"Test if set (0) is identical to set (1)."

PROBLEM 43

Code J200 as an IPL routine whose name is P96.

PROBLEM 44

Code P97--"Evaluate the algebraic expression named (0)." P97 assumes that (0) is a structure like X10 of Problem 27 (p. 40), where a single variable is equated to an arbitrary subexpression involving only the addition or multiplication operators. Assume that the variables name floating point data terms. Thus, in the case of X10, P97 should multiply the data terms named C and D, add the data term named B to this product, and set the data term named A equal to this sum.

PROBLEM 45

Write a routine to evaluate the recursive function:

$$A(M,N) = A(M-1,A(M,N-1))$$
$$A(M,0) = A(M-1,1)$$
$$A(0,N) = N+1$$

This function increases very rapidly with M, as the table shows; so, evaluation with large M should not be attempted.

$$A(0,N) = N+1$$
$$A(1,N) = N+2$$
$$A(2,N) = 2N+3$$
$$A(3,N) = 2^{N+3}-3$$
$$A(4,N) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} N+3 \text{ terms} \; -3$$

Can this function be coded in any natural way without using data terms and the arithmetic processes?

6.0 DESCRIPTION LISTS

Suppose we were building a program for playing contract bridge. We have certain symbols for each card of the deck: C2 for the deuce of clubs; D7 for the seven of diamonds; S13 for the king of spades; F1 for the ace of hearts (the H's are already being used); and so on. A hand would then be a list of 13 cards, such as X1:

NAME	PQ	SYMB	LINK	COMMENTS
X1		0		Hand consists of:
		C13		Clubs: K,Q,3.
		C12		
		C3		
		D7		Diamonds: 7,4.
		D4		
		F11		Hearts: J,7,6.
		F7		
		F6		
		S1		Spades: A,J,10,5,3.
		S11		
		S10		
		S5		
		S3	0	

During the course of play, we will find out various facts about the hand, such as the number of quick tricks. (There are two in X1.) This number is recorded with an integer data term, say N2. The problem is where to keep this information so that it is available when we want it. IPL has a general device, called a description list, for doing this. Let QO, say, stand for the number of quick tricks. Then we construct a list as follows:

NAME	PQ	SYMB	LINK
X1		9-1	
		C13	
		C12	
		...	
		S3	0
9-1		0	
		QO	
		N2	0

9-1 is the description list of list X1. Its name occurs in the head of X1--hence, given X1, it is possible to get at any information on the description list. Since the information on the description list is named (by Q0, here), it is possible to have many pieces of information on a description list, as long as they have different names. N1 is called the <u>value</u> of attribute Q0 of list X1. If Q1 stands for length of the longest suit, another relevant property of the bridge hand, we could also put this property on the description list:

NAME	PQ	SYMB	LINK
X1		9-1	
		C13	
		C12	
		...	
		S3	0
9-1		0	
		Q0	
		N2	
		Q1	
		N5	0

For such a device to be useful, there must exist processes to get information from description lists, and to add and modify the information on description lists. The following three J's are basic (although there are some additional ones also used with description lists):

J10: Find the value of attribute (0) on the description list of list (1). If it is found, it is the output (0) and H5 is set + ; if it is not found, there is no output, and H5 is set - .

J11: Assign the symbol (1) to be the value of attribute (0) on the description list of list (2). If attribute (0) already has a value, replace it with (1). If there is no description list, create it (with a local name).

J14: Erase the attribute (0) from the description list of list (1). This removes both the attribute symbol and the value symbol.

H5	NAME	PQ	SYMB	LINK	HO	COMMENTS
+		10	X1		0	
+		10	Q0		X1	
+			J10		Q0	
					N2	J10 has found N2, the value of Q0.
+		10	X1		0	
+		10	Q3		X1	
+			J10		Q3	
-					0	J10 found no value, hence H5-.
+		10	X1		0	After J11, the description list of X1 is:
+		10	N3		X1	
+		10	Q0		N3	NAME PQ SYMB LINK
+			J11		Q0	9-1 0
+					0	Q0
						N3
						Q1
						N5 0
+		10	X1		0	After J11, the description list is:
+		10	N6		X1	
+		10	Q4		N6	NAME PQ SYMB LINK
+			J11		Q4	9-1 0
+					0	Q4
						N6
						Q0
						N2
						Q1
						N5 0
						Note that J11 puts the new attribute at the front of the description list.
+		10	X1		0	After J14, the description list is:
+		10	Q1		X1	
+			J14		Q1	NAME PQ SYMB LINK
+					0	9-1 0
						Q0
						N2 0

Fig. 5--Trace of Description List Processes

To show how these operations work, we give a few examples of tracing in Fig. 5, each independently using list X1 above.

The description list has a local name, which means it belongs to the list structure of the main list. If X1 is ever erased as a list structure (see J72), then its description list will automatically be erased also. In the examples above, none of the attribute symbols or value symbols are local, so any lists they name would not be erased. However, any that were local would be erased. A local value would also be erased by J14, and by J11 if it replaced one (otherwise, the sublist would be lost space to the system). For example, if we had an attribute whose value was the biddable suits, this value would be a locally named list of from one to three symbols, each designating a suit. If we remove the value from the description list with a J14, we want to erase this list also.

6.1 PROBLEMS

PROBLEM 46

L12 below represents a list of books, B1, B2, etc. B1 is an empty described list which represents a book by the attributes A1 = Author, P1 = Publisher, and C1 = Cost. The value symbol of attribute A1 is regional if a book has a single author; otherwise, it is local.

TYPE	NAME	PQ	SYMB	LINK	COMMENTS
5		01			Type = 5, Q = 1, DATA HEADER.
	L12		0		L12 represents list of books.
			B1		
			B2		
			B3		
			B4	0	
	B1		9-0	0	9-0 is description list
	9-0		0		of B1.
			A1		A1 = Author.
			9-1		9-1 is list of authors.
			P1		P1 = Publisher.
			Y1		Publisher is Y1.
			C1		C1 = Cost.
			9-2	0	9-2 is data term giving cost.
	9-1		0		Authors of B1 are X1, X2.
			X1		
			X2	0	
	9-2	01		450	Cost of B1 is $4.50.

Code the empty described lists B2, B3, and B4, similar to B1 above. Assume each of the books has a single author, X2, X3, and X4, respectively; no known costs; and publishers Y2, Y3, and Y4, respectively.

PROBLEM 47

Code Q1--"Find the cost of B1, assign a copy of this data term to B2 as a local data term value of the attribute C1, and delete the attribute C1 and its value from B1."

PROBLEM 48

Code Q2--"Search the list of books (1) for books whose author or co-author is (0). Print (0) once, followed by the books he has authored. If he has authored none, print the word NONE."

PROBLEM 49

Assume that L13 is an empty described list whose attributes are the names of some of the books on L12 of Problem 46, above. The values are local integer data terms representing the number of copies of the particular book that have been sold. Code Q3--"For each book on L12, add the attribute D1 to the book's description list if the book is found as an attribute on L13. The value of D1 (which represents "Distribution") for each book should be a local copy of the integer data term found as the value of the book on L13."

PROBLEM 50

Code Q4--"For each book on list (0), add the author Y1 at the end of the list of authors if the book already has more than one author. If the book has only a single author, delete the attribute A1 and its value."

7.0 DESCRIPTION LISTS, CONTINUED

Description list processes are very powerful and can be used in many flexible ways. To give some feeling for this, let us pursue further the problem of the bridge hand. As we pose more complicated problems, we gradually move into the area of real programming, where a representation must be chosen for the problem and where this representation determines the efficiency and elegance of the code.

Our problem is to proceed with the analysis of the hand prior to bidding it. We have already chosen a straightforward representation of the bridge hand. It comes originally as a simple list in random order from the deal. The first task is to reconstitute the hand (which we shall continue to call X1), so that it consists of a list structure with four sublists, one for each suit. Each of the sublists should be in descending order on the value of the card, just as some people normally arrange a bridge hand. We want the "point count" value of the hand, which we shall denote by the attribute Q2. For the point count, each card gets a specified number of points--4, 3, 2, 1, or 0-- depending on its type--ace, king, queen, jack, or small card. We will need to associate with each of the 52 card symbols (the C3, S11, etc.) its type according to an attribute, say Q3. To avoid complexity, we can represent the types by data terms corresponding to their point counts. (We could also look up the point count in a table, which would be another description list with types for attributes and point counts for values.) We can use T's for types: T1 for ace, T13 for king, and so on. The small cards will have no type. An example is shown below:

```
NAME   PQ SYMB   LINK   COMMENTS
C12       9-1    0      The queen of
9-1       0             clubs has the
          Q3            point count 2
          T12    0
T12    01            2
```

We also need to associate with each card symbol its "card value," which is a number between 2 and 14 that determines the ordering (and which card takes which in play). We can use Q4 as the attribute for this, and the symbols N2 through N14 for the integers. Finally, we must have a suit symbol--say, CO, DO, FO, and SO--associated with each card symbol; we will use Q5 as the attribute for this.

Before we can begin planning the routines, we must agree on a format for the hand list. We could have it contain the names of the four sublists in some specified order. Alternatively, we can make the hand list in the form of a description list, with suit symbols as attributes and sublists as values. Then J10 can use the suit symbols to select the sublist for that suit. With this convention, the final arranged form of X1 would be:

```
NAME   PQ SYMB   LINK
X1        0
          CO
          9-1
          DO
          9-2
          FO
          9-3
          SO
          9-4    0
9-1       0
          C13
          C12
          C3     0
9-3       0
          F11
          F7
          F6     0
9-2       0
          D7
          D4     0
```

```
NAME   PQ SYMB   LINK
9-4       0
          S1
          S11
          S10
          S5
          S3     0
```

Now we can formally define our routine. At the top level we have R4, whose task is to sort the hand and "analyze" it (get the point count). As subroutines within this, we have R5 whose task will be to rearrange the list, and R6 whose task will be to total the point count:

R4: Rearrange and analyze the hand list (0), which initially consists of 13 card symbols in random order.

R5: Rearrange the hand list (0) into ordered form, as above.

R6: Compute the total point count of the hand list (0) (assume already arranged). Output is a new integer data term, (0).

Neither R4 nor R5 has any output in HO, since they only modify a structure whose name is already known to the routine using them. Given R5 and R6, R4 is almost trivial to code (we have simply put all the work off by giving names to the subroutines):

```
NAME   PQ SYMB   LINK   COMMENTS
R4     40 WO             Push down WO.
       60 WO             Put name of list in WO,
                         but leave in HO.
          R5             Rearrange.
       11 WO
       40 HO
          R6             Compute point count.
          J136           Make local (to tie it to
                         hand list).
       10 Q2
          J11    J30     Put on description list with
                         attribute Q2, clean up.
```

One of the problems in coding R5 is that we want to modify the same list from which we are taking information.

We need a way of detaching the body of the list. The following J does the job:

J75: Divide list after location named (0). ((0) gets a 0 for LINK.) Construct a new list, consisting of a new head (which is empty) plus all the remaining symbols after (0). Output the name of this list as (0).

The sequence below illustrates J75 quite generally:

List L1 at beginning:

NAME	PQ	SYMB	LINK
L1		0	
9-1		S1	
9-2		S2	
		S3	
		S4	0

Program sequence that gets second list cell, then does J75 to divide list:

NAME	PQ	SYMB	LINK	H0
	10	L1		0
		J60		L1
		J60		9-1
		J75		9-2
				3125

L1 after program, and new list, 3125, with remaining symbols:

NAME	PQ	SYMB	LINK
L1		0	
		S1	
		S2	0
3125		0	
		S3	
		S4	0

Using J75, we can form a plan for R5, shown in Fig. 6 together with the code. The routine is rather long, but it accomplishes a good deal; there is very little lost motion except for the repeated searches of the symbols on

```
R5:  Split the hand list to get all the symbols on a separate working list.
     For each symbol on the working list:
        Find the right sublist:
           Use Q5 to find suit symbol;
           Use suit symbol to find sublist:
           - If not find, create it.
        Insert card symbol at right place on sublist:
           Compare card values for successive symbols (Q4):
           - If new value is lower, keep going;
           - If new value is higher, insert before symbol on list.
     Erase working list.
```

NAME	PQ	SYMB	LINK	COMMENTS
R5	60	9-10		Place hand list in 9-10, leave in H0.
		J75		Detach list from head, (0).
	40	H0		Save remainder list for erasing at end.
9-5		J60		Find next card symbol to be sorted.
	20	9-20		Place in cell 9-20.
	70	J71		If no more, through; erase remainder list and quit.
	12	9-20		
	10	Q5		
		J10		Find suit symbol, assume it exists.
	20	9-30		Place in cell 9-30.
	10	9-10		Input hand list (must be 9-10, not list name, for J10).
	11	9-30		Input suit symbol.
		J10		Find sublist.
	70	9-1		If not exist, go to create it.
	12	9-20		Input card symbol.
	10	Q4		
		J10		Find card value, assume it exists.
	20	9-40		Place in cell 9-40.
9-3		J60		Locate the next card symbol on sublist.
	70	9-2		
	12	H0		
	10	Q4		
		J10		Find card value of symbol on sublist, assume it exists.
	11	9-40		Input card value of new symbol to be sorted.
		J116		Compare: H5 + if old greater than new; H5 - if not.
	70		9-3	If H5 +, to go next symbol on sublist.
	12	9-20		New greater than old.
		J63	9-4	Insert on sublist ahead of old symbol.
9-2	12	9-20		End of list.
		J64		Insert on sublist at end (after current end).
9-4	11	9-20	9-5	Common input of remainder list, go to next symbol.
9-1		J90		No sublist, create it.
		J136		Mark it local (to tie list structure together).
	40	H0		Push down to save it for after J11.
	10	9-10		Input hand list.
		J6		Invert order in H0 to agree with J11 conventions.
	11	9-30		Input suit symbol.
		J11	9-2	Assign sublist, return to put symbol on it.
9-10		0	0	Local cell: holds input hand list.
9-20		0	0	Local cell: holds location in remainder list.
9-30		0	0	Local cell: holds suit symbol.
9-40		0	0	Local cell: holds card value of symbol to be sorted.

Fig. 6--Plan and Code for R5

the sublist for their card values. We used local cells, rather than the W's, for temporary storage, although we could just as well have used the latter. We did not need to push down the local cells, since we knew exactly what they were to be used for.

One common device in working with description lists is to define attribute symbols to be routines that find the attribute values. In our present case, for example, we might define two routines:

NAME	PQ	SYMB	LINK	COMMENTS
Q4	10	Q4	J10	Q4 is the routine that finds the value of attribute Q4.
Q5	10	Q5	J10	Q5 is the routine that finds the value of attribute Q5.

This will replace the various occurrences of two lines--10Q4, J10--with a single routine, Q4. Although the total processing remains the same, it saves a line of code at each occurrence. More important, perhaps, it makes it natural to think of an attribute as a single-valued function, so that Q4(X1) has the same form as $\sin(30^{\circ})$--i.e., both are functions that produce values, given arguments.

To accomplish R6, we need to add up all the point counts of all the card symbols. We must proceed separately for the four sublists. (It might have been simpler to get the point count before the list was arranged, but, in general, the various features of the hand will depend on suit and order, so we preferred to proceed this way.) A code for this is as follows:

```
NAME   PQ  SYMB   LINK   COMMENTS
R6         J41           Preserve WO and W1.
       20  WO            Put hand list in WO.
           J90           Create cell for out data term.
           J124          Set it to 0, leave in HO.
9-1    11  WO
           J60           Locate next symbol.
           J60           Locate next symbol (names sub-
                         list).
       20  WO            Put back in WO before
                         branching.
       70  J31           If no more sublist, clean up
                         W's, quit.
       12  WO            Input name of sublist.
9-2        J60           Locate next card symbol.
       20  W1            Put back in W1 before
                         branching.
       70  9-1           If no more, go to get next
                         sublist.
       12  W1            Input card symbol.
           Q3            Find type (Q3 = 10Q3 J10).
       70  9-1           If not find, go to next
                         sublist.
           J6            Invert order in HO for J110.
       40  HO            Double up output data term
                         for J110.
           J110          Add point count, put in
                         data term.
       11  W1     9-2    Input location in sublist,
                         continue.
```

As a matter of technique, it is perhaps worth noting that we delayed branching after the J60's until we had put the result back in the W's. This avoided an additional pop up of HO in case the H5- branch occurred. Notice also, that although J90 created a new cell for us, it has to be preset to zero to make it into a legitimate integer data term. Finally, in R5 we used the hand list as a description list; here in R6 we used it as a regular list, iterating down it and selecting out every other symbol. The moral is that, although the description list processes are especially fitted to description lists, they are also general purpose search and insert operations, to be intermixed to advantage with other list processes.

Computing the point count was simple, because we could treat each card independently. If we had chosen quick trick count instead, we would have been involved with combinations of cards. Suppose quick tricks (QO) are defined for each suit according to the table in Fig. 7. Since bridge books

Combination	Count	Equivalent value
AKQ	2-1/2	N10
AK	2	N8
AQ	1-1/2	N6
AJ	1 +	N5
A	1	N4
KQJ	1-1/2	N6
KQ	1	N4
KJ10	1	N4
Kx	1/2	N2
QJx	1/2	N2
Qxx	+	N1

Fig. 7--Quick Trick Table

take the liberty of adding "plus" values, we have taken the liberty of considering these to be quarter points and multiplying everything by four to get integer terms. We now wish to add to R4 the computation of quick tricks, say by a routine R7. Reference to the previous code shows that we can simply add another small sequence at the end:

```
NAME  PQ SYMB  LINK  COMMENTS
R4    .. ...
      11 WO          (This occurs after J11 in
                      prior code, with J30 removed.)
      40 HO
         R7
         J136
      10 QO
         J11   J30
```

R7 is defined similarly to R6. It takes the arranged list as input and provides a new data term with the total count as output. In fact, examination of R6 shows that if we had a routine for getting the quick trick count of a sublist

(suit), say R8, we could simply replace the 9-2 loop by:

```
NAME   PQ SYMB   LINK   COMMENTS
R6     .. ...
       12 W0           Input name of sublist.
          R8           Find count for suit (not a
                       new data term).
          J6           Invert for J110.
       40 H0           Double up output data term
                       for J110.
          J110  9-1    Add in count to total
                       (leave in H0).
```

Hence, we can restrict ourselves to the interesting problem, which is to code R8:

> R8: Find quick trick count of list of cards, (0). The list is arranged according to R5; that is, with sublists for each suit. The output data term is not new (hence, not to be erased).

The problem is how to arrange the information conveyed by the table. We could build it into a large routine with many conditional transfers. Better, we could construct a data list structure with this information in it in some natural way, such that a relatively simple routine could find the value by consulting it in conjunction with the information in the list of cards. The ordering of the cards in the list gives the clue: If we pick up the first card, it is the highest type that occurs, and should allow us to select a subset of the table corresponding to all those cases where this type is the highest. We can visualize a net of questions, shown in Fig. 8. We have put in only some of the nodes. The path through this net goes down to the left until a yes answer is received--that is, until the type of the first card is known. Then we pick up the next card (move one step down to the right) and again travel down to the left until its type is known. Eventually a value is reached. This suggests that, instead of viewing the tree of tests as symmetric in "yes" and "no," we might better depict it as multiple branches,

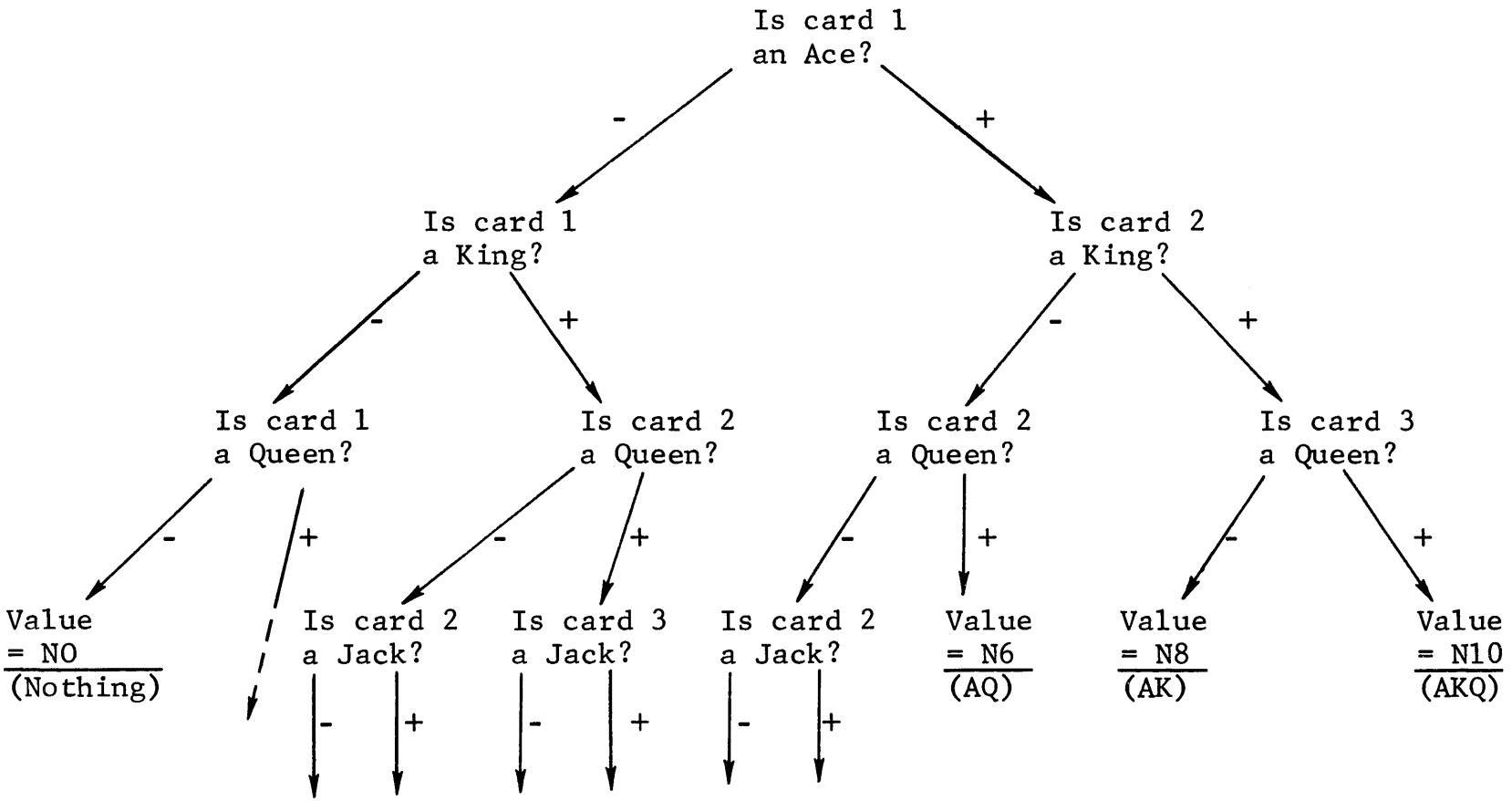
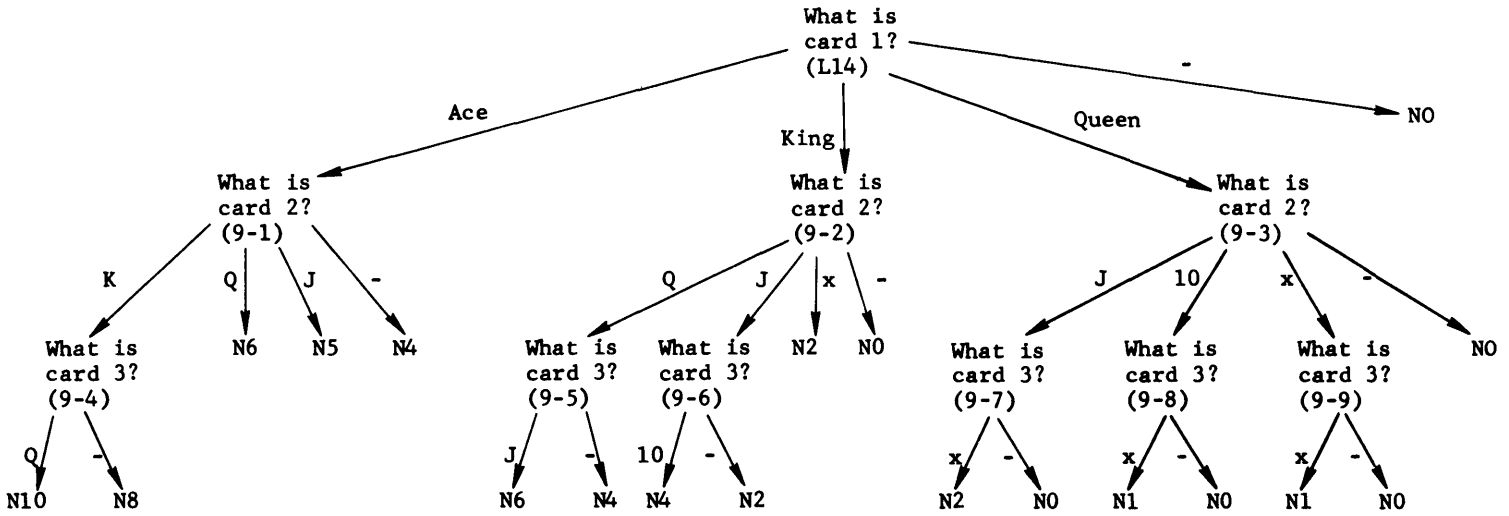


Fig. 8--Quick Trick Table Structure (Incomplete)

as in Fig. 9. Moreover, rather than do a sequence of tests on card 1, we can use J10, the search on description lists, to determine what type it is. The types will be the attributes, and their values will either be the counts or the sublists giving further differentiation in the table. The tree of Fig. 9 uses "x" for a small card and "-" for anything else. This latter includes the case of finding no additional cards, whereas "x" implies that at least one card is found. The bottom half of the figure shows the table laid out as a list structure. Type-x is represented by T0. The count corresponding to "anything else" appears in the head of each sublist. (It cannot appear in the list as a value, since there will be no attribute to which it corresponds.) We have had to differentiate T10, the Type-10 card, from T0, the small card. The names of the lists are given on the tree to show the correspondence.

The difficult part of most problems is finding a good representation of the data. Having found one, the coding is direct:

```
NAME   PQ SYMB  LINK  COMMENTS
R8     10 L14        Input the name of the table.
          J51        Put table name in W0,
9-3    11 W1         card list name in W1.
          J60        Locate next card.
       20 W1         Put back in W1 prior to branching.
       70 9-1        If no more cards, must be
                     "anything else" case.
       10 W0         Input sublist of table (J10
                     requires the symbol W0 rather
                     than the list name).
       12 W1         Input card.
          Q3         Find type.
       70       9-2
       10 T0         If no type found, must be type T0.
9-2       J10        Find entry in sublist of table.
       70 9-1        If not find, must be
                     "anything else" case.
       60 W0         Replace table sublist with new
                     sub-sublist.
          J132       Test if local.
       70       9-3  If it is, another sublist;
                     if not, W0 holds count.
       11 W0    J31  Input count, clean up.
9-1    12 W0    J31  Input count from head for
                     "anything else" case, clean
                     up.
```



NAME	PQ	SYMB	LINK
L14		N0	
		T1	
		9-1	
		T13	
		9-2	
		T12	
		9-3	0
9-5		N4	
		T11	
		N6	0

NAME	PQ	SYMB	LINK
9-1		N4	
		T13	
		9-4	
		T12	
		N6	
		T11	
		N5	0
9-6		N2	
		T10	
		N4	0

NAME	PQ	SYMB	LINK
9-2		N0	
		T12	
		9-5	
		T11	
		9-6	
		T10	
		N2	
		T0	
		N2	0

NAME	PQ	SYMB	LINK
9-3		N0	
		T11	
		9-7	
		T10	
		9-8	
		T0	
		9-9	0
9-7		N0	
		T10	
		N2	
		T0	
		N2	0

NAME	PQ	SYMB	LINK
9-4		N8	
		T12	
		N10	0
9-8		N0	
		T0	
		N1	0
9-9		N0	
		T0	
		N1	0

Fig. 9--Graphic Representation of Quick Trick Table and its IPL Equivalent

This is a remarkably short routine, which only demonstrates that we have been successful in organizing the relevant information in a convenient way. We even accepted the convention laid down earlier that small cards would have no type symbol, and we provided for it in the routine (10TO). If we were actually to use R8 in a program, we would probably stipulate that all cards would have type symbols. Then Q3 would always find a result, and the instruction that follows it could be eliminated.

7.1 PROBLEMS

PROBLEM 51

Below is an alternate version of L14; it represents the information in the quick trick table of Fig. 7 in a simple format; each sublist of L14 represents one line of the table. The symbols on each sublist represent a particular combination of card types (denominations); the symbol in the head of the sublist names the data term value of that particular combination. Recode the routine R8 (p. 61), so that it will work with the version of L14 shown below.

NAME	PQ	SYMB	LINK	COMMENTS
L14		0		L14 represents the table of quick tricks.
		9-1		AKQ
		9-2		AK
		9-3		AQ
		9-4		AJ
		9-5		A
		9-6		KQJ
		9-7		KQ
		9-8		KJ10
		9-9		KJ
		9-10		K10
		9-11		Kx
		9-12		QJ10
		9-13		QJx
		9-14		Q10x
		9-15	0	Qxx
9-1		N10		AKQ = 10
		T1		ACE
		T13		KING
		T12	0	QUEEN

NAME	PQ	SYMB	LINK	COMMENTS
9-2		N8		AK = 8
		T1		
		T13	0	
9-3		N6		AQ = 6
		T1		
		T12	0	
9-4		N5		AJ = 5
		T1		
		T11	0	
9-5		N4		A = 4
		T1	0	
9-6		N6		KQJ = 6
		T13		
		T12		
		T11	0	
9-7		N4		KQ = 4
		T13		
		T12	0	
9-8		N4		KJ10 = 4
		T13		
		T11		
		T10	0	
9-9		N2		KJ = 2
		T13		
		T11	0	
9-10		N2		K10 = 2
		T13		
		T10	0	
9-11		N2		Kx = 2
		T13		
		TO	0	
9-12		N2		QJ10 = 2
		T12		
		T11		
		T10	0	
9-13		N2		QJx = 2
		T12		
		T11		
		TO	0	
9-14		N1		Q10x = 1
		T12		
		T10		
		TO	0	
9-15		N1		Qxx = 1
		T12		
		TO		
		TO	0	

PROBLEM 52

Given the version of L14 shown in Problem 51 above, code the routine R90 which transforms L14 into the version shown in Fig. 9. R90 has no input or output, and erases any temporary lists or cells that it creates. It also erases the original structure L14.

8.0 USING GENERATORS

It is possible to write compact codes that do lots of processing simply by using various sequences of instructions repeatedly. Already in IPL we have made extensive use of three devices for achieving this: we have written loops, making use of the conditional branch; we have built subroutines, which can be used in many different places; and, we have written recursive programs, in which the same routine is used within itself. There is another device in IPL to accomplish repeated processes, called a generator. The generator reflects the fact that the generation of a set of things to be processed is often quite independent of the processing that is to be done on them. To take an example, suppose we wanted to recode J77, which tests if a symbol, (0), is on a list, (1). The relevant generator is J100:

J100: Generate all the symbols on list (1). Input each one to H0 and apply the process named (0) to it in turn.

The code for J77 using J100 would be:

```
NAME   PQ SYMB   LINK   COMMENTS
J77       J50           Put test symbol in W0.
       10 9-10          Input name of subprocess;
                        name of list already in H0.
          J100          Execute generator.
          J5     J30    Result is H5+ if looked at
                        all symbols in vain; reverse
                        sign.
9-10   11 W0            Subprocess: input test symbol.
          J2     J5     Test; reverse sign to stop
                        generator if find symbol.
```

In order to understand this code, we can visualize three routines, as shown in Fig. 10. There is J77, the superroutine. Besides a little housekeeping (the J50, J30, and J5), it consists of two parts: J100, which produces a stream of symbols; and 9-10, which tests each symbol

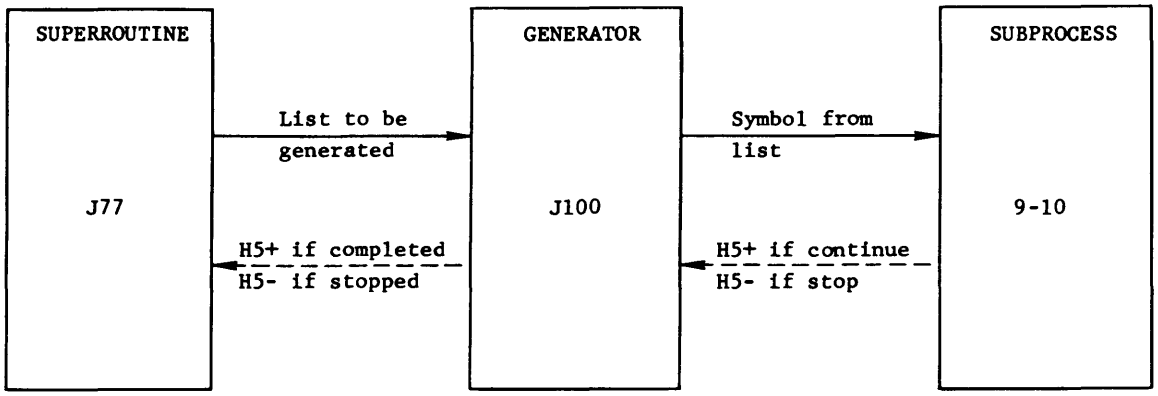


Fig. 10--Generator Relationships

against the symbol in WO. Since, in the computer, they cannot both work at once, control alternates between them. First, J100 operates long enough to produce a symbol; then, 9-10 tests it; then, J100 produces the next symbol; and so on. Processing should stop, of course, if 9-10 ever finds the symbol for which it is testing. There is communication back from 9-10 to J100 telling it whether to continue or not. This follows the convention: 9-10 exits with H5+ if the generator is to continue; that is, if it is to find the next symbol, put it in HO, and execute the subprocess 9-10 again. 9-10 exits with H5- if the generator is to stop and return control to the superroutine. Thus, 9-10 reverses the sign of H5 (with J5) which results from J2, since J2 gives H5+ if it finds the symbol and H5- if it doesn't. There must also be communication to the superroutine; otherwise, it could not determine whether the symbol was found or not. This communication is from J100 to the superroutine at the time J100 finishes. It follows the convention: J100 exits with H5+ if it generates all the symbols and was not asked to stop by the subprocess; J100 exits with H5- if it was asked to stop by 9-10. Thus, J77 forms its output by reversing this sign (J5), since if J100 exits with H5+, it means that the symbol was not found, and so J77 should exit with H5-. These two communication

conventions are not unique to J100. They hold universally for generators in IPL, whether in the basic J's, or constructed for a particular program.

We can do multiple iterations with generators. Suppose, for example, we had a generator R9, that produced only the local symbols on a list:

> R9: Generate all the local symbols on list (1). Input each one to H0 and apply the subprocess (0) to it in turn.

Now we can use J100 and R9 to rewrite R6 of the previous section, to total up the point count of a bridge hand by a double iteration, first over the lists for each suit, and within this, over the cards of the suit.

```
NAME   PQ SYMB   LINK   COMMENTS
R6        J90           Create output data term.
          J124          Preset it to 0.
          J6            Invert to put list on top.
       10 9-10  R9       Input subprocess, iterate
                         over suit sublists.
9-10   10 9-20           Input subprocess, iterate
                         over cards symbol in sublist.
          J100  J4
9-20      Q3             Subprocess for card symbol,
                         find type.
       70 0              If not find, exit H5-, stop-
                         ping iteration of this sublist.
          J6             Invert for J110.
       40 H0    J110     Accumulate count, exits with
                         H5+ to continue.
```

Note that 9-10 must set H5+ (with a J4) as a signal to R9 to continue with the next suit. If 9-10 does not do this, the H5- by which 9-20 signals the end of a sublist will stop both 9-10 and R9, and no further points will be counted.

One additional convention shows in this routine: H5 is safe over basic J processes. Thus, after Q3 had set H5+, the J6 and J110 did not modify this, since neither of them sets H5 as part of its output. As this routine indicates, sometimes large space savings can be achieved by using generators rather than by looping. This space is

purchased at the price of additional processing to shuffle the contexts back and forth. The main virtue of generators is to permit a complex generation process to be coded once, and then used repeatedly without further coding or check-out.

While the subject of generators is fairly complex, a good deal of confusion will be avoided if pains are taken to distinguish clearly the coding that is needed to use or execute a generator, and the coding that is needed to construct a generator. To use a generator is purposely quite simple; the main concern should be that the sub-process is coded to set H5 to continue or to terminate the generator, as desired, after each element is processed. Even when the subprocess itself uses a generator, and its subprocess uses a generator, etc., our attention is focused on what each subprocess is doing with each element of the set being presented, rather than on what processing is going on in the generators. (See § 7.2 of Part Two, CONVENTIONS FOR USING GENERATORS.)

Besides J100, which generates the symbols of a list, a few other generators are already constructed for the programmer to use. For example, J102 generates all the cells (not the symbols) of a tree.

J102: Generate the cells of tree (1) for sub-process (0). The subprocess named (0) is performed successively with each of the names of the cells of the tree named (1) as input. A tree is a data list structure in which each sublist appears once and only once. The cells of each sublist are generated before going on with the super-list; the cell containing the name of the sublist occurs immediately before the sub-list and all its sublists are generated. H5 is set + to the subprocess if input (0) is the head of a new sublist, and is set - otherwise. (Nothing is marked processed, since there is no need to keep track of multiple occurrences.) The name of the next cell to be generated is found before

the cell is presented to the subprocess--i.e., it is possible to erase a tree with J102.

J102 will move in list structure (1) if it is on auxiliary.

R77 below is a simple example using J102 to generate the cells of L10 of Problem 33 (p. 41). L10 represents an algebraic expression as a general list structure rather than a tree, but since there are no sublists which contain their own name, J102 will not loop endlessly. R77 purports to print the algebraic expression in Polish prefix form--e.g., to print X+Y as +XY and (X-Y)·Z as ··-XYZ. The subprocess 9-10 contains a common error: It fails to set H5 properly.

TYPE	NAME	PQ	SYMB	LINK	COMMENTS
5					ROUTINE HEADER FOR R77.
	R77	10	L5		Input structure name.
		10	9-10	J102	Input subprocess name, do J102.
	9-10	70		J8	Skip heads of sublists.
		52	HO		Operator or operand to HO.
		40	HO		Save it in case we print.
			J132		Test for local. Operand may be subexpression yet to be generated.
		70	J152	J8	Print if not local; otherwise, pop HO.

R77 would terminate after printing the first operator, since J132 in the subprocess 9-10 has set H5-, which the generator J102 interprets as a signal to terminate. H5 should be set + after the negative branch is selected and before 9-10 terminates.

8.1 PROBLEMS

PROBLEM 53

Code Q5--"For each regional symbol on L12 of Problem 46 (p. 51), copy the structure named by that symbol, assign the regional symbol as the value of the attribute N1 (meaning name) on the description list of the copy, and add the internal name of the copy to the end of list X12." Use J100 to generate L12.

PROBLEM 54

Code Q6--"For each regional symbol on L12 of Problem 46 (p. 51), copy the structure named by that symbol and replace that symbol on L12 with the internal name of the copy." Use J102 to generate the cells of L12.

PROBLEM 55

Code Q7--"Test if any locally named value list found on the description list of any of the lists named by regional symbols on L12 contains the symbol (0)." Q7 exits with H5+ if the input symbol (0) is found on any of the locally named value sublists, and with H5- otherwise. Use J100 three times: to generate the lists named on L12; to generate the attribute-value pairs on the description lists of these lists; and to generate the symbols of the locally named value lists found on these description lists.

PROBLEM 56

Code Q8--"Test if L10 of Problem 33 (p. 41) contains the variable (0)." Q8 exits with H5+ if yes, - if no. Use J102.

9.0 GENERATOR CONSTRUCTION

J100 is the simplest of generators; let us examine how it is constructed. Basically, of course, it consists of a loop to go down the list. But it must have some place to keep both the location in the list and the name of the subprocess while the subprocess is working. Likewise, it must be sure that it doesn't destroy any of the information used by the subprocess. To see this problem clearly, suppose we code J100 as follows, which seems perfectly straightforward:

```
NAME   PQ SYMB   LINK   COMMENTS
J100      J51           Put subprocess in W0, list
                        name in W1, push down W's.
9-2    11 W1            Input location on list.
          J60           Locate next.
       20 W1            Put back in W1 prior to branch.
       70 9-1
       12 W1            Input symbol in cell of list
                        for subprocess.
       01 W0            Execute subprocess (P=0, Q=1).
       70 J31   9-2     If subprocess returns
                        H5-, quit with H5-.
9-1       J4    J31     If went to end of list,
                        quit with H5+.
```

Let us trace through the beginning of J77 with this version of J100, keeping track of what is in W0 as well as H0:

```
LEVEL  NAME   PQ SYMB   LINK  H5  H0    W0    COMMENTS
1      J77       J50          +   S4    0     S4 is the test symbol.
1             10 9-10         +   L1    S4    L1 is the list to be
                                               searched.
1                J100         +   9-10  S4
 2     J100      J51          +   9-10  S4    9-10 goes into W0,
 2             11 W1           +   0     9-10  L1 into W1.
 2                J60          +   L1    9-10
 2             20 W1           +   9-1   9-10  9-1 names the first
                                               cell list.
 2             70 9-1          +   0     9-10
 2             12 W1           +   0     9-10
 2             01 W0           +   S2    9-10  S2 is the symbol in
                                               cell 9-1.
  3    9-10    11 W0           +   S2    9-10  9-10 executed by
                                               1W0 instruction.
  3                J2           +   9-10  9-10  Get 9-10 instead of
                                               S4; ERROR.
```

From the last line of trace we see that the subprocess within J77, while expecting S4 to be in W0, finds 9-10 instead.

This latter symbol was being used by J100 (so that, in fact, S4 is sitting in W0 right below it), and it is necessary somehow for J100 to remove its working information from W0 before executing the subprocess of J77. That is, both J100 and J77 wish to use the same cell, each in ignorance of the other. Thus, the problem is one of changing <u>contexts</u> of information. 9-10 works in the same context as J77; J100 works in an entirely different context. These contexts must alternate, just as the control alternates back and forth between generator and subprocess. The basic assumption of a strict hierarchy of routines is violated here, and with it the ability to have cells safe just by pushing down and popping up.

We show below the correct code for J100, which will illustrate the additional processes that are needed to accomplish generators.

```
NAME  PQ SYMB  LINK  COMMENTS
J100  10 W0         Input the symbol W0 for J17:
                    want one cell of working storage.
         J17        J17 sets up housekeeping,
                    stores away subprocess.
9-1      J60        Locate next symbol.
      20 W0         Put back in W0 prior to branch.
      70 J19        If end of list, exit with
                    generator clean-up, J19.
      12 W0         Input symbol for subprocess.
         J18        Execute subprocess; J18
                    changes contexts.
      70 J19        If subprocess returns with
                    H5-, exit with J19.
      11 W0   9-1   Input current location in
                    list and cycle.
```

Three routines handle the housekeeping. Roughly, J17 sets up a "generator hideout" in which to store the context information. The "highest" W to be used by the generator is input to J17 (here just W0). J17 pushes down the W's. J18 executes the subprocess stored away in the hideout by J17. It also removes all the generator's information, so that the context of the superroutine is returned to the W's. After the subprocess is through, J18 brings back the generator's context again. It thus assures that the difficulty that occurred in our trace will not occur. Finally,

J19 cleans everything up: It cleans up the hideout, pops up the W's, and sets the correct output signal for H5 (J18 has kept track of how the subprocess reported to the generator).

Generators can be cascaded to produce other generators. That is, a generator can be formed by taking the output of another generator and modifying it. This is best understood by an example. In the previous section we defined a generator, R9, which produced only local symbols. R9 can be constructed by using J100 and simply discarding those symbols it produces which are not local.

R9: Generate all the local symbols on list (1), and apply the subprocess (0) to each of them.

NAME	PQ	SYMB	LINK	COMMENTS
R9	10	WO		
		J17		Set up context.
	10	9-10		Input subprocess for J100.
		J100	J19	Execute J100, and quit afterwards with J19.
9-10	60	WO		Save symbol in WO.
		J132		Test if local.
	70	J4		If not local, set H5+ to call for next one.
	11	WO	J18	Input symbol and execute subprocess to R9.

Notice that our use of J100 within the code of R9 does not cause any difficulty, even though J100 is used in R6 within the subprocess to R9. In short, generators can be combined safely in almost any way.

As a more complex example of constructing generators, we will describe the code for J101, a basic process which generates all the cells on a list structure. J101 generates the cells rather than the symbols in the cells, since it is meant to be useful in manipulating list structures as well as searching them. There are several possible ways to generate the cells of a list structure, and it is

necessary to settle on one of them. J101 generates the cells in an order called print order, which is the order used to print a structure for human inspection. It is illustrated below:

NAME	PQ	SYMB	LINK	COMMENTS
L1		0		The main list is printed first.
		S1		
		9-1		
		9-2		
		S2	0	
9-1		0		The first sublist of L1 comes next.
		9-3		
		9-3		
		S3	0	
9-2		0		The other sublists of L1 come next, before the sublists of 9-1.
		9-4	0	
9-3		0		The sublists of 9-1 come before the sublists of 9-2.
		S5	0	Sublist 9-3 is only printed once, although it occurs twice.
9-4		0		Finally 9-4 is printed.
		S6	0	

The general rule is to generate all the cells of a list in sequence, accumulating its sublists, and to take sublists in the order in which they are first encountered.

A second question concerns the problem of marking processed with J137. Some marking must be done or J101 will not successfully generate many legitimate list structures, such as the following:

NAME	PQ	SYMB	LINK	COMMENTS
L2		0		
		9-1	0	
9-1		0		
		9-1	0	

The convention adopted is to have J101 mark each sublist processed prior to presenting it to the subprocess, and to have J101 remove the processed marks as part of its

clean up. Local symbols that name data terms are treated in the same way as other sublists and are marked processed. A consequence of this is that every time J101 presents the head of a sublist to the subprocess, that cell has a processed mark in it. The actual contents of the head are in the cell following, which was added to the list by the operation of marking processed.

In order to work with a list structure effectively, it is necessary to know when a new sublist is being generated. The output from J101 to the subprocess includes setting H5+ if the cell is the head of a new sublist; and setting H5- for any other cell. (Since names of sublists will be marked processed, they could be distinguished by J133, and the H5 convention is redundant; however, it is useful.)

A final issue concerns how much manipulation of the generated list structure is allowed. It is undoubtedly possible to construct a subprocess that can fool any generator into misbehaving, by searching ahead in the list structure and recomposing the structure in various ways. On the other hand, certain manipulations should be acceptable. For J101, we adopt the principle that it should be possible to erase a list structure with J101. This means that J101 must have the name of the next cell to be generated safely stored before it presents a given cell. To show this sort of manipulation, we give below an IPL code for J72, which erases the list structure (0).

```
NAME   PQ  SYMB    LINK   COMMENTS
J72    40  H5             Save H5, since H5 safe over J's.
       10  9-10           Input subprocess; list is
                          already in H0.
           J101
       30  H5      0      Restore original H5 and quit.
9-10   70  9-11           Test if start of sublist:
           J75     J9     If yes, disengage head (marked
                          processed), and erase cell;
9-11       J9      J4     If no, erase cell and set
                          H5+ to continue.
```

Besides the H5 instructions, which occur because of the special convention for J's, the only noteworthy feature of this routine is the use of J75 in erasing a head. Cells with processed marks (heads) cannot be thrown away before the end of the generation, at which time J101 will do it if these cells are all that remain of the structure. J75 disconnects the heads from the structure and at the same time creates a new head which is unwanted and must be disposed of with J9.

A code for J101 is given in Fig. 11. The key problem, as usual, is how to keep the necessary information. J101 has two lists: a record list, which is used to keep the names of all sublists; and a generation list, which is used to keep the current location in the list structure, plus the ungenerated sublists. The name of the record list is kept in W0. As each new sublist is found, it is marked processed and its name put on the front of the list. This is accomplished by putting the name in the head and pushing down the record list, rather than by an insert. All this occurs in Process new sublist. At the end of the generation, in Clean up, each sublist on the record list is popped up. If the sublist is still there, then this simply unmarks it. If the sublist has been erased, there will be no cells after the cell carrying the processed mark (recall the J75 in the code for J72). The pop up will leave the cell as a private termination symbol and the cell is erased with a J9.

There are two points of access to the generation list. The name is kept in W1. Its head contains the name of the next cell to be generated. Just before the subprocess is executed, the next cell is found and put into the head (see Find next cell.) All the other sublists found to date are also in the generation list in order of generation. Thus, when one sublist is finished (either J60 in Find next cell fails to find another, or J131 in Diagnose current cell

```
NAME   PQ SYMB   LINK   COMMENTS
J101   10 W2            Set up:
          J17
          J105              Bring list in from auxiliary, if
                            there.
          J90               Create generation list (and record list,
                            below):
       40 HO                    WO holds name of record list.
          J90                   W1 holds name of generation list.
          J22                   W2 holds location for inserting
                                on generation list.
          J4
       40 H5                 Set H5+ and save for initial subprocess.
       40 HO      9-1        Pushdown to match entry; jump into
                             middle of loop.
9-4    40 HO            Find next cell:
          J60               Current cell to be generated
                            is already in HO.
       21 W1             Place name of next cell in head of
                         generator list.
       70 9-2     9-9    Test if current cell is end of sublist.
9-2    31 W1            Obtain next sublist.
9-9    30 H5            Execute subprocess:
          J18            Pop up H5 to set for subprocess.
       70 9-3            Test if subprocess says to quit (H5-).
       11 W1            Diagnose current cell:
          J80            Find symbol in cell (if none,
                         generation list termination).
       70 9-3
       40 HO
          J133           Test if marked processed.
       40 H5             Save sign for output to subprocess.
       70         9-4
       40 HO
          J131           Test if a data term.
       70         9-2
       12 HO
          J133           Test if symbol in cell already processed.
       70         9-4
       12 HO
          J132           Test if symbol in cell is local.
       70 9-4
```

Fig. 11--J101

```
NAME   PQ SYMB  LINK   COMMENTS
       11 W1           Process new sublist:
          J60              Test if only one sublist.
       70       9-10
       20 W2            Reset insert location.
9-11   41 W2            Add a cell to the end of the genera-
                        tion list.
       11 W2
          J60           Advance insert pointer to new cell.
       20 W2
       41 WO            Add a cell to the beginning of the
                        record list.
       12 HO
9-1       J137          Mark cell processed (entry point for
                        main list).
       61 WO            Put sublist name in record list (in
                        head).
       21 W2    9-4     Put sublist name in generation list
                        (at end).
9-10   30 HO    9-11
9-3    11 W1           Clean up:
          J71           Erase remaining fragment of generation
                        list.
9-8    11 WO            Clean up record list:
          J80               Test if end of record list.
       70 9-5
       31 HO                Pop up sublist, getting rid of
                            processed mark.
          J80               Test if sublist was erased (cell
                            now termination).
       70 9-6
       30 HO    9-7          (Discard unwanted symbol from
                             J80 on + branch).
9-6    12 WO
          J9                Erase cell for non-existing sublist.
9-7    31 WO    9-8      Obtain next sublist on record list.
9-5    11 WO
          J9    J19          Erase head of record list; quit.
```

Fig. 11--(Continued)

shows the cell to be a data term), the next sublist is found simply by popping up the generation list. This is done with a 31W1 in Obtain next sublist. The name of the last cell on the generation list is kept in W2. This is where newly found sublists should be added to the generation list for "print order."

The initial setup creates the two working lists. It also uses J105 to bring in the list structure from auxiliary storage if it is there. If the list structure is already in main storage, J105 does nothing. (Notice that the setup enters the main loop at the lower end of 9-1.)

J101 is some 62 instructions long, and seems like a very large routine. It accomplishes a good deal in the way of generality, however: handling multiple occurrences; allowing data terms; finding the next cell prior to presenting the current one; bringing in the list from auxiliary; and following a particular order of generation. Each of these adds instructions to the code. By way of contrast, if we eliminated most of these features we would have a (still useful) routine, defined as follows:

R10: Generate all the cells in a tree (a list structure that allows sublists to occur only once) containing only symbols (no data terms). The order of generation will be sublist first order: the cells of each sub-tree will be generated immediately after the cell that holds the name of the sub-tree. H5 will be set + to the subprocess if a new sublist is starting, and set - otherwise.

```
NAME   PQ SYMB  LINK   COMMENTS
R10    10 WO          Set up:
          J17
          J90             Create generation list.
       20 WO              Store generation list in
                          WO.
          J4              Set initial H5+.
9-3    61 WO              Put cell name in generation
                          list.
          J18          Execute subprocess:
       70 9-1             Test if quit from subprocess.
       12 WO          Test if cell holds sublist:
       12 HO
          J132            Test if local symbol in cell.
       70 9-4
       52 HO          Set up new sublist:
       41 WO    9-3       Push down generation list,
                          saving current cell.
```

```
9-4       J60          Find next cell:
      70 9-2
         J3      9-3       Set H5- to subprocess.
9-2   51 WO          Obtain cell in prior sublist:
         J68               Recall prior cell.
      70 9-1
      12 WO       9-4       Get prior cell (already
                            generated).
9-1   11 WO          Clean up:
         J71      J19       Erase generation list.
```

This routine has only 23 instructions, about one-third the number of instructions in J101. It also shows very clearly the form of this kind of routine.

When constructing your own generators, it is permissible to establish private conventions about communication between the generator and subprocess as long as such conventions do not conflict with the standard conventions (given in § 7.3 of Part Two). Thus, for example, the subprocess could use H0, or even some private regional cells reserved for this purpose, to communicate information back to the generator, if this proves useful. The following is an example which demonstrates the freedom to add other conventions as well.

R78: Repeat the generation of the list cells of list (1) for subprocess (0) until signaled to stop by the subprocess setting H5- (stop immediately) or setting the symbol one-down in H5- (stop after the current iteration through the list is finished). R78 signals the start of a new iteration through the list by setting H5+ to the subprocess on the first list cell, - on all others.

```
TYPE  NAME  PQ SYMB  LINK  COMMENTS

      R78   10 W1          J17 removes subprocess name
               J17         from H0 and preserves W0, W1.
            60 W1          W1 holds name of list (1).
                           W0 holds current location
      9-0   20 W0          on list.
```

TYPE	NAME	PQ	SYMB	LINK	COMMENTS
			J4		Set H5+ and make sure it is 3 deep. The symbol 1 down in H5 is plus to signal sub-
		40	H5		process that this is lst cell.
		40	H5		It is 1 down in H5 to protect it from J60, below.
	9-1	11	WO		Get next list cell and save
			J60		it in WO, as well as in HO
		60	WO		for subprocess.
		70	9-10		If end of list, go to 9-10;
		30	H5		otherwise, pop H5 to signal
			J18		if this is first cell, and execute subprocess via J18. H5 is 2 deep.
		70	9-20		Quit immediately if subprocess set H5-;
			J3		otherwise, set H5- for sub- process since next cell will be list cell, and protect this signal from
		40	H5	9-1	J60 above by preserving H5.
	9-10	30	H5		At the end of list, check 1 down in H5 to see if sub- process wants to continue.
		70	9-20		If not, clean up and quit via J19.
		30	H5		If continuing, restore H5 to be 1 deep,
		11	W1	9-0	get name of list from W1, and start another iteration through list, at 9-0.
	9-20	30	H5	J19	Restore H5 to original depth, quit.

9.1 PROBLEMS

NOTE: In constructing the following generators, follow the standard conventions.

PROBLEM 57

Code Q9--"Replace every occurrence of variable (1) in the algebraic expression (2) with the variable (0). Q9 sets H5- if no replacements occurred, sets H5+ otherwise." Assume that (2) names a general list structure like L11 of Problem 33 (p.41), and use J101 to generate it.

PROBLEM 58

Code Q10--"Delete all operators in the algebraic expression (0)." Assume (0) names an expression like L11 of Problem 33 (p. 41), and use J101. Note that great care must be taken in modifying list structures when a generator is involved, since the generator in general does not know what manipulations are taking place. In this problem, J101 always obtains the name of the next cell before outputting the name of the current cell to H0 for the subprocess. If the next cell is erased by the subprocess and returned to available space, there is bound to be trouble.

PROBLEM 59

Code the generator Q11--"Generate the names of the list cells of list (1) for subprocess (0)." Q11 does not generate the head cell of the list. Do not use J101 or J102. Q11 has the name of the next cell saved before handing the name of the current cell to the subprocess.

PROBLEM 60

Code the generator Q12--"Generate the names of the list cells of list (1) for subprocess (0). Set H5+ to the subprocess if the cell being output to the subprocess contains a local symbol; otherwise, set H5- to the subprocess." Q11 should be used in the construction of Q12. (This does not mean to cannibalize or modify the code for Q11 to produce Q12; it means that the code for Q12 should contain an execution of Q11.)

PROBLEM 61

Code the generator Q13--"Generate the symbols of list (1) two symbols at a time for the subprocess (0). The first of the pair of symbols output to the subprocess is in H0, the second is one-down in H0. The symbol in the head of the list is not generated." Q13 is considered to have run to completion when it cannot find two more symbols on the list. Do not use any other generators in constructing Q13.

PROBLEM 62

Code the generator Q14--"Generate every even numbered symbol from list (1) for subprocess (0). The symbol in the first list cell is symbol number one." The code for Q14 should contain an execution of Q13.

PROBLEM 63

Code the generator Q15--"Generate the names of corresponding cells of list (1) and (2) for the subprocess (0), until one of the lists is exhausted. When outputting to the subprocess, H0 holds the name of a cell from the input list (1), and one-down in H0 is the name of a cell from the input list (2)." Q15 is called a parallel generator for obvious reasons. Observe that this cannot be coded using two J100's "in parallel"; this is one of the few limitations in how generators can be combined.

PROBLEM 64

Code the generator Q16, which has the same definition as R78 in this section, except that the subprocess uses the cell R0 instead of the second cell of H5 to signal whether another iteration through the list is desired." Q16 should preserve R0 sometime before the first execution of the subprocess and restore it before terminating via J19. Q16 should be shorter than R78.

PROBLEM 65

Code the generator Q17--"Generate the names of the cells which contain operator symbols in the algebraic expression (1) for the subprocess (0)." Q17 assumes the algebraic expression is represented by a general list structure like L11 of Problem 33 (p. 41). Use J101.

PROBLEM 66

Code the generator Q18--"Generate the location of the variable (1) in the algebraic expression (2) for the subprocess (0)." Q18 assumes the algebraic expression is represented by a tree like L10 of Problem 33 (p. 41). Use J102.

PROBLEM 67

Code Q19--"Replace every occurrence of the variable (1) in the algebraic expression (2) with a locally named copy of the subexpression (0)." Use Q18. The subexpression (0) is guaranteed not to contain an occurrence of the variable (1). The algebraic expression (2) is like L10 of Problem 33 (p. 41).

PROBLEM 68

Code D99--"Delete all operator symbols in the algebraic expression (0)." Use Q17 of Problem 65 above. The expression (0) is like L11 of Problem 33 (p. 41). Note the remarks in Problem 58 (p. 85) on use of generators for deletion.

10.0 LINE PRINTING

J150 through J153 are primitives for printing structures, lists, symbols, and data terms in a standard vertical format. For composing and printing horizontal lines of information, the primitives J154 through J161 are provided. (Detailed definitions of these primitives and conventions for using them are given in §§ 16.1, 16.2 of Part Two.)

As an example, we develop the coding to print a sentence and a row of numbers horizontally, on separate lines. The sentence is represented by L15, a list of BCD data terms. The data terms themselves contain blanks to separate the words of the sentence, so there is no formatting to be done by the program. The numbers are represented by L16, a list of integer, octal, and floating point data terms. The program introduces formatting by leaving five blanks between adjacent numbers. Neither L15 nor L16 are long enough to produce several lines of printing in this case, but the code handles lists that do. The routines are described as follows:

R80: Generate the data term names from list (1) for the subprocess (0). Clear the print line 1W24 before beginning generation and print the line when generation terminates.

R81: Subprocess for printing sentences. Enter the data term (0) into the print line 1W24. If line 1W24 is full, print it, and then enter data term (0).

R82: Subprocess for printing rows of numbers. Skip five columns (blanks) and enter data term (0) into print line 1W24 if room for it; if not, print the line, then skip five columns and enter data term (0).

R83: Executive routine for this example. Enter and print the sentence L15, then enter and print the list of numeric data terms, L16.

TYPE	NAME	SIGN	PQ	SYMB	LINK	COMMENTS
5						Routine header.
						R80: Generate list (1) for subprocess.
	R80			J154		Clear the print line 1W24.
				J100		Generate list (1) for subprocess (0).
				J155	0	Print the line 1W24.
						R81: Subprocess for entering a data term into line 1W24 without any formatting.
	R81		60	9-30		Save data term name, in case line is full.
	9-10			J157		Enter data term left-justified at column 1W25, if there's room.
			70		0	Quit, with H5+, if there was room.
				J155		Otherwise, print full line
				J154		and then clear it, resetting 1W25.
			11	9-30	J157	Now enter data term at left margin.
	9-30	+		0	0	9-30 holds name of data term.
						R82: Subprocess for entering data term (0) + 5 blanks.
	R82		60	9-30		Save data term name, in case line is full.
	9-10		10	9-40		Tab 5 columns (same as
				J161		entering blanks).
				J157		Enter data term (0) left-justified at column 1W25, if there's room.
			70		0	Quit subprocess with H5+ if there was room.
				J155		Otherwise, print full line and
				J154		Clear it, resetting 1W25 to left margin.
			11	9-30	9-10	Now get data term, tab 5 columns, and enter it.
	9-30	+		0	0	9-30 holds name of data term.
	9-40	+	01		5	9-40 is number of columns to tab.
						R83 (Executive): Print line L15 as a sentence, then print list L16 as a row of numbers.
	R83		10	L15		Input data,
			10	R81		input format routine, and
				R80		fire generator.

```
TYPE  NAME   SIGN   PQ  SYMB   LINK   COMMENTS
                    10  L16           Input data,
                    10  R82    R80    input format routine, and
                                      fire generator.
5                   01                Data header.
      L15               0             L15 represents a sentence as
                                      list of BCD data terms.  Each
                        9-1           word will be separated from
                        9-2           next by 1 blank when printed.
                        9-3           Note that two blanks follow
                        9-4    0      word THIS because trailing
                                      blank in 9-1 will be eliminated
                                      on entry into print line.
      9-1    +      21  THIS
      9-2    +      21   IS A
      9-3    +      21   SENT
      9-4    +      21  ENCE.
      L16               0             L16 is list of numeric data
                        9-1           terms.
                        9-2
                        9-3    0
      9-1    -      01          33     Integer data terms = -33.
      9-2    -      11  33     -29    Floating point -.33 to the -29th.
      9-3    -      31  77777  77777  Octal data term -7777777777 (octal).
5                       R83           Start at R83.
```

10.1 PROBLEMS

PROBLEM 69

Assume that L17 is a list and that each symbol on L17 names a list like L15 above; i.e., L17 is a list of sentences. Code R84--"Print the sentences named by the symbols on L17. Each sentence should not start on a new line, but should be separated from the preceding sentence by two blanks." Use R81 and J100, but not R80 above. The two blanks between sentences must be provided by the program; they are not contained in the first data term of each sentence.

PROBLEM 70

Construct the generator R85--"Generate only the integer data terms of list (1) for subprocess (0)." List (1) is assumed to be like L16 above. Code the subprocess R86--"If data term (0) is positive, tab five columns and enter the data term into line 1W24. If (0) is negative, tab five columns and enter the absolute value of the data term and enclosing parentheses into line 1W24. Assume that there will never be enough data to fill the line." Then code R87--"Each symbol on L18 names a list like L16 above. Print the integer data terms on each list on a separate line." Use R85 and R86.

PROBLEM 71

Code R88--"Print the symbols on list (0), preceding each symbol with three blanks. Print the name of the list (0) first, enclosing it in parentheses." The list may produce several lines of printing.

11.0 LINE READING

J154 through J161 allow one to build lines of text from symbols and data terms; J180 through J189 are primitives that allow one to build symbols and data terms from lines of text. These primitives are described in § 22.0 of Part Two. Below, we use some of these primitives to read an English sentence from a card and to represent the sentence as a list of BCD data terms, exactly like list L15 of the previous section.

R95: Read a card into line X1 and create a list of local BCD data terms from the line, starting from the leftmost non-blank column. Output (0) names the list, and H5 is set + . If no card is present, or if the card is blank, there is no output and H5 is set - .

(The card is assumed to contain one English sentence, with one blank separating each word. We wish to be able to print this sentence later by entering the data terms of list (0) into a print line via J157. Since J157 eliminates trailing blanks, R95 will test the rightmost character of each data term. If this character is a blank, it will be inserted as the leftmost character of the next data term, which will insure that a blank will appear between the words when list (0) is printed with J157.)

TYPE	NAME	SIGN	PQ	SYMB	LINK	COMMENTS
2	B				100	Define Regional Symbols.
2	M				100	
2	N				100	
2	R				100	
2	X				100	
3	X1		01		80	Print line X1, 80 char. long.
5			01			Data header for constants.
	M1	-	01		1	M1 = Minus one.
	N1	+	01		1	N1 = Plus one.
	N5	+	01		5	N5 = Plus five.
	N80	+	01		80	N80 = Plus eighty.
	B1		21			B1 = BCD BLANK.

```
TYPE  NAME   SIGN  PQ SYMB   LINK   COMMENTS
5                                   Routine header.
      R95          10 X1
                      R96           Set up to read into line X1.
                                    Read a card from unit 1W18
                      J180          into X1.
                   70 R97           R97 cleans up and quits, H5-
                                    if no card present.
                   11 W25
                      J184          Set 1W25 to 1st non-blank col.
                   30 HO
                   70 R97           Quit if card was blank.
                      J90           Create output list (0) and
                      J50           save it in WO.
      9-10         10 B1
                      J90           Create a cell, make it
                      J121          into a blank BCD data
                      J136          term with a local name.
                      J182          Fill data term with 5 char.
                                    of line.
                   70 9-200         To 9-200 if data term all blanks.
                   11 WO
                      J6
                      J65           Add data term to end of output
                   11 W25           list.
                   10 N80           N80 is integer data term = 80.
                      J116
                   70       9-300   To 9-300 if 1W25 = 80 or more
                                    (Done).
                   10 M1            M1 is integer data term = -1.
                      J161          1W25 points to last char. of
                      J186          data term.
                   70 9-10          Final blank of this D.T. will
                                    be initial char. of next
                                    data term.
                   50 N1            N1 is integer data term = 1.
                      J161  9-10    Reset 1W25 and loop.
      9-200           J9            Erase blank data term.
      9-300        11 WO            Name of output list to HO.
                      J30           Pop WO.
                      R97   J4      Clean up print line controls,
                                    set H5+.

5                                   Routine header.
1                                   R96:  Set up to read from
1                                         line (0).  Preserve W21,
1                                         W24, W25, W30.  Set 1W21,
1                                         1W25 = 0.  Set 1W24 = (0).
1                                         Set 1W30 = 5.
```

```
TYPE  NAME  SIGN  PQ SYMB  LINK  COMMENTS
      R96         40 W24
                  20 W24        1W24 = (0).
                  40 W21
                  40 W25
                  40 W30
                     J90        Create and
                     J124       clear data term.
                  20 W21        1W21 = 0.
                     J90
                     J124
                  20 W25        1W25 = 0.
                  10 N5
                     J120       Create copy of N5
                  20 W30        for W30 and
                     J154  0    clear print line.
1                                R97:  Clean up after reading.
1                                      ERASE 1W21, 1W25, 1W30.
1                                      Pop W21, W24, W25, W30.
      R97         11 W21
                  11 W25
                  11 W30
                     J9         ERASE data terms.
                     J9
                     J9
                  30 W21
                  30 W24
                  30 W25
                  30 W30   0
5                    R95        Start card for reading.
      THIS IS A SAMPLE SENTENCE FOR R95 TO READ.
```

11.1 PROBLEMS

PROBLEM 72

Code Q20--"Read a card from unit 1W18 into print line X1. The card is either blank or contains a series of decimal integer numbers separated from each other by one or more blanks. The number of digits in each number varies from one to eight. Create a locally named data term for each number on the card and read the number into it. Create and output the internally named list (0), which names the data terms, and set H5+. If the card is blank, set H5- and do not create a list." Q20 assumes the first number may start in any column of the card. Treat W21, W24, W25, and W30 as safe cells as usual, preserving them before use, restoring them afterwards.

PROBLEM 73

Code Q21, whose definition is the same as that of Q20 except that the series of numbers extends over several cards. All the numbers should be added to the same output list (0). The end of the series of numbers is signaled by a blank column followed by the character slash (/) followed by all blanks to the end of the card. Modify the code for Q20 so it is a suitable subroutine doing most of the work for Q21.

PROBLEM 74

Code Q22, whose definition is that of R95 except that the sentence extends over several cards. The end of a sentence is signaled by a period (.) immediately following the last word, with no intervening blank; a period will not occur anywhere else in the sentence. All columns following the period are blank, and the period should be included in the final data term. Words are complete on one card, and the first word on each card begins in column 2 or later.

PROBLEM 75

Code Q23--"Generate the words of the sentence in print line (1) for the subprocess (0). The output of the generator to the supperroutine is H5+ if words were encountered and the generator ran to completion; H5- if no card was present, or if the card was blank, or if the subprocess turned the generator off before it ran to completion. The output of the generator to the subprocess is an internally named list in HO. The list has an empty head and contains the locally named BCD data terms needed to represent the characters in the word. Only the last data term on the list may contain blanks, from 0 to 4 of them. Print line (1) is 80 characters long. The sentence begins in column 1 or later, and is terminated by a period, as in Problem 74 above; the period should <u>not</u> be included in the last data term of the last word.

12.0 SAVING FOR RESTART AND RESTARTING

When very large programs have been constructed in IPL-V, it is common practice to load the program and data into the computer from the symbolic cards only once, then save the contents of memory on tape for fast restart on subsequent runs. J166, Save on Unit (0) for Restart, is used for this purpose. (0) is the name of an integer data term whose value designates one of the IPL tapes, 1 through 10. The program does not stop after executing J166, so a normal run can be executed after saving for restart. When making subsequent runs from a restart tape, the programmer usually wishes to make corrections to existing routines or data structures, or to add new ones. Thus, the normal practice is to save for restart by executing J166, followed immediately by an execution of J165, Load More Routines and Data. For example, in the program of the previous section involving R95-R97, if the following deck of cards were to replace the Type-5 start card, a restart tape would be created on IPL Unit 1 by the execution of J166 in R99. Then, J165 in R99 would encounter the start card for R95 and a normal execution of the problem would occur.

```
TYPE  NAME  PQ SYMB  LINK  COMMENTS
5                        Routine header.
                         R99:  Save for restart on
                         unit 1, then load more
                         routines and data from 1W18.
      R99   10 N1        N1 = Integer data term = 1.
               J166  J165
5             R99        Start at R99, Save.
5             R95        Start problem, R95.
```

The following deck is all that would be required, along with the restart tape, of course, to perform subsequent runs of R95.

```
TYPE  NAME   PQ SYMB  LINK  COMMENTS
5            40 7           Restart from tape on Unit 7.
5               R95         Start problem at R95.
    THIS IS A SENTENCE FOR R95 TO WORK ON.
```

The Type-5 card, with P = 4, is encountered by the initial load process, causing memory to be reloaded from the IPL tape named by SYMB; interpretation begins with the instruction following the J166 which created the restart tape--in this case, with J165. J165 immediately encounters the start card for R95 and executes the problem normally. Generally, we will have corrections to make, as in the deck below, and these corrections will be loaded by J165 if placed between the Type-5 restart card and the Type-5 start card.

```
TYPE  NAME   PQ SYMB  LINK  COMMENTS
5            40 1           Restart from tape on Unit 1.
5            04             Routine header--corrections.
      R97    30 W24   0     R97:  Pop W24 only.
      2222   20 W24   0     Terminate R96 at 2nd
                             instruction.
5               R99         Start card to save original
                            plus corrected code onto Unit 1.
5               R95         Start problem.
 THIS IS ANOTHER SENTENCE FOR R95 TO PROCESS.
```

The deck above demonstrates two ways of making corrections: by replacing an entire routine or data structure; or by making an absolute patch at the proper place. The old, and longer, version of R97 is completely replaced by the one line above. The cells making up the original R97 are lost to the system. The correction to R96 at cell 2222 demonstrates patching at a known location, obtained by examining the original assembly listing of R96. Note that the routine header must have Q = 4 if absolute internal symbols are to be used. (See § 18.5 of Part Two for discussion.) Most people prefer to make corrections by correcting the original symbolic deck for the routine

and reloading the entire routine, as R97 was above, since this simplifies the problem of keeping the symbolic deck updated. The old version of R97 is erasable by J201 if desired.

The deck above makes changes to R96 and R97, then immediately makes a new restart tape which includes these changes. One usually prefers to test the changes before this is done, and it is certainly risky to destroy the previous restart tape until the new one has been tested at least once. An inopportune hardware failure could produce a useless restart tape and thus force a reassembly of the complete symbolic deck if the new restart tape has been written over the old one. The new restart tape may be tested in the same run which created it merely by inserting an appropriate restart card in the deck immediately following the start at R99.

There is another use of J166 worth mentioning. When a program runs a long time on the computer, it is usually subject to termination by the computer operator through a signal from the console. In this case, we would like to create a restart tape which will allow us to resume exactly where we were interrupted, without having to supply a start card on restart. The solution is to plant the name of a routine, say R98, in W14, which will only be executed when the interrupt signal from the console is detected. R98 could be exactly like R99 above except that the final LINK would be zero rather than J165. Restarting from a tape thus created requires only a restart card, and interpretation resumes at the point of interruption. Since details of the interrupt and restart procedure may vary on different computers, the conventions of the local installation should be checked. It is worth noting that H5 is set + when J166 is executed, but that H5 is set - by the system when a run is restarted. Thus, in the code following the occurrence of a J166, it is possible to detect whether one is continuing the original run or doing a rerun. See the example in § 20.0 of Part Two, SAVE FOR RESTART.

12.1 PROBLEMS

PROBLEM 76

Code Q24--"Save on Unit 2 for restart, with provision for making corrections on restart."

PROBLEM 77

Make up the deck and run the code in the example of R95-R97, using Q24 to save for restart before starting the problem, R95. Use the restart tape and do a second run with the definitions of R96 and R97 changed, as follows:

R96: Preserve W24 and set it to (0). Preserve W25, create a new data term and put its name in W25, then clear print line 1W24.

R97: Return 1W25 to available space, then pop W24 and W25.

Make these changes by replacing R96 and R97, rather than by patching them. Erase the old versions of R96 and R97 with J201 in order to recover the list cells they occupied.

PROBLEM 78

Code Q26--"Save for restart on Unit 5, without provision for corrections on restart." Check the conventions of your local installation and test Q26 to verify that it does indeed provide interruption and continuation of long problems without using a start card on restart.

13.0 DEBUGGING

The primary debugging facilities of IPL-V are the trace, the snapshot cells, the post mortem, and the error trap (see § 15.0, MONITOR SYSTEM, and § 21.0, ERROR TRAP, in Part Two). Other uses can be found for some of these mechanisms, but we shall consider them here only in terms of their utility for checking out a program. A common philosophy underlies these various facilities: namely, (a) one should be able to preserve a logical and physical separation between the program being debugged and the code for debugging it; and (b) the debugging code should be capable of using the full power and generality of IPL-V.

The trace is simplest to use. If the program is small and the run will be short, all of the routines can be traced simply by setting the external trace mode, 1W31, to FULL TRACE initially. None of the routines need have monitor points (Q = 3) in them. However, this option should be used intelligently; small routines with small bugs can produce reams of useless trace under this mode of operation. One can guard against this by using local installation measures either to limit the output to a specified number of pages, or to limit the running time. IPL measures are also available to prevent runaway tracing. W33 can contain a limiting cycle count which will cause trapping; providing J7 as the trap action will terminate the run. When running with FULL TRACE, the number in W33 is also the limit to the number of lines of trace that will be produced. Or, 1W14 can name a terminating routine (e.g., J7) that is invoked by the external interrupt mechanism.

For large programs, setting 1W31 to FULL TRACE for the entire run is unacceptable. However, there will be occasions when it may be useful to obtain full tracing of all the routines executed between two points in the

program. If these points can be identified by approximate cycle counts, then the H3 trap is an effective mechanism. In the example below, W33 is set to cause trapping when 1999 interpretation cycles have occurred; the trap action routine D1 is then executed. D1 causes FULL TRACE to be turned on, sets W33 so it will trap again 1000 cycles later, and arranges that the trap action at that point will set the external trace mode back to its normal SELECTIVE TRACE value. The effect is to produce a trace of every instruction executed between the two trapping points; the names of the routines that would be traced need not have been known beforehand.

```
TYPE  NAME   SIGN   PQ SYMB   LINK   COMMENTS
5                                    ROUTINE HEADER.
      W33    +      01         2000   H3 trap at 2000 cycles.
1                                    EO is a debugging execution.
      EO            10 W26            Set the trap action of H3
                    10 D1             to be D1.
                    10 H3
                       J11
                       R1     0       Execute R1, the program
                                      being debugged.
      D1            10 9-1            Set 1W31 = 1, FULL TRACE.
                    11 W31
                       J121
                    50 9-999
                    10 W33            Set W33 to trap after
                    40 HO             another 1000 cycles.
                       J110
                    50 W26            Action for next trap is D2,
                    10 D2             which will return 1W31 to
                    10 H3             SELECTIVE TRACE.
                    03 J11    0       Cause MONITOR POINT.
      9-1    +      01             1
      9-999  +      01          1000
      D2            10 9-2
                    11 W31
                       J121           Set 1W31 = 2, SELECTIVE TRACE.
                    03 J8     0       Cause MONITOR POINT.
      9-2    +      01             2
```

A useful variant of this technique would leave the trace in the SELECTIVE TRACE mode, but cause specified routines to trace only if executed during the 1000-cycle

interval. This can be done by having the first trap action routine mark the specified routines to trace with J147, and having the second trap action routine undo the trace mark with J149. This method might be used when known routines are suspected, but are used too frequently to allow tracing each time--routines which seem to perform correctly up to a certain point in the run, and then appear to misbehave.

Since the system only examines the external trace mode at monitor points, each of these routines should include a monitor point after changing 1W31, in order to make the change effective immediately.

The snapshot mechanism is triggered by the occurrence of monitor points in the executed code. The snapshot cells, W12 and W13, name routines which are executed before and after (respectively) the execution of the routine containing the monitor point. This mechanism can be used to provide the equivalent of a "data trace"; i.e., selected lists can be printed to observe the changes which the program is producing in them. The routines named by W12 and W13 should not be traced, but are otherwise unrestricted.

Each of the routines which are activated by the error trap, the snapshot cells, or the post mortem routine cell, W15, may be used to control the course of debugging in a dynamic fashion; in particular, they may create or destroy monitor points, control the trace mechanism via 1W31, cause trapping on the cycle count 1W33, execute post mortems, change the contents of the snapshot cell, etc.

13.1 PROBLEMS

PROBLEM 79

Code the debugging routines and data needed to cause each of the routines R80 through R83 (§ 10.0, pp. 87-89) to be traced for their first execution only. If monitor points are needed, create with J147.

PROBLEM 80

Code the debugging routines needed to produce a snapshot of the output list being built by F1 (§ 2.0, pp. 17-22) each time a new symbol has been added to the list, and at no other time. No trace output is desired. (The J routines may be marked to trace.)

PROBLEM 81

Code the debugging routines needed to produce a snapshot of list L1 after each execution of a routine R1, a snapshot of L2 and L3 before each execution of R2, and a post mortem every 100,000 interpretation cycles. No tracing is desired.

PROBLEM 82

Code the debugging routines needed to cause a routine R1 to trace only on its tenth execution. Assume that other routines are being traced selectively in the same run.

14.0 ORGANIZING COMPLETE TASKS

The important concepts of IPL-V have been presented in the earlier sections and the meaning of most of the primitive processes has been illustrated by using them in solutions to the problems. These illustrative problems have called for coding precisely-defined routines, so that there has been little opportunity for the reader to attempt to organize complex tasks. A few guidelines on the larger issues of organization are offered below, even though of necessity our comments are somewhat general. The problems at the end of this section are more complex than those in the earlier sections and provide an opportunity to try out some of these organizational ideas. However, they can be no substitute for organizing and constructing programs to handle one's own problems.

The transition from the small problem, which is simply an exercise in coding, to the large problem, involving all the skills of the programming art, introduces several new concerns. First, a large program must be organized into parts. These subdivisions are not given, but must be created by the programmer. Second, the scheme for representing the data of the problem is not given, but must be chosen by the programmer. Finally, a big system will not be created once and for all, but will evolve; that is, parts of it will require modification while other parts still retain their original form. Needs for modification arise because understanding about what the program really should do and how it might do it increases slowly, and because demands arise during construction for the program

to perform additional functions. Thus, the initial organization must be chosen with an eye to the unknown future.

Notably absent from this list is concern about how well defined the task is. Programming does not usually extend to defining the task; and, many tasks that are extremely well defined pose very large design problems of the kinds we have just mentioned. For example, implementing any programming language (such as IPL-V) on a new machine poses major problems of program organization, data representation, and preparation for future modification, despite the fact that the language is completely defined and already runs on other machines.

14.1 PROGRAM SUBDIVISION

With current programming languages, and especially with IPL, any large program will be organized as a hierarchy of subroutines. The alternatives to hierarchical organization are not very well understood, and we shall not consider them here. The problem for the programmer, then, is how to get the most out of a hierarchical organization.

One programming strategy, often called the "top-down" approach, is to divide each large process, no matter how complicated, into a small number of subprocesses. Each of these subprocesses is given a name, and its function--the processing it accomplishes--is defined precisely by specifying exactly what inputs it requires and what outputs it produces. How this subprocess will carry out this processing does not matter at this stage, although it may

be necessary to make some decisions about data representation at the same time that subprocesses are defined.

When all its subprocesses have been defined, it should be possible to write the code for the process itself. The routine defining the process will then consist entirely of instructions executing the subprocesses, instructions positioning inputs for the subprocesses and disposing of their outputs, and instructions using the outputs of some subprocesses to decide which subprocesses to execute next. (These latter decisions, if they are at all complex, should be packaged as additional subprocesses, of course.)

Once any process is coded, attention can be directed to developing and coding each of its subprocesses, using exactly the same strategy of decomposing these into subprocesses. Ultimately, subprocesses are reached that can be defined directly in terms of the IPL primitive processes, so that the decomposition comes to a stop. Although apparently at each stage all the complexities are being relegated to the subprocesses and only codes for trivial processes are being written, it will be found at last that nothing complicated remains to be done at the bottom of the hierarchy.

Following the top-down strategy provides a framework within which the programmer can concern himself with developing a subroutine hierarchy that is at once compact, flexible, efficient, and comprehensible. If the number of subprocesses at each stage is kept small, the code for each routine constitutes an immediately understood outline of the process accomplished by that routine, especially if appropriate titles and comments are added.

Such routine hierarchies are easy to read and comprehend to any desired level of detail.

14.2 CENTRALIZATION OF FUNCTION

One of the important general principles for designing program hierarchies is to centralize functions. That is, each function that is required in one or more processes should be performed by a single subroutine, insofar as possible. The advantages of doing so are many. Most obviously, space is saved by not duplicating the same segment of code in the total program; likewise, programming time is saved by not having to code essentially the same routine more than once. By far the most important advantage, however, is the flexibility achieved by centralization. Almost always, significant proposed program changes are initially described as modifications or additions of function (or of representation). If the functions are centralized, modification is usually quite easy; if they are not, modification may be impossible, requiring multiple changes throughout the program.

As an example, in one of the earlier versions of IPL the functions of getting cells from available space and putting cells back on available space were decentralized; each routine that called for space or returned cells to available space did the job itself in its own way. The rationale for so doing was that the amount of code required for these functions in any given routine was small and standardization would sometimes have doubled the amount of processing required to do it. At a later time in the development of the language design, it was

proposed to use many available space lists, analogous to the mechanisms in IPL-V for loading into separate blocks. This whole proposed line of modification had to be abandoned because of the decentralized scheme for handling available space--to change this scheme was tantamount to recoding the entire language.

A final advantage of centralization is that it increases the comprehensibility of the program. Some of this clarification comes about simply because the centralized scheme contains fewer subroutines. But, some of the gain is due to the fact that the centralized program structure matches well the way humans normally describe large operating structures; namely, in terms of functions.

On the other hand, there are costs associated with centralization. As suggested by the example above, centralization requires standardizing the manner of calling on the standard subroutines, and this increases the processing cost of subroutinization. In IPL, for example, a DESCEND process and an ASCEND process must be executed in the interpreter to enter and return from a subroutine. In this case, however, the cost of centralization is almost negligible in comparison with its advantages.

Working through the program from the top down, and accumulating a list of the routines as they are defined, is an excellent way to discover when the same function recurs in several places. Often, the routines for the several uses of the function are not identical; for example, several details of the input and outputs may be different. In this case, it is still possible to create a single routine, perhaps with additional parameters,

that will perform both jobs. The price of this standardization may be appreciable if much extra processing is required to interpret the additional parameters. However, the price is usually well worth it, for changing similar functions into identical ones by such modification is one of the main mechanisms for exposing the underlying structure of a task.

A concomitant principle to centralization is the separation of function. That is, each process should accomplish only a single function. The main advantage of this is to permit each process to be used in several parts of the total program. By the nature of things, processes composed of several functions are less likely to recur than processes incorporating only the separate functions.

There is a conflict between writing the program so that each function is performed by a corresponding routine and the principle of dividing each routine, no matter how complex, into a small number of subroutines. The subroutines at the top of the program hierarchy are often highly particular, corresponding to no familiar functional concept. They occur only once in the entire structure. Only as one works down the hierarchy of subprocesses does one encounter functions that recur at different points in the program.

Of course, the concept of function is not very precisely defined. There is considerable freedom in analyzing a task into functions. However, functional analysis appears to be a natural way for humans to cope with tasks, so that the principles just enumerated are useful guides to programming.

14.3 ISOLATION OF SUBROUTINES

Another principle may be called the principle of isolation. The flexibility in hierarchical organization depends on each subroutine being isolated from the rest of the program, except for a small number of well-defined connections. Only with such isolation is it possible, in making changes in other parts of the program, to know what consequences these have for a given routine; or conversely, in changing a routine to know that other routines are unaffected by the change. The connections between a routine and its environment should be describable solely in terms of the routine's inputs and of general conventions about the form of data representation.

Concretely, one subroutine should not link to another; rather, all subroutines should end by simply terminating. If two routines are to occur in sequence, a higher routine should be used to execute them. As another example, the information available to a subroutine about its inputs should be clearly stated and no other information about them should be used, even if the programmer possesses additional information (which he often seems to, due to the concrete context in which the routine is first defined).

This seemingly obvious principle is easily violated. For instance, a rather complex routine was once written to keep track of the space used in various structures of a program. To permit future expansion of the program, the routine took as input a list, each of whose sublists contained the names of all the data structures of a given type; thus, additions could be made to the lists. It

happened that some types of substructures required special processing. Since, at the time of coding, the input list was already available, the programmer made use of what he knew about the types of structures filed on that list and where they occurred--e.g., the third sublist contained structures of a certain type to be treated in a certain manner. As a consequence of these coding conventions, when the input list was finally changed (a year after it was coded) chaos resulted, since position on the list no longer corresponded to the positions assumed during the original coding. Ignoring the principle of isolation was not unmotivated; to have written the program correctly would have required having the routine determine at each stage what kind of structures were stored on the sublist being processed. This would have called for additional processing, as well as for reliable criteria by which the different types of structures could be identified. Consequently, real costs were associated with preserving independence.

In keeping programs independent, it is often useful to channel all inputs and outputs through HO. Whenever, for example, the W's are used as direct communication devices between routines (to avoid additional input and output actions), unintended connections between routines easily occur. The use of common "blackboards" by a whole system of routines is a useful organizing scheme, but one that takes a good deal of thought. To press the analogy, one man with an eraser can wreak havoc on a blackboard.

14.4 DATA REPRESENTATION

So far we have discussed only organizing the processing. Organizing the data--i.e., choosing representations of the information in the task--is perhaps even more crucial. Unfortunately, little can be said about it in general by way of guidelines.

Concern with representation leads to what is often called the "bottom-up" approach; that is, first creating a set of "standard" subprocesses, which are then used as the basic components of higher-level routines. Given a representation, various processes must be performed on the information so represented: reading information out and writing information into the representation. There may be a variety of these processing requirements, forming a collection of processes that recur again and again in any program utilizing the representation. Consequently, the programmer can attend to the design and implementation of these basic processes before he goes on to consider how the entire program will be structured.

Programming languages themselves provide excellent illustrations of the bottom-up approach. Consider lists in IPL-V, for example. Once the list was introduced as a representation of information, it was necessary to provide processes to read lists (J60 and J80) and to write lists. The latter process involves not only writing in an already existing list (21WO if the name of the list cell is in WO), but also creating lists (J90), modifying the structure of lists (J64 and J68), and erasing lists (J71). Since a useful set of basic processes, and not just a minimal set, was being provided, additional

specialized processes were also defined (e.g., J61, J62, etc.). IPL-V as a whole, of course, includes not only lists, but list structures, description lists, cells, push down lists, and data terms. Each of these requires additional reading and writing processes. Thus, for description lists, J10 reads; J11, J12, and J13 write and create; J14 and J15 erase. All these reading and writing processes are internal to the computer. In addition, reading and writing must occur with respect to the outside world; thus, we get J140 and J151 for reading and printing list structures.

By the time one takes care of the different representations, the different media in which reading and writing are possible, and the desire to build useful collections of processes and not just minimal ones, most of the basic primitives in IPL-V have been accounted for. A similar story holds true for other programming languages.

This story of the programming languages is repeated, although on a smaller scale, every time a new representation with distinctive conventions is created within a program. Building up the collection of reading and writing processes, including ones for printing and reading from cards in some convenient external form, is, in effect, constructing a small programming language. However, this language remains embedded in IPL-V, so that it amounts to considerably less than a completely new language.

The advantages of working up from the bottom of the program toward the top are several. Having a unified conceptual scheme for a complete set of processes for reading and writing leads to a much cleaner set of

routines than just coding each particular "read" or "write" when it happens to first be called for, as is the case in working down from the top. In addition, the new read and write processes form higher conceptual units that can be worked with directly in discovering how to code higher processes. Instead of having to think continually in terms of the detailed features of the representation, one simply thinks of "putting it there" and "getting it." Perhaps the most important advantage is in keeping a clean separation between the conventions of the representation, on the one hand, and the use of the information held in the representation, on the other. For example, having created a complete set of read and write operations, it is often possible at some later time to change the underlying representation completely without modifying any part of the program except the read and write processes themselves. This rather dramatic demonstration of program modifiability is seldom possible unless such care has been taken beforehand.

Occasionally, the bottom-up approach provides important information about the qualities of a proposed representation. If a representation is created when it is needed, only a few of the read or write operations are uppermost in the programmer's concern at the time. Although the ad hoc representation may be extremely good for these few operations (say reading information and writing it into an already constructed representation), when additional operations are needed (say deleting information from the structure), they may be difficult or even literally impossible to code. Coding up the full

collection of reading and writing processes at the outset can reveal this situation before large amounts of coding and analysis have been completed which have to be redone.

An additional issue often arises with representations that permit constructions, such as lists and list structures. After a data structure is no longer needed it has to be erased and returned to the available space list. If the cautions mentioned above have been heeded, there will be no difficulty about coding the erasing process; but, there may still be a real problem in deciding what routine should do the erasing. This is usually termed the problem of responsibility: The routine that is responsible for a structure knows when it is no longer needed and can erase it. In simple situations where a structure is created, used, and discarded all within a single routine, the issue of responsibility is easily settled. However, if various structures are created by routines that have no control over or knowledge about what other routines will be using these structures and what conditions will lead to their becoming superfluous, then some more elaborate set of responsibility conventions becomes necessary. Although it is hard to give general advice, the principle that functions should be centralized wherever possible suggests that responsibility rest in a single routine (often called the "garbage collector") that has some means of independent access to all structures of concern and some processes for determining when these structures have lost their usefulness. Schemes of this kind often require additional processing, but bring their rewards by removing a major problem once and for all from the concern of the programmer.

14.5 SYSTEM EVOLUTION

At the beginning of this section we noted three considerations that distinguish large complex problems from the small exercises in the Manual. Two of these, the organization of processing and the choice of representation, have already been discussed. The third is the evolution of the system. A large programming structure represents a considerable investment, not just in programming effort, but in the discovery of sets of conventions that fit together to permit the task to be performed. Programs almost always undergo repeated modifications which seek to preserve intact as much of the system as possible, while still permitting the program to run under more general conditions or getting additional useful output from it.

Almost the sole guide in preparing for this unknown future is to keep the program flexible. Most of the specific recommendations for flexibility have been discussed already. Flexibility is always worth more than one is prepared to admit at the moment of coding, if only because other criteria, such as speed and space, are easier to respond to. The only other guideline is one that is trivial--but often ignored. Each routine should be documented to indicate specifically what is assumed for each input and output, and to state succinctly the general conventions of representation and coding philosophy. It is this documentation that permits discovery of possibilities for modification of the program, and that allows the assessment of the consequences of changes for apparently remote parts of the program. Without adequate documentation,

it becomes progressively more difficult to introduce changes into a large, complex program without losing control over their indirect effects, and without a gradual accumulation of complexities that soon produce incomprehensible chaos.

14.6 PROBLEMS

PROBLEM 83

Design a different representation for the bridge hand example which does not use description lists. Recode R4, R5, and R6 of that example for the new representation.

PROBLEM 84

Each card in a poker deck has a suit (spades, hearts, diamonds, clubs) and a value (ace, king, queen, jack, 10, 9, 8, 7, 6, 5, 4, 3, 2).

a) Design an IPL representation for a card and for a poker hand of five cards.

b) For each of the following types of poker hand, write a routine to determine if a given poker hand is of that type:

   1) One pair (two cards of same value, others of different values).
   2) Two pair (two cards of one value, two cards of another value, fifth card of a third value).
   3) Three of a kind (three cards of same value, others of different values).
   4) Straight (five successive contiguous values).
   5) Flush (all cards of same unit).
   6) Full house (three cards of one value, two of a second value).
   7) Four of a kind (four cards of one value).
   8) Straight flush (both straight and flush).
   9) Royal flush (straight flush, highest value is ace).

c) Write an overall routine to evaluate a poker hand, assuming the existence of each of the routines in (b) above.

d) Sometimes a poker deck includes one additional card, the joker, which may be used in place of any card in evaluating the hand. Write routines to determine the type of poker hand when a joker is in the deck.

e) In some poker games, the dealer may specify some of the cards as wild cards (e.g., deuces wild). Any of these cards may be used in the same way as the joker in evaluating the hand. Write routines to determine the type of poker hand in this case.

PROBLEM 85

Devise an information retrieval and storage system for a university room assignment office, which stores a list of all the rooms on campus together with their capacities and the hours during which they are in use. The system should have three operators: S1, which accepts a room (0) and an hour (1), and marks that room "occupied" during that hour; S2 which accepts the same inputs and unmarks the room at that hour; S3, which accepts an hour (0) and a number (1) of students in a class, and outputs the smallest room available at that hour that can accommodate the class. If a room can be found, H5 is set + ; otherwise, it is set - , and there is no output.

PROBLEM 86

Design a representation of a chessboard and the white and black pieces, and code routines which will generate all the legal moves for a given piece at a given position on the board, taking into account the positions of the other pieces at the time.

PROBLEM 87

Design and code a program to solve crossword puzzles. There should be a dictionary of synonyms and antonyms, capable of retrieving several alternate responses to a definition. Assume definitions are single words, negated when the antonym is intended. Allow the puzzle form to be

rectangular with unused or blacked-out squares appearing where needed. Provide a convenient way to add to the dictionary, to specify new puzzles, and to print the solution in crossword format.

PROBLEM 88

Design and code a program which would be useful to the instructor of an IPL course for checking the solutions to the problems in this Manual. Make some reasonable assumptions about the required format for submitting decks, whether the program or the student should provide test data, maximum running time for each solution, safeguards against clobbering the system, criteria for evaluating a solution, providing useful diagnostic information to the student, etc. State these assumptions explicitly on comment cards.

SAMPLE SOLUTIONS TO SELECTED PROBLEMS

```
PROBLEM 1 **********************

HEAD CELL IS EMPTY.                          X1      0
SYMBOL FOR SUNDAY.                                   D1
SYMBOL FOR MONDAY.                                   D2
SYMBOL FOR TUESDAY.                                  D3
SYMBOL FOR WEDNESDAY.                                D4
SYMBOL FOR THURSDAY.                                 D5
SYMBOL FOR FRIDAY.                                   D6
SYMBOL FOR SUNDAY, END OF LIST.                      D7      0


PROBLEM 2 **********************

HEAD CELL IS EMPTY.                          X2      0
SYMBOL FOR 'THE'.                                    T1
SYMBOL FOR 'MORE'.                                   M1
SYMBOL FOR 'THE'.                                    T1
SYMBOL FOR 'MERRIER', END OF LIST.                   M2      0


PROBLEM 3 **********************

                                             X3      0
N1 CORRESPONDS TO NEW YORK.                          N1
C1 CORRESPONDS TO CHICAGO.                           C1
D1 CORRESPONDS TO DENVER.                            D1
L1 CORRESPONDS TO LOS ANGELES.                       L1      0


PROBLEM 4 **********************

                                             X4      0
                                                     A0
                                                     +0
                                                     B0
                                                     =0
                                                     A0
                                                     +0
                                                     C0      0


PROBLEM 5 **********************

THE ERROR IS IN 5P.
AFTER EXECUTION H0 SHOULD BE...              H0      X5      0

SOLUTION 5A.                                 H0      X1
                                                     X1
                                                     X2      0
SOLUTION 5B.                                 H0      X3
                                                     X3
                                                     X2      0
SOLUTION 5C.                                 H0      Y1
```

```
                                          Y1
                                          X2      0
SOLUTION 5D.                        H0    0       0
                                    X1    X2      0
SOLUTION 5E.                        H0    0       0
                                    X3    X2      0
SOLUTION 5F.                        H0    0       0
                                    X3    0       0
SOLUTION 5G.                        X1    0       0
SOLUTION 5H.                        X3    0       0
SOLUTION 5I.                        X5    0       0
SOLUTION 5J.                        X1    X3
                                          X3
                                          X3
                                          X4      0
SOLUTION 5K.                        X3    X5
                                          X5
                                          X5
                                          X6      0
SOLUTION 5L.                        X5    X7
                                          X7
                                          X7      0

SOLUTION 5M.  NO CHANGE.
SOLUTION 5N.  NO CHANGE.
SOLUTION 5P.  NO CHANGE.
SOLUTION 5Q.  NO CHANGE.
SOLUTION 5R.  NO CHANGE.
SOLUTION 5S.  NO CHANGE.


PROBLEM 6 **********************

SOLUTION 6A.
RESTORE H0.                               30H0

SOLUTION 6B.
RESTORE H0.                               30H0

SOLUTION 6C.
RESTORE H0.                               30H0

SOLUTION 6D.
INPUT 1X1.                                11X1
INPUT X3.                                 10X3
OUTPUT TO X1.                             20X1

SOLUTION 6E.
INPUT 2X1.                                12X1
INPUT A1.                                 10A1
OUTPUT TO 1X1.                            21X1

SOLUTION 6F.
INPUT 2X1.                                12X1
REPLACE WITH 2H0.                         52H0
```

```
INPUT B4.                                10B4
OUTPUT TO 2X1.                           22X1

SOLUTION 6G.
PRESERVE X1.                             40X1
INPUT X3.                                10X3
OUTPUT TO X1.                            20X1

SOLUTION 6H.
PRESERVE 1X1.                            41X1
INPUT X5.                                10X5
OUTPUT TO 1X1.                           21X1

SOLUTION 6I.
PRESERVE 2X1.                            42X1
INPUT X7.                                10X7
OUTPUT TO 2X1.                           22X1

SOLUTION 6J.
RESTORE X1.                              30X1

SOLUTION 6K.
RESTORE 1X1.                             31X1

SOLUTION 6L.
RESTORE 2X1.                             32X1

SOLUTION 6M.
REPLACE WITH Y4.                         50Y4

SOLUTION 6N.
REPLACE WITH Y4.                         50Y4

SOLUTION 6P.
REPLACE WITH X1.                         50X1

SOLUTION 6Q.
INPUT X3.                                10X3
OUTPUT TO X1.                            20X1

SOLUTION 6R.
INPUT X5.                                10X5
OUTPUT TO 1X1.                           21X1

SOLUTION 6S.
INPUT X6.                                10X6
OUTPUT TO 2X1                            22X1


PROBLEM 7 **********************

                                2 T      2
                                2 M      3
                                2 D      8
```

```
                                        2 X                     5

PROBLEM 9 **********************

                                        5
                                          E5      13K0
                                                    J151
                                                  10K0
                                                    F1
                                                    J151    0
                                        5           01
                                          K0          0
                                                      A1
                                                      B3
                                                      E9
                                                      A1
                                                      C9
                                                      A1
                                                      C9
                                                      B3      0
                                        5             E5

PROBLEM 11 **********************   5

PRESERVE W0,W1. PUT SYMBOL (0) IN       R66         J51
    W0, PUT LIST (1) IN W1.                       11W1
                                                  11W0
TEST IF SYMBOL IS ON LIST.                          J77
REVERSE SENSE OF H5.                                J5
IF NOT ON LIST, RESTORE W'S AND                   70J31
    QUIT WITH H5-.                                11W1
OTHERWISE ADD SYMBOL TO LIST,                     11W0
    RESTORE W'S, QUIT +.                            J65     J31


PROBLEM 13 **********************   5

                                        P33       10X7
PRINT LIST X7.                                      J151
PRESERVE W0.                                    40W0
                                                10X7
LOCATE FIRST OCCURRENCE OF                      10A1
    SYMBOL A1 ON LIST X7.                         J62
OUTPUT LOCATION TO W0.                          20W0
OUTPUT SYMBOL C1 INTO LOCATION.                 10C1
                                                21W0
LOCATE NEXT OCCURRENCE OF                       11WC
    SYMBOL A1 ON LIST X7.                       10A1
                                                  J62
OUTPUT LOCATION INTO W0.                        20W0
OUTPUT SYMBOL C1 INTO LOCATION.                 10C1
                                                21W0
```

```
LOCATE FIRST OCCURRENCE OF                      10X7
    SYMBOL B1 ON LIST X7.                       10B1
                                                  J62

LOCATE NEXT OCCURRENCE OF                       10B1
    SYMBOL B1 ON LIST X7.                         J62
LOCATE THIRD OCCURRENCE OF                      10B1
    SYMBOL B1 ON LIST X7.                         J62
DELETE THIS CELL FROM X7.                         J68
PRINT LIST X7.                                  10X7
                                                  J151   J30


PROBLEM 15 **********************  5

PUT SYMBOL IN W0, LIST IN W1.          P65        J51
LOCATE SYMBOL ON LIST.                 9-1     11W1
                                                  J62

IF IT DOESN'T OCCUR, GO TO 9-2.                709-2
COPY LOCATION INTO 9-3.                        609-3
PUT SYMBOL INTO LOCATION.                      11W0
GO TO 9-1.                                     219-3   9-1
                                       9-2       J31   J8
WORKING STORAGE.                       9-3        0     0


PROBLEM 17 **********************  5

REVERSE (0) AND (1).                   P67        J6
LOCATE LAST CELL ON LIST.                         J61
TEST IF CONTENTS OF CELL = SYMBOL.             52H0    J2


PROBLEM 19 **********************  5

PUT (0) IN W0, (1) IN W1.              P69        J51
LOCATE 1W1 ON LIST.                    9-1     11W1
                                                  J62

IF IT DOESN'T OCCUR GO TO 9-2.                 709-2
PRESERVE LOCATION.                             40H0
INSERT 1W0 BEFORE THE SYMBOL                   11W0
    IN THIS LOCATION.                           J63
LOCATE NEXT SYMBOL, GO TO 9-1.                  J60   9-1
CLEAN UP W'S AND H0.                   9-2      J31   J8


PROBLEM 21 **********************  5

PUT SYMBOL IN W0, LIST IN W1.          P71        J51
DELETE SYMBOL FROM LIST.                       11W1
                                               11W0
                                                  J69

IF IT DOESN'T OCCUR QUIT -.                    70J31
DELETE SYMBOL FROM LIST.               9-1     11W1
                                               11W0
```

```
                                                J69
IF IT WAS ON LIST REPEAT AGAIN.                 70          9-1
OTHERWISE CLEAN UP, QUIT +.                       J31       J4


PROBLEM 23 **********************  5

PUT LIST (0) IN W0, (1) IN W1.          P73       J51
LOCATE LAST CELL ON LIST 1W1.                   11W1
                                                  J61
INSERT LIST 1W0 AFTER LIST 1W1.                 11W0
                                                  J76
PRINT COMBINED LIST.                            51W1
CLEAN UP.                                         J151      J31


PROBLEM 25 **********************  5

PRESERVE H5                             P75     40H5
PUT LIST (0) IN W0, (1) IN W1.                    J51
LOCATE NEXT CELL ON LIST 1W0.           9-1     11W0
                                                  J60
                                                20W0
IF NONE GO TO 9-3.                              709-3
OTHERWISE LOCATE THE SYMBOL             9-2     11W1
    IN CELL 1W0 ON LIST 1W1.                    12W0
DELETE IT FROM LIST 1W1.                          J69
IF DELETED, GO TO 9-2, ELSE 9-1.                709-1     9-2
RESTORE H5 AND THE W'S.                 9-3     30H5      J31


PROBLEM 28 **********************  5

PUT SYMBOL (0) IN W0.                   P77       J50
DO TEST 9-1, THEN CLEAN UP.                       9-1       J30
PRESERVE H0.                            9-1     40H0
TEST IF SYMBOL IN H0 IS LOCAL.                    J132
IF NOT GO TO 9-2.                               709-2
OTHERWISE PRESERVE H0.                          40H0
FIND THE 2ND SYMBOL ON THIS LIST.                 J82
APPLY TEST 9-1 TO THIS SYMBOL.                    9-1
IF TEST IS + POP H0 AND QUIT +.                 70          J8
ELSE APPLY 9-1 TO 3RD SYMBOL.                     J83       9-1
TEST IF SYMBOL (0) = SYMBOL 1W0.        9-2     11W0      J2


PROBLEM 31 **********************  5

PUT (0) IN W0.                          P80       J50
CREATE LIST OF LOCATIONS.                         P78
IF NONE RESTORE W0 AND QUIT.                    70J30
OTHERWISE SAVE NAME OF LIST IN H0.              40H0
LOCATE NEXT CELL ON LIST.               9-1       J60
IF NONE GO TO 9-2.                              709-2
```

```
OTHERWISE COPY LOCATION IN 9-3.          609-3
PUT SYMBOL INTO THE CELL STORED          11W0
    IN THIS LOCATION, GO 9-1.            229-3    9-1
POP LAST LOCATION OUT OF H0.       9-2   30H0
ERASE CREATED LIST, POP W0, QUIT.          J71    J30
WORKING STORAGE.                   9-3     0      0


PROBLEM 34 **********************  5

CREATE LOCAL NAME LIST.            P87     J90
PUT LIST IN W0, TEST SYMBOL IN W1.         J51
APPLY TEST 9-10.                           9-10
PRESERVE TEST RESULT IN H5.               40H5
INPUT LOCAL NAME LIST.                    11W0
FIND NEXT LOCATION IN LIST.        9-2     J60
IF NONE GO TO 9-1.                        709-1
ELSE UNMARK LOCAL NAMES, GO 9-2.          32H0     9-2
ERASE LOCAL NAME LIST.             9-1    51W0
                                           J71
RESTORE H5 AND W'S.                       30H5     J31
PRESERVE H0.                       9-10   40H0
TEST IF SYMBOL IN H0 IS LOCAL.             J132
IF NOT GO TO 9-12.                        709-12
MARK (0).                                  J137
PUT (0) ON LOCAL NAME LIST.               61W0
                                          41W0
PRESERVE H0.                       9-11   40H0
FIND 3RD (COUNTING MARK) SYMBOL.           J83
APPLY TEST 9-10 TO THIS SYMBOL.            9-10
IF TEST IS + POP H0 AND QUIT +.           70       J8
ELSE APPLY 9-10 TO 4TH SYMBOL.             J84    9-10
                                   9-12   11W1     J2


PROBLEM 37 **********************  5

PUT (0) IN W0.                     P90     J50
CREATE LIST OF LOCATIONS.                  P88
IF NOT RESTORE W0 AND QUIT.               70J30
OTHERWISE SAVE NAME OF LIST IN H0.        40H0
LOCATE NEXT CELL ON LIST.          9-1     J60
IF NONE GO TO 9-2.                        709-2
OTHERWISE COPY LOCATION IN 9-3.           609-3
PUT SYMBOL 1W0 INTO CELL STORED           11W0
    IN THIS LOCATION, GO 9-1.             229-3    9-1
POP LAST LOCATION OUT OF H0.       9-2    30H0
ERASE CREATED LIST, POP W0, QUIT.          J71    J30
                                   9-3     0      0


PROBLEM 40 **********************  5

COUNT LIST H2.                     P93    10H2
```

```
                                                J126
COPY NAME OF DATA TERM INTO 9-0.                609-0
TEST IF (0) IS GREATER THAN (1).                J115
ERASE CREATED DATA TERM.                        119-0    J9
WORKING STORAGE.                        9-0       0      0


PROBLEM 43 **********************  5

STORE (0) IN W0.                        P96       J50
CLEAR 9-2.                                        109-2
LEAVE 9-2 IN (0), INPUT (1) IN (1).               J124
DOES 1W0 = 9-2.                         9-1     11W0
                                                  J114
IF SO POP W0 AND QUIT +.                          70       J30
OTHERWISE LOCATE NEXT CELL.                       J60
IF NONE POP W0 AND QUIT -.                        70J30
OTHERWISE TALLY 9-2.                              109-2
LEAVE 9-2 IN (0), LOCATION IN (1).                J125     9-1
                                        9-2     +01                0


PROBLEM 46 **********************  5              01

9-1 = DESCRIPTION LIST.                 B2          9-1      0
                                        9-1          0
A1=AUTHOR.                                          A1
P1=PUBLISHER.                                       X2
                                                    P1
                                                    Y2       0
9-1 = DESCRIPTION LIST.                 B3          9-1      0
                                        9-1          0
A1=AUTHOR.                                          A1
P1=PUBLISHER.                                       X3
                                                    P1
                                                    Y3       0
9-1 = DESCRIPTION LIST.                 B4          9-1      0
                                        9-1          0
A1=AUTHOR.                                          A1
P1=PUBLISHER.                                       X4
                                                    P1
                                                    Y4       0


PROBLEM 49 **********************  5

PUT L12 IN W0.                          Q3        10L12
                                                  J50
LOCATE NEXT CELL AFTER 1W0.             9-1     11W0
                                                  J60
                                                20W0
IF NONE POP W0 AND QUIT.                        70J30
OTHERWISE FIND THE VALUE                        10L13
    OF ATTRIBUTE 2W0                            12W0
```

```
    (BOOK) ON LIST L13.                          J10
IF NONE GO TO 9-1.                              709-1
OTHERWISE COPY DATA TERM.                        J120
ASSIGN COPY AS VALUE OF                         12W0
    ATTRIBUTE D1 ON LIST 2W0.                    J6
                                                10D1
GO TO 9-1.                                       J11     9-1


PROBLEM 52 **********************  5

PRESERVE W0-W4.                         R90      J44
DIVIDE LIST L14.                                10L14
                                                 J75
COPY NAME IN W0.                                60W0
GENERATE SYMBOLS ON ORIGINAL L14                109-0
    FOR SUBPROCESS 9-0.                          J100
STORE N0 IN HEAD CELL OF L14.                   10N0
                                                20L14
PRINT L14.                                      10L14
                                                 J150
ERASE ORIGINAL L14.                             11W0
RESTORE W0-W4.                                   J72     J34
PUT NAME OF SUBLIST IN W1 AND W2.       9-0     60W1
                                                20W2
PUT L14 IN W3 AND W4.                           10L14
    THIS IS THE TABLE WE'RE                     60W4
    GOING TO CREATE.                            20W3
LOCATE NEXT CELL IN SUBLIST.            9-1     11W2
                                                 J60
                                                20W2
PUT LOCATION IN TABLE INTO H0.                  11W3
LOCATE NEXT CELL.                       9-2      J60
IF NONE GO TO 9-4.                              709-4
TEST IF THE ATTRIBUTE (2H0) IN                  12H0
    THIS LOCATION IS IDENTICAL TO               12W2
    THE SYMBOL (2W2) WE'RE AT.                   J2
IF SO, GO TO 9-3.                               70      9-3
IF NOT SKIP VALUE, GO TO 9-2.                    J60    9-2
TEST IF LIST AT THIS LOCATION IN        9-3     11W2
    THE TABLE IS EMPTY.                          J78
IF EMPTY GO TO 9-2.                             709-6
IF NOT FIND FIRST SYMBOL.                        J81
PUT SYMBOL IN W3 AND W4.                        60W4
GO TO 9-1.                                      20W3    9-1
FIND FIRST SYMBOL.                      9-6      J81
PUT IT IN W4.                                   20W4
PUT NAME OF SUBTABLE INTO CELL                  12W1
    NAMED BY THIS SYMBOL.                       21W4    J4
ADD THE SYMBOL WE'RE AT                 9-4     51W3
    (2W2) TO THE END OF                         12W2
    THIS SUBTABLE.                               J65
PUT NAME OF SUBTABLE IN H0.                     11W3
TEST IF LIST AT LOCATION 1W2                    11W2
```

```
    IS EMPTY.                                   J78
IF EMPTY GO TO 9-5.                            709-5
IF NOT EMPTY CREATE A CELL.                     J90
MAKE THE CELL LOCAL.                            J136
PUT A COPY OF THE CELL NAME INTO               60W3
    W3 AND W4.                                 60W4
ADD CELL NAME TO END OF SUBTABLE.               J65
STORE NO INTO HEAD CELL OF THIS                10N0
    SUBTABLE, GO TO 9-1.                       21W4     9-1
ADD THIS SUB-SUBTABLE (2W1)          9-5       12W1
    TO SUBTABLE (1W3).                          J65      J4


PROBLEM 55 **********************  5

PUT SYMBOL (0) IN W0.                Q7         J50
GENERATE SYMBOLS ON LIST L12                   10L12
    FOR SUBPROCESS 9-1.                        109-1
                                                J100
POP W0, REVERSE SENSE OF H5, QUIT.              J30      J5
TEST IF (0) IS REGIONAL.             9-1       40H0
                                                J130
IF NOT POP H0, QUIT +.                         709-4
ELSE FIND DESCRIPTION LIST.                     J80
IF NONE QUIT +.                                70J4
PUT J3 IN 9-2.                                 10J3
GENERATE SYMBOLS ON DESCRIPTION                209-2
    LIST FOR SUBPROCESS 9-3.                   109-3    J100
SIGNAL = - IF ATTRIB., + IF VALUE.   9-2        J3       0
IS THIS A VALUE.                     9-3       019-2
REVERSE SENSE OF H5.                            J5
PUT 1H5 IN SIGNAL, SETTING                     11H5
    IT FOR NEXT CYCLE.                         209-2
IF NOT A VALUE POP H0, QUIT +.                 70       J8
IS THIS VALUE A LOCAL SYMBOL.                  40H0
IF NOT POP H0 AND QUIT +.                       J132
GENERATE SYMBOLS ON LOCAL VALUE                709-4
    LIST FOR SUBPROCESS 9-5.                   109-5    J100
                                     9-4        J8       J4
TEST IF THIS IS REQUIRED SYMBOL.     9-5       11W0
IF SO STOP GENERATOR, ELSE GO ON.               J2       J5


PROBLEM 58 **********************  5

CREATE 'CELLS TO BE DELETED' LIST.   Q10        J90
PUT LIST IN W0.                                 J50
GENERATE CELLS ON (0) FOR                      109-1
    SUBPROCESS 9-1.                             J101
DELETE ALL CELLS ON THE                        11W0
    'CELLS TO BE DELETED' LIST.                10J68
                                                J100
ERASE 'CELLS TO BE DELETED' LIST.              11W0
POP W0 AND QUIT.                                J71      J30
```

```
TEST IF THE SYMBOL IN CELL (0)          9-1    12H0
    IS AN OPERATOR.                            109-3
                                                 J6
                                                 J77
IF IT IS NOT, POP H0 AND QUIT +.                709-2
ELSE ADD CELL (0) TO                            21W0
    'CELLS TO BE DELETED' LIST.                 41W0      0
                                           9-2    J8     J4
LIST OF OPERATORS.                         9-3     0
                                                   +
                                                   -
                                                   *
                                                   /       0


PROBLEM 61 **********************  5

SET UP GENERATOR.                          Q13    10W0
                                                  J17
COPY LIST CELL (0) IN W0.                  9-1    60W0
FIND 2ND SYMBOL.                                  J82
IF NONE CLEAN UP GENERATOR.                     70J19
FIND 1ST SYMBOL.                                11W0
                                                  J81
EXECUTE SUBPROCESS.                               J18
IF H5 IS - CLEAN UP GENERATOR.                  70J19
LOCATE CELL AFTER NEXT.                         11W0
GO TO 9-1.                                        J60
                                                  J60    9-1


PROBLEM 64 **********************  5

SET UP GENERATOR.                          Q16    10W1
                                                  J17
PUT LIST NAME IN W0.                            60W0
KEEP NAME OF HEAD CELL IN W1.                   20W1
FIND NEXT LOCATION.                        9-0    11W0
                                                  J60
                                                20W0
IF NONE GO TO 9-1.                              709-1
PUT LOCATION IN H0.                             11W0
EXECUTE SUBPROCESS.                               J18
IF H5 - QUIT, ELSE GO TO 9-0.                   70J19    9-0
SHOULD WE ITERATE AGAIN.                   9-1    01R0
IF NOT QUIT.                                    70J19
ELSE PUT NAME OF HEAD CELL                      11W1
    IN W0.                                      20W0     9-0


PROBLEM 67 **********************  5

PUSH DOWN W0 AND W1.                       Q19    J41
PUT SUBEXPRESSION (0) IN W0.                    20W0
```

```
GEN LOCS OF VAR(1) FOR S/P 90,QUIT.          109-0
                                               Q18     J31
PUT LOCATION IN W1.                  9-0     20W1
COPY SUBEXPRESSION 1W0.                      11W0
                                               J74
MAKE COPY LOCAL.                               J136
PUT COPY IN GEN LOC, QUIT +.                 21W1      J4


PROBLEM 70 **********************  5

SET UP GENERATOR.                    R85     10W0
                                               J17
GENERATE SYMBOLS ON LIST (1)                 109-1
    FOR SUBPROCESS 9-1, QUIT.                  J100      J19
TEST IF SYMBOL NAMES AN              9-1     40H0
    INTEGER DATA TERM.                       109-2
                                               J127
IF SO EXECUTE ORIGINAL SUBPROCESS.           70          J18
OTHERWISE POP H0 AND QUIT +.                   J8        J4
                                     9-2   +01                0
INCREMENT COLUMN BY FIVE.            R86     109-5
                                               J161
TEST IF DATA TERM IS POSITIVE.               40H0
                                               J118
IF SO ENTER IT IN PRINT LINE.                70          J157
IF NOT ENTER A ( IN PRINT LINE.              10(
                                               J156
MAKE 9-0 IDENTICAL TO DATA TERM.             109-10
                                               J121
TAKE ABSOLUTE VALUE.                           J122
ENTER ABSOLUTE VAL IN PRINT LINE.              J157
                                             10)         J156
                                     9-5   +01                5
                                     9-10  +01                0
GENERATE THE SYMBOLS ON LIST L18     R87     10L18
    FOR SUBPROCESS 9-1.                      109-1       J100
CLEAR THE PRINT LINE.                9-1       J154
GENERATE THE INTEGERS ON                     10R86
    LIST FOR S/P R88, PRINT LINE.              R85       J155


PROBLEM 73 **********************  5

PRESERVE W21, W24, W25, AND W30.     Q21     40W21
                                             40W24
                                             40W25
                                             40W30
                                             109-0
                                             20W21
                                             10X1
                                             20W24
                                             109-1
LEAVE 1W25 IN H0.                            60W25
```

```
                                        109-2
                                        20W30
                                        10J3
                                          J90
PUT LIST IN W0, SIGNAL IN W1.           J51
DO 9-3.                                 9-3
POP 1W25 OUT OF H0.                     J8
RESTORE W21, W24, W25, AND W30.         30W21
                                        30W24
                                        30W25
                                        30W30
PUT LIST IN H0.                         11W0
                                        01W1
                                          J31
ERASE LIST IF SIGNAL IS STILL -.        70J9      0
                                9-0     +01           0
                                9-1     +01           0
CLEAR PRINT LINE.               9-2     +01           0
READ A CARD.                    9-3       J154
                                          J180
STOP IF THERE IS NO CARD TO READ.       70J7
SCAN (WITH 1W25) FOR NON-BLANK. 9-4       J184
GO TO 9-3 IF REST OF CARD IS BLANK.     709-3
PUT CHARACTER AT 1W25 INTO H0.            J186
TEST IF IT IS /.                        10/
                                          J2
QUIT IF IT IS /.                        70        0
OTHERWISE CLEAR 1W30.                   11W30
                                          J124
SCAN (WITH 1W30) FOR NEXT BLANK.          J183
REPLACE 1W30 WITH LIST 1W0 IN H0.       51W0
COPY 9-0.                               109-0
                                          J120
MAKE COPY LOCAL.                          J136
INPUT LINE DATA TERM.                     J182
ADD IT TO LIST 1W0.                       J65
SET SIGNAL +.                           10J4
GO TO 9-4.                              20W1      9-4
                                  5       .0
```

Part Two

PROGRAMMERS' REFERENCE MANUAL

1.0 GENERAL DEFINITIONS

1.1 IPL LANGUAGE

IPL is a formal language in terms of which information can be stated and processes specified for processing the information. IPL allows two kinds of expressions: data list structures, which contain the information to be processed; and routines, which define information processes. A complete program consists of a set of data list structures and the set of routines that define the processing to be done.

1.2 IPL COMPUTER

No computer currently available can process the IPL language directly, but any general purpose digital computer can be made to interpret this language by writing a special program in the language of the computer. Such a program is called an IPL-V interpretive system. The interpretive system interprets IPL expressions as equivalent expressions in the language of the particular computer, and causes the computer to carry out IPL processes. When a computer is running with the IPL interpreter system, its main storage has two major sections, one containing the IPL interpretive system, and the remainder--called the total available space--in which IPL programs and data may be stored. The particular computer, together with the interpretive system, is known as the IPL computer. The total available space is the "storage" of the IPL computer.

The interpretive system consists of several parts:

1) A loader, for loading IPL programs into the available space from cards or tape;
2) A set of primitive processes, for manipulating IPL expressions;
3) An interpreter, for executing the instructions in the IPL routines;

4) A monitor, for providing debugging information.

1.3 IPL SYMBOLS

IPL is a system for manipulating symbols. The IPL computer distinguishes three types of symbols--regional, internal, and local. It keeps track of the type of each symbol being used, and will behave differently in some cases, according to the type of symbol encountered.

To the programmer, a regional symbol is a letter or punctuation mark followed by a positive decimal integer no greater than 9999; e.g., A 1, *12, R3496. Regional symbols are the programmer's stock of symbols. An internal symbol is a positive decimal integer. Internal symbols are the computer's stock of symbols, and will generally not be used by programmers. Inside the computer--that is, except for input and output--internal and regional symbols are treated identically. Each symbol corresponds to a particular storage address. However, there are means to tell regional and internal symbols apart, if needed.

Local symbols are used to connect lists and list structures. Their identity is transitory--they are erased, generated, and changed at will by the IPL computer. To the programmer, a local symbol is a 9 followed by a positive decimal integer no greater than 9999; e.g., 9-1, 9-34, 9-123. The 9 takes the place of the letter in the regional symbols. The use of local symbols will be explained in § 2.0, DATA LIST STRUCTURES.

All symbols are printed out in the same form as they are input: regionals are printed in the letter-numbers form; internals are printed as decimal integers; and locals are printed as integers prefixed by a 9.

1.4 STANDARD IPL WORDS

All IPL expressions, both data list structures and

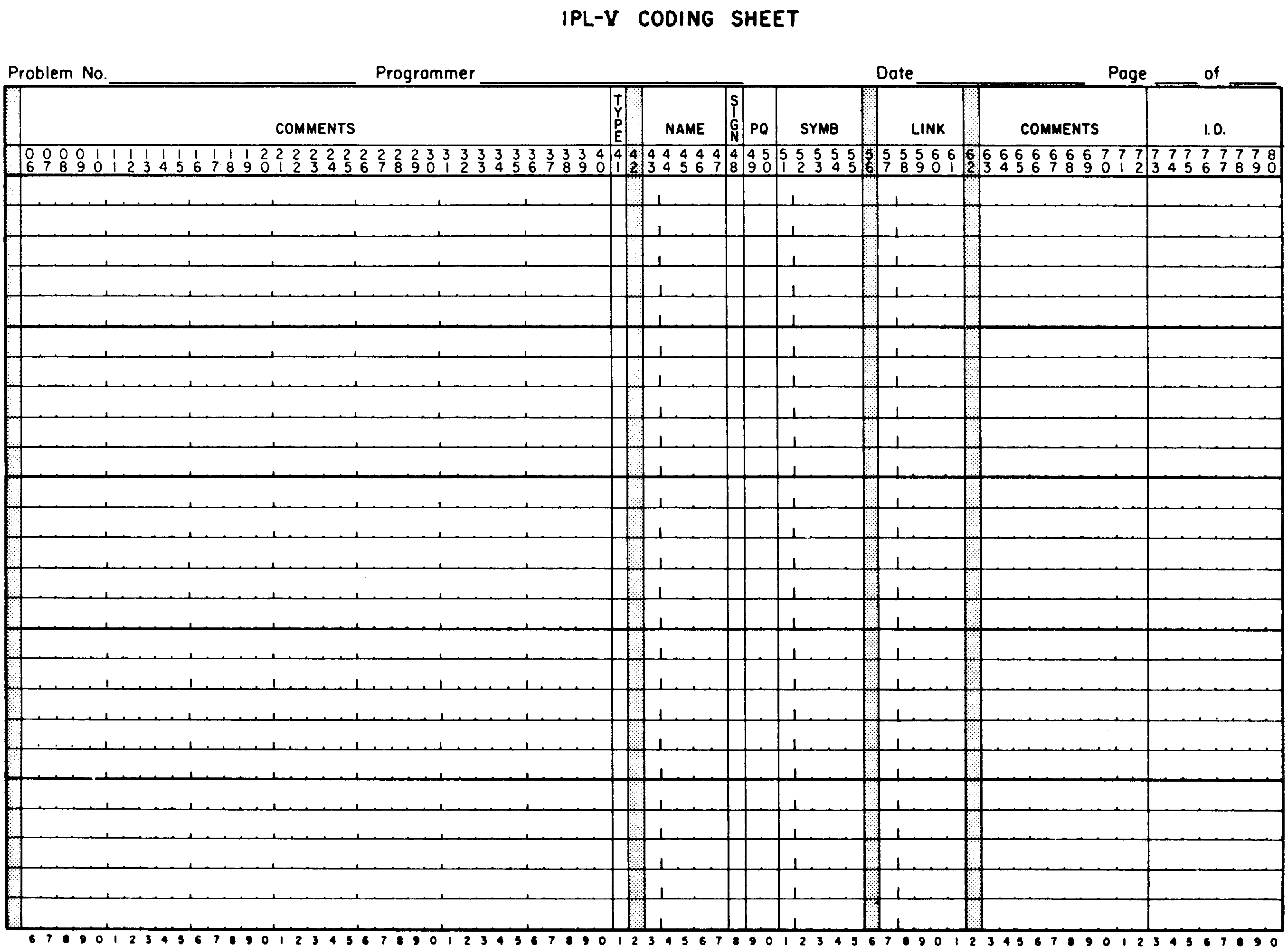
routines, are written in terms of an elementary unit, called the IPL word. Each word occupies a single cell of the total available space in the IPL computer. A standard word consists of four parts: P, Q, SYMB, and LINK. P and Q are called the prefixes of the word. Q is the designation prefix and P is the operation prefix (for routines) or the data type prefix (for data list structures). Each prefix is an octal digit--i.e., it may take on the values 0, 1, ..., 7. Its meaning depends on whether it occurs in routines or data. The SYMB part of the word is an IPL symbol, and is called the symbol of the word. The LINK part is also an IPL symbol.

1.5 SPECIAL IPL WORDS: DATA TERMS

Different formats are necessary to represent integers, floating point numbers, alphabetic characters, etc. Words containing such information are called data terms, and have three parts: P, Q, DATA. P and Q are prefixes, and DATA contains the special datum. The Q prefix is always 1, indicating that the word is a data term. The P prefix specifies the type of data. (Q = 1 is also used in routines with a different meaning; program and data are kept separate by context.)

1.6 THE CODING FORM

To put IPL words into the IPL computer, they must first be coded and punched into cards. The cards can then be read by the interpretive system. The cards are prepared from the standard coding form, one card per line, each card representing one IPL word (see Fig. 1). For standard IPL words, the columns labeled NAME, P, Q, SYMB, and LINK are used. Type is 0 or blank, Sign (+ -) is irrelevant (but see § 18.0, INITIAL LOADING), and all other columns are ignored by the IPL computer. (Certain columns are excluded from use.) P and Q may each contain



IPL-V CODING SHEET

Problem No. ____________ Programmer ____________ Date ____________ Page ____ of ____

COMMENTS	TYPE		NAME	SIGN	PQ	SYMB		LINK		COMMENTS	I.D.
06–40	41	42	43–47	48	49–50	51–55	56	57–61	62	63–72	73–80

Fig. 1--IPL-V Coding Sheet

any digit from 0 through 7. Blank is regarded as 0. For data lists, P and Q are always blank (or 0) unless the word is a data term. NAME, SYMB, and LINK may contain any IPL symbol. If LINK is left blank, the IPL computer automatically fills in the address of the next cell, represented by the next line on the coding sheet. This is also true for SYMB. However, if the next line has a regional or internal symbol as NAME, the blank LINK or SYMB is taken as a termination symbol 0.

NAME, SYMB, and LINK each occupy five columns. The first (leftmost) column holds the region character--i.e., the letter for regions, or 9 for local symbols. The other four columns hold the four-digit integer associated with the symbol. The integer may be located anywhere within the field in consecutive digits. For example, A1, A 1, A 1, and A0001, are all instances of A1. Likewise, 910, 9 10, and 9-10 are all instances of the local symbol 9-10, as long as the 9 occurs in the leftmost column. (In the Manual, we shall use "9-" to indicate a local symbol.) The exact rules for writing legitimate IPL symbols in NAME, SYMB, and LINK are the following:

- Regional and local symbols must have their initial character in the leftmost column of the field (columns 43, 51, and 57 respectively). Internal symbols may start anywhere in the field, except that if the initial digit is "9", that digit cannot be in the leftmost column.

- Except for the character in the leftmost column, all non-numeric characters and blanks are ignored.

- The numeric part of the symbol may occur anywhere in the field with any spacing. The field is scanned, and the digits are accumulated as they are found and composed into a number.

1.7 DATA TYPE CODE P

The format for data terms is shown in Fig. 2. Data terms have been defined for P = 0, 1, 2, 3, only. The other

IPL-V CODING SHEET

Problem No. _____ Programmer _____ Date _____ Page ____ of ____

COMMENTS	TYPE	NAME	SIGN	PQ	SYMB	LINK	COMMENTS	I.D.
P = 0 DECIMAL INTEGER								
General Format --			±	01	ddddd	ddddd		
Examples: -12,192			-	1	1	2192		
15				1		15		
P = 1 FLOATING POINT								
General Format --			±	11	ddddd	d -ee		
(Exponent ee ranges from +50 to								
-50. Any character in col. 59								
is taken as "-".)								
Examples: -.15 x 10^3			-	11	15	3		
.732 x 10^{-2}				11	732	-2		
P = 2 ALPHANUMERIC								
General Format --				21	aaaaa			
Example: IPL V				21	IPL 5			
P = 3 OCTAL								
General Format --			±	31	ddddd	ddddd		
Example: 7777777777				31	77777	77777		

Fig. 2--Format for Data Terms

four values, 4 through 7, are available for private use (see machine system write-ups).

1.8 CELLS

Each IPL word resides in a cell in the IPL computer (that is, a register in the total available space). We say a cell contains the word, also that the cell contains a symbol; i.e., the SYMB part of the word. Alternatively, we refer to SYMB as the symbol in a cell. LINK is also a symbol, but this is referred to as the link in a cell.

1.9 AVAILABLE SPACE

Since each IPL word resides in a cell in the IPL computer, during a run the routines and data list structures require a certain amount of the total available space--that is, of the total set of cells. At any moment during a run there is a set of cells which are not part of any routine or data list structure. This set is called the available space at that moment. It is the stock of cells out of which new list structures can be constructed. The available space is continually depleted as new structures are built, but continually replenished as old structures are no longer needed and are erased--i.e., the cells composing them returned to available space. All the available space is on a list, named H2, and called the available space list. The mechanics for transferring cells to and from available space will be described later.

1.10 LIMITS ON THE NUMBER AND TYPES OF STRUCTURES

All data list structures and routines are built up from the available space, and any cell may be used for any purpose in such constructions. Consequently, as long as cells are available, construction can continue. No separate limits exist on how many data list structures,

storage cells, symbols, and so on, can be used. The only limit is in the total amount of available space.

1.11 AUXILIARY STORAGE

The storage that holds the interpretive system and the available space is called the main storage. Access is also possible to secondary storages--fast-auxiliary storage and slow-auxiliary storage--when available on the object machine.

1.12 CELL NAMES

Access to a word requires access to the cell that holds the word, and this requires that the cell have a known IPL name. The name of a cell is the IPL symbol that represents the machine address of the cell. All cells in use have names, either regional, local, or internal. The cells in available space are not considered to have names since only when they are taken for a specific use is the name determined. On the coding sheet, putting a symbol in the NAME field specifies that the word on that line will be in the cell named. In essence, cells are named by writing a symbol for NAME. The programmer need name only those cells he wishes to refer to explicitly; hence, NAME is left blank in most instances.

1.13 HEADS, LIST CELLS, TERMINATION CELLS

Cells are used to construct the various structures in IPL. There are three kinds of cells: heads, which start structures; list cells, which form the bodies of structures; and termination cells, which mark the end of structures. (Data terms occur in heads.) We will need these distinctions in giving the conventions for each type of structure. A termination cell contains the word 00 00000 00000, and the symbol that names it is called a termination symbol. The internal symbol 0 is a termination symbol, and is used by the programmer in preference to other termination symbols.

Hence, it is referred to as the termination symbol. The need for other termination symbols arises from the delete processes (see § 9.4, DELETE). Any cell containing 0--i.e., SYMB = 0--is called empty.

1.14 STORAGE CELLS

A storage cell is one whose purpose is to hold symbols. A storage cell is created by giving a cell a regional name and putting the termination symbol, 0, for LINK. SYMB is then the symbol contained in the cell; it may be put in initially by writing in the symbol on the coding sheet, or the cell may be left empty and a symbol put in during processing.

Examples:

	NAME	PQ	SYMB	LINK
The empty storage cell, A1:	A1		0	0
The cell, A2, containing B3:	A2		B3	0

Any cell may function as a storage cell (assuming it is not being used in some other capacity).

1.15 PUSH DOWN LISTS FOR STORAGE CELLS

Associated with each storage cell, is a system for storing symbols contained in the cell. This system is a data list, called a push down list. The storage cell is the head of the list, and the cells used in the storage system are list cells. The symbol currently in the storage cell may be put onto the push down list, so that the cell can be used for another purpose, and then recovered at a later time. The system is a "Last-In-First-Out" system (LIFO); that is, the symbols are recovered from storage in the inverse order of their entry. The most recently preserved symbol is the first one recovered. The system is fully specified by the operation for putting symbols in storage, preserve or push down, and the operation

for recovering symbols from storage, restore or pop up.

PRESERVE To preserve a storage cell is to put a copy of the symbol contained in the cell on the push down list associated with the cell. The operation leaves the symbol still in the cell.

RESTORE To restore a storage cell is to move into the cell the symbol most recently put on the associated push down list of that cell. The symbol occurrence in the cell just prior to restoring is lost, and the symbol moved from the push down list is no longer on the list.

Examples: Let the storage cell W3 contain the symbol S5:

NAME	PQ	SYMB	LINK
W3		S5	0

If W3 is preserved, than a copy of S5 goes into storage, while W3 continues to hold S5:

W3		S5	
		S5	0

If another symbol, B1, is now put into W3, we have:

W3		B1	
		S5	0

If W3 is preserved again, we have:

W3		B1	
		B1	
		S5	0

And if another symbol, G3, is put into W3, we have:

W3		G3	
		B1	
		S5	0

If W3 is restored, then:

W3		B1	
		S5	0

And if W3 is restored again:

W3		S5	0

After two preserves followed by two restores, W3 is brought back to the original condition; and similarly for any number of preserves followed by the same number of restores.

Each cell, then, really consists of a stack of symbols. The one on top is accessible, and the others are in storage in the order in which they are put in the stack. There is no limit to the number of symbols that may be stored in a push down list; it is always possible to add another as long as some available space remains in the IPL system.

2.0 DATA LIST STRUCTURES

The data list structure is the IPL expression that contains the data to be processed. The total data for a program will be given as a set of data list structures. Each data list structure is made up of data lists, which in turn are made up of IPL words. (Routines are also list structures, but satisfy different conventions.)

2.1 DATA LISTS

A data list is a sequence of cells containing IPL words whose order is defined by the rule: the LINK part of the cell contains the name of the next cell in the list. The first cell in a list--the cell which does not have its name as the LINK of any cell of the list--is the head of the list. All other cells of the list are list cells. The following rules apply to all data lists:

- -Only the names of list cells can occur as the LINK of a cell.
- -Only names of heads can occur as the SYMB of a cell.
- -The name of each list cell occurs once and only once as LINK (this is equivalent to making lists linear, without cycles).
- -The LINK of the last cell in a list is a termination symbol.

A list with a termination symbol for the LINK of the head is called an empty list. Cells containing data terms (cells with Q = 1), while not subject to the above rules, are considered to be the heads of empty data lists when manipulating data list structures.

To create a data list, write down a symbol in the NAME field of some line. This symbol is the name of the list, and the cell corresponding to it is the head of the list. (Thus, the same symbol names both the list and the head cell.) Write down the IPL words of the list on successive lines of the coding sheet. These lines are the list cells,

and they occur in the list in the order they appear on the coding sheet. No names are given to the list cells (NAME left blank) and the LINKs of all cells but the last one are left blank. The public termination symbol, 0, is written for LINK of the last cell.

Examples:	NAME	PQ	SYMB	LINK
The list with name L1, containing the symbols S1, S5, S12, and S7 in that order: (the first symbol occurs in the head here; conventions for heads will be given presently)	L1		S1 S5 S12 S7	 0
The list with name 9-5, containing the symbols A5 and 9-3.	9-5		A5 9-3	 0

The termination symbol, 0, is used, although any other termination symbol is perfectly legal. The latter would require an additional cell, and thus take extra space without any compensating gain.

2.2 NAMING LIST CELLS

The IPL computer will assign an internal name to any cell that is not explicitly named by the programmer. The programmer may give names to list cells by using local symbols (using regional symbols would start a new list, in effect). The IPL computer interprets a blank SYMB or LINK in a cell as referring to the next cell, and the name of this next cell is filled in. This occurs properly either when the next cell has a blank NAME or a local symbol for NAME. If the next cell has a regional name, the blank SYMB or LINK is taken as the termination symbol, 0.

Example:	NAME	PQ	SYMB	LINK
The usual reason for naming data list cells is to break the sequential order on the coding sheet:	L1 9-2 9-1 9-3		0 S2 S1 S3	9-1 9-3 9-2 0

2.3 DESCRIBABLE LISTS

It is possible to associate with a list, a description list, similar in concept to a function table, which can contain information about the list being described. The SYMB of the head is reserved for the name of the description list. A list with the head so reserved is called describable. If a list is describable, descriptive information can be added to it or requested about it, at any time during processing, by means of a set of processes, J10 - J15. Since the head of a describable list is reserved, the first symbol is in the first list cell after the head, and so on. Lists that use the head for any other purpose are called non-describable. If no information has been associated with a describable list, then there will exist no description list. However, the head is still reserved, and hence is empty. (The list in the previous example has no description list associated with it but has a reserved head.)

2.4 POLICY ON DESCRIBABLE LISTS

The basic processes (the J's) assume that data lists are describable whenever this is relevant to their operation. In the Manual we will assume a list to be describable, unless explicitly stated otherwise.

2.5 ATTRIBUTES AND VALUES

The information that can be associated with a describable list is in the form of values to specified attributes. Suppose L1 is a describable list, and A1 is some attribute, say the number of symbols on a list. Then, the value of A1 for L1 is some symbol, say N3. This can be expressed in mathematical notation as A1(L1) = N3. Any symbol at all may be used as an attribute, no matter what its other functions in the total program might be. The value of an attribute is always a single symbol. However, any symbol

may be the value--for example, the name of a data term, the name of a list, or the name of a list structure--so that there is no restriction at all on the kind of information that can effectively be the value of an attribute. Only a single value is possible for a given attribute, but it is always possible for the value of an attribute to be the name of a list of "values," thus achieving the effect of multivalued attributes. The usefulness of descriptions stems from the generality of what constitutes an attribute or a value. Any number of attribute values may be associated with a describable list.

2.6 DESCRIPTION LISTS

A description list is a list that contains alternately the symbols for attributes and their values. The attribute symbol occurs first, followed by its value for the list the description list is describing. Description lists are themselves describable, so that the first attribute symbol occurs in the first list cell, its value in the second list cell, the next attribute symbol in the third, and so on. The same symbol cannot occur more than once as an attribute on the description list.

2.7 CREATING DESCRIPTION LISTS

Processes exist to create, modify, interrogate, and erase description lists during processing (see J10 to J15). Such lists can also be created on the coding sheet prior to loading. A local name is written for SYMB of the head of the list to be described. The description list is defined in the same manner as any other list: its name is written for NAME on some line (the same symbol as occurred in the head of the main list); the head of the description list is made empty since the description list is describable; then follow the attributes and values in sequence

on the coding sheet; the final value has a termination symbol for LINK. (No other list structures may intervene on the coding sheet between the describable list and the description. See § 2.9, DOMAIN OF DEFINITION OF LOCAL SYMBOLS.)

Examples:	NAME	PQ	SYMB	LINK
The describable list, L1, with no descriptions:	L1		0	
			S1	
			S2	
			S3	0
L2 described by the attributes A1 and A2 with values V1 and V2, respectively:	L2		9-0	
			S1	
			S2	
			S3	0
	9-0		0	
			A1	
			V1	
			A2	
			V2	0

2.8 DATA LIST STRUCTURES

A list structure is a set of lists connected together by the names of the lists occurring on other lists in the set. A data list structure is characterized by the following conditions:

- -All the component lists are data lists (hence, linear--that is, not re-entrant).
- -There is one list, called the main list, that has a regional or internal name.
- -All lists, except the main list, have local names, and are called sublists.
- -All local names that occur in the list structure--that is, as SYMB of some cell--name lists that belong to the list structure.
- -No cell belongs to more than one list (no merging of lists).
- -The name of each component list, except the main list, occurs at least once on some list of the list structure; it may occur many times.

A data list structure is thus a fairly simple form of list structure--many complicated ways of linking lists together having been excluded. It is not the simplest, which would be a tree, since it is possible for the name of a sublist to appear in several places in the structure. Data terms are included in the definition, as are storage cells, since they are also data lists. The name of a list structure is the name of its main list. (Thus, this symbol does triple duty as the name of a list structure, a list, and a cell.) Not all symbols occurring in a list structure refer to other lists in the structure: if they are regional or internal symbols, their referents cannot belong to the same list structure. Thus, there can be complicated cross references between a set of data list structures.

2.9 DOMAIN OF DEFINITION OF LOCAL SYMBOLS

The domain of definition of a local symbol is a list structure. Within a single list structure, a local symbol can be the name of only one data list--that for which it occurs as NAME. All occurrences of a local symbol within a list structure are understood to refer to this data list. However, there is no connection between the local symbols in one list structure and those in another (which is why they are called local). Thus, the symbol 9-1 will stand for many things in a total program. Contrariwise, a regional symbol, like A1, or an internal symbol, like 1622, always stands for the same object throughout the total program. On the coding sheet, the occurrence of a regional or internal symbol for NAME marks the start of a list structure. All local symbols that occur after this line belong to this list structure, until another regional or internal NAME occurs.

2.10 LEVELS

It is often convenient to refer to the lists of a data list structure as having levels. The main list has the highest level, and a sublist is one level below its superlist--i.e., the list on which its name occurs. (It is possible for the name of a list to occur on several lists at different levels.) If numbers need to be assigned to levels, the main list is assigned level 1 and increasing positive integers are used for successively lower levels.

Examples:	NAME	PQ	SYMB	LINK
A single list can be a data list structure:	L1		0	
			S1	
			S2	
			S3	0
A single data term can be a data list structure:	B5	21	BILL	
A list of lists can be a data list structure (the spaces between lists are for clarity in the Manual; no such spaces need occur on the coding sheet):	L2		0	
			9-1	
			9-2	
			9-3	0
	9-1		0	
			S1	
			S2	
			S3	0
	9-2		0	
			S3	
			S1	
			S2	0
	9-3		0	
			S2	
			S3	
			S1	0
A list of numbers can be a list structure; in the example, two of the numbers belong to the structure and the other, N3, does not:	L3		0	
			9-3	
			N3	
			9-1	0
	9-1	01		15
	9-3	- 01		19

	NAME	PQ	SYMB	LINK
A list can have multiple occurrences of sublists, as well as mutual references and self references:	L4		0	
			9-1	
			9-1	0
	9-1		0	
			9-2	0
	9-2		0	
			9-1	
			9-2	0
If the name of the main list, which is internal or regional, appears in the list structure, it is treated like any other regional or internal symbol; the example, L5, is a simple list.	L5		0	
			L5	
			L5	
			L5	0
The algebraic expression, (X1+X2)·(X3-X4) can be written as a list structure where the sublist arrangement indicates the parenthetical structure:	X0		0	
			9-1	
			·	
			9-2	0
	9-1		0	
			X1	
			+	
			X2	0
	9-2		0	
			X3	
			-	
			X4	0

2.11 OTHER LIST STRUCTURES

Other kinds of list structures besides data list structures are possible and useful--e.g., circular lists, in which the "last" cell links to the "first" cell. The programmer is free to invent and use any such structures he desires, but he is then responsible for being aware of their special nature. Almost any kind of structure can be loaded in the computer (see § 18.0, INITIAL LOADING). We have defined the class of data list structures, in order to provide useful processes which take into account their particular conventions--e.g., copy and erase an entire data list structure.

3.0 ROUTINES AND PROGRAMS

The IPL expressions used to specify information processes are generally similar to their data counterparts, but differ in detail. Corresponding to the word of data is the instruction, to the data list is the program list, and to the data list structure is the routine.

3.1 PRIMITIVE PROCESSES

A primitive process is one that can be directly performed by the computer without further IPL interpretation; i.e., one that is coded directly in machine language. IPL symbols can name primitives. Most of the basic processes (the J's) are primitives, and it is possible to add primitives to the language (see machine system write-ups).

3.2 INSTRUCTIONS

The IPL word that specifies an information process is called an instruction. It always has the standard form: PQ SYMB LINK. The process to be done is designated by PQ SYMB, while the LINK, as usual, designates the next cell in a list. The P and Q codes are entirely different from the data P and Q codes. They denote operations to be carried out rather than types of symbols and data. (The information that SYMB is regional, internal, or local is lost in an instruction, but is not needed for interpretation.) The definitions of P and Q, given presently, completely define the process designated by an instruction.

3.3 PROGRAM LISTS

A program list is a sequence of cells containing instructions, whose order is defined by the following rule: the LINK of a cell is the name of the next cell in the list.

The first cell in a list is the head; all others are list cells. The head contains an instruction, so no program list is describable. In interpretation, the program list gives a sequence of instructions to be carried out in the order of the list. Almost anything is possible with program lists: they may be re-entrant, or merge; they may have regional symbols as LINKs, and names of list cells as SYMB.

3.4 ROUTINES AND PROGRAMS

A routine is a list structure characterized by the following conditions:

- Some of the lists are program lists.
- There is one program list, called the main list, that has a regional or internal name.
- All lists, except the main list, have local names and are called sublists (and initiate local subroutines).
- All local names that occur in the list structure as SYMB of some cell, name lists that belong to the list structure.
- The name of each sublist occurs at least once on some list of the list structure; it may occur many times.
- The main list is not describable (since it is a program list).

Local symbols follow the same rules for the domain of definition given in connection with data list structures. It is also possible to talk about the levels in a routine in the same manner as with data list structures. Each routine specifies a process. A routine is executed when this specified process is carried out by the IPL computer. This implies that the subroutines out of which the process is composed, are also executed (as required). A program is the set of routines that specifies a process in terms of primitive processes. The routine first executed is at the highest level. The routines of the program are all

routines required in the execution of this top routine, taking into account that routines require other routines for their execution.

3.5 DATA IN ROUTINES

Normally, routines consist purely of program lists. However, it is sometimes convenient to include various kinds of data along with the routine, such as constants, storage cells, and so on. Since data list structures are handled differently from program lists on input (P and Q are treated differently), it is necessary to indicate which cells are to be interpreted as data. A + or - in the Sign column is used for this, and every cell in routines to be interpreted as data must be so marked. (The + or - contributes to the data only in the case of numeric data terms, as defined earlier; in all other cases it has no effect.)

3.6 SAFE CELL

A storage cell is called safe over a routine if that routine leaves the symbol in the cell (and the push down list) the same as it was prior to the execution of the routine, except as modification is explicitly required by the definition of the routine. If there is no guarantee that the contents of the storage cell will remain unmolested, the cell is called unsafe over the routine. A routine can use a safe cell, as long as it returns the cell to the original condition. Safe cells are useful in IPL because the preserve and restore operations make it easy to use a storage cell and then return it to an earlier condition. From the point of view of the using routine, a safe cell is one into which it can put a symbol, then execute a subroutine, and expect to find the symbol still in the cell afterwards.

3.7 INPUTS AND OUTPUTS OF ROUTINES, HO

A routine can have a set of operands, called the input symbols. It can also produce a set of symbols as outputs. It may also modify existing data list structures, either those designated by input symbols, or those implicit in the construction of the routine. The number of inputs or outputs is unlimited. They are always symbols, but these symbols can name list structures (either data or routines), so that the types of inputs and outputs are completely general.

All inputs for a routine are placed in a special storage cell, HO, called the communication cell. If there are multiple inputs, they are placed in the push down list of HO in a sequence determined by the definition of the routine. All outputs from a routine are also placed in the communication cell, HO. If there are multiple outputs, they are placed in the push down list of HO in a sequence determined by the definition of the routine. In the Manual we will let (0), (1),..., represent, respectively, the symbols in HO and its push down list. They will serve as names for the inputs and outputs. The communication cell is safe over all routines: in connection with inputs, this means that a routine must remove (before it terminates) all the input symbols from the communication push down list. The outputs, of course, are explicitly required to be in HO at the end of processing. (Of course, routines can be defined with any input-output conventions the programmer desires. The above ones are used by the basic processes (the J's), and means are provided to make them easy to use generally.)

3.8 EXPLICIT STATEMENT OF INPUTS AND OUTPUTS

The safety of HO implies that a routine must remove all its input symbols from HO. Its outputs, of course,

are to be left in HO. In order to avoid confusion, we adopt the policy of explicitly stating all inputs and outputs. For example, if a routine leaves one of its input symbols in HO, this is to be stated explicitly as one of the outputs.

3.9 TEST CELL, H5

The result of many processes involves a binary distinction--a "yes" or "no." For example, a process may be a "test" whose purpose is to make a binary choice, or it may produce an output where there is no guarantee that the output can be produced, so that a binary indication, "Yes, the output was produced," or, "No, the output was not produced," is needed as well as the output symbol in those cases where it can be produced. A special storage cell, H5, called the test cell, is used for this binary information. It can contain either of two special symbols,"+", which stands for yes, or "-", which stands for no. The + and - are symbols used only in the Manual. In the computer, J4 is the symbol for + and J3 for - . These are, respectively, the names of the basic processes that set H5 + or - . The test cell is safe over the basic processes (the J's); that is, if a J-process does not set H5 as part of its definition, then H5 will be the same after performance of the process as it was before. (This means that conditional transfers may be delayed after the decision has been made and recorded in H5, as long as only J's which do not set H5 are performed.)

3.10 THE DESIGNATION OPERATION, Q, AND THE DESIGNATED SYMBOL, S

In instructions, the Q prefix specifies an operation, called the designation operation, whose operand is SYMB. The result of performing the designation operation on SYMB is a new symbol, S, called the designated symbol of the instruction. We give below all eight values of Q. The first five Q's, Q = 0, 1, ..., 4, are normally the only ones that appear on the coding sheet.

- Q = 0 S = the symbol in the instruction itself--i.e., SYMB.
- Q = 1 S = the symbol in the cell named in the instruction--i.e., in SYMB.
- Q = 2 S = the symbol in the cell whose name is in the cell named in the instruction--i.e., in the cell named in SYMB.
- Q = 3 Trace this program list (otherwise equivalent to Q = 0).
- Q = 4 Continue tracing (otherwise equivalent to Q = 0).
- Q = 5 SYMB is the address of a primitive--i.e., of a machine language subroutine.
- Q = 6 Routine is in fast-auxiliary storage.
- Q = 7 Routine is in slow-auxiliary storage.

Examples:

Given the memory situation:

NAME	PQ	SYMB	LINK
B1		C1	0
C1		D1	0

For the three instructions below we get the following designated symbol:

S = B1	0	B1
S = C1	1	B1
S = D1	2	B1

3.11 THE OPERATION CODE, P

The P prefix specifies an operation, called simply the operation of the instruction, whose operand is the designated symbol, S. The result is an action related to the set up, execution, and clear up of routines. The eight operations are:

- P = 0 EXECUTE S. S is assumed to name a routine or a primitive; it is executed--i.e., the process it specifies is carried out--before the next instruction is performed.
- P = 1 INPUT S. H0 is preserved; then a copy of S is put in H0.
- P = 2 OUTPUT TO S. A copy of (0) is put in cell S; then H0 is restored.
- P = 3 RESTORE S. The symbol most recently stored in the push down list of S is moved into S; the current symbol in S is lost.
- P = 4 PRESERVE S. A copy of the symbol in S is stored in the push down list of S; the symbol still remains in S.
- P = 5 REPLACE (0) BY S. A copy of S is put in H0; the current (0) is lost.
- P = 6 COPY (0) IN S. A copy of (0) is put in S; the current symbol in S is lost, and (0) is unaffected.
- P = 7 BRANCH TO S IF H5 - . The symbol in H5 is always either + or - . If H5 is + , then LINK names the cell containing the next instruction to be performed. (This is the normal sequence.) If H5 is - , then S names the cell containing the next instruction to be performed.

Thus, P = 0 is used to execute subroutines; P = 1, 2, 5, and 6, are used to transfer symbols to and from the communication cell, H0; P = 3 and 4 are used in connection with safe cells; and P = 7 is a centralized transfer of control.

Examples: On the right we give small segments of program lists--i.e., sequences of instructions. On the left we give a verbal statement of the action.

	NAME	PQ	SYMB	LINK
It takes two instructions to put the symbol in W0 into the cell W1. The first instruction, 11W0, inputs the symbol 1W0 to H0, and the second, 20W1, moves the symbol into cell W1.		11	W0	
		20	W1	
It is desired to execute a process, P15, which takes two inputs and produces one output. The inputs are to be L1 and the symbol in W0; and the output is to be in W1. 10L1 inputs L1 to H0, pushing the symbol in H0 down, so it is not destroyed. 11W0 inputs the symbol in W0 to H0, again pushing down. Then P15 is fired; it removes the two symbols just put in H0, and places its own output there. 20W1 takes this output from H0 and puts it in W1 (destroying the symbol in W1). H0 is left as it was at the beginning.		10	L1	
		11	W0	
			P15	
		20	W1	
It is desired to put (0) into Y5, but without destroying the symbol already there. Hence, 20Y5 is preceded by 40Y5, which preserves Y5.		40	Y5	
		20	Y5	
It is desired to replace a symbol in the cell named in W1 by the symbol in the cell named in W0. 12W0 brings the symbol into H0, and 21W1 puts it in 1W1--i.e., in the cell named in W1. Notice that H0 is left just as it was before the two operations were performed.		12	W0	
		21	W1	
A process whose name is in Y2 is fired with input from W0. Assume it has one output. This is put into W1 by 60W1, which also leaves it in H0 so that J2 can test if it is equal to S5. The result of J2 is either a + or - in H5. 709-1 transfers control to the part of the program list starting at 9-1 if H5 is - . If H5+, then control proceeds down the list.		11	W0	
		1	Y2	
		60	W1	
		10	S5	
			J2	
		70	9-1	
			....	
	9-1		....	
Process P30 is fired on an input from W0. W0 is restored by 30W0 to bring it back to its earlier condition.		11	W0	
			P30	
		30	W0	

3.12 INTERPRETATION

The interpretation of a program consists of generating a sequence of primitives according to the lists in the program, and executing each primitive in turn. The part of the IPL computer that carries this process out is called the interpreter. The process consists of a cycle of operations, which we define in two alternative ways: first, as a series of rules, by the RULES OF INTERPRETATION; and second, as a step-by-step sequence of actions, by the INTERPRETATION CYCLE, similar to a flow diagram.

3.13 CURRENT INSTRUCTION ADDRESS CELL, H1

Execution of a routine in a program involves executing its subroutines. While executing a subroutine, it is necessary to remember the current location in the higher routine, so that when the subroutine is finished, interpretation can proceed from the correct instruction in the higher routine. The hierarchy of in-process subroutines is necessarily unlimited, since a subroutine can be composed of other subroutines of unknown composition. A special storage cell, H1, called the current instruction address cell, or CIA, is used to mark locations in the hierarchy of in-process routines. The symbol in H1 is the address of the current instruction; the symbol one-down in the push down list is the address of the instruction in the routine one level up; the next symbol down is the address of the instruction in the routine two levels up; and so on. (The programmer never uses H1; it is used solely by the interpreter.)

3.14 RULES OF INTERPRETATION

1. An instruction is interpreted by first applying Q to SYMB to get S and then applying P to S to get the action.

2. Generally, the instructions in a program list are interpreted in the order of the list. Control advances.

3. In case P = 7, the sequence may be broken (if H5-), but control remains at the same level and continues along the list from the cell with name S. Control branches.

4. A process designated in a program list is executed by remembering the address of its instruction in H1 (with a preserve), and then interpreting its program list--i.e., the list with the instruction in the head. Control descends a level.

5. A primitive process designated in a program list is executed by transferring machine control to the machine language subroutine corresponding to the primitive process; no descent occurs.

6. Interpretation of a program list terminates with a LINK = 0, the end of the list; or with LINK = name of a routine, in which case this routine is executed as the last process of the program list. (Termination is also achieved by branching to a 0 or the name of a routine via P = 7.)

7. Upon termination of a program list, control ascends a level, and interpretation proceeds in the program list that contained the name of the program list just finished, from the point at which it was executed (H1 is restored). If H1 is empty, the computer halts.

8. If the routine of a designated process is in auxiliary storage, the auxiliary block it belongs to is brought into main storage, and interpretation proceeds.

3.15 THE INTERPRETATION CYCLE

START: H1 contains the name of the cell holding the instruction to be interpreted.

INTERPRET Q:
- Q = 0, 1, 2: Apply Q to SYMB to yield S; go to INTERPRET P.
- Q = 3, 4: Execute monitor action (see § 15.0, MONITOR SYSTEM); take S = SYMB; go to INTERPRET P.
- Q = 5: Transfer machine control to SYMB (executing primitive); go to ASCEND.
- Q = 6, 7: Bring blocks of routines in from auxiliary storage; put location of routine in block into H1; go to INTERPRET Q.

INTERPRET P:
- P = 0: Go to TEST FOR PRIMITIVE.
- P = 1, 2, 3, 4, 5, 6: Perform the operation; go to ADVANCE.
- P = 7: Go to BRANCH.

TEST FOR PRIMITIVE: Q of S:
- Q = 5: Transfer machine control to SYMB of S (executing primitive); go to ADVANCE.
- Q ≠ 5: Go to DESCEND.

ADVANCE: Interpret LINK:
- LINK = 0: Termination; go to ASCEND.
- LINK ≠ 0: LINK is the name of the cell containing the next instruction; put LINK in H1; go to INTERPRET Q.

ASCEND: Restore H1 (returning to H1 the name of the cell holding the current instruction, one level up); restore auxiliary region if required; go to ADVANCE.

DESCEND: Preserve H1: Put S into H1 (H1 now contains the name of the cell holding the first instruction of the subprogram list); go to INTERPRET Q.

BRANCH: Interpret Sign in H5:
- H5-: Put S as LINK (control transfers to S); go to ADVANCE.
- H5+: Go to ADVANCE.

Figure 3 gives a schematic picture of the connections between the parts of the interpretive cycle. (The various machine systems may not correspond exactly to this diagram--see machine system write-ups for details.)

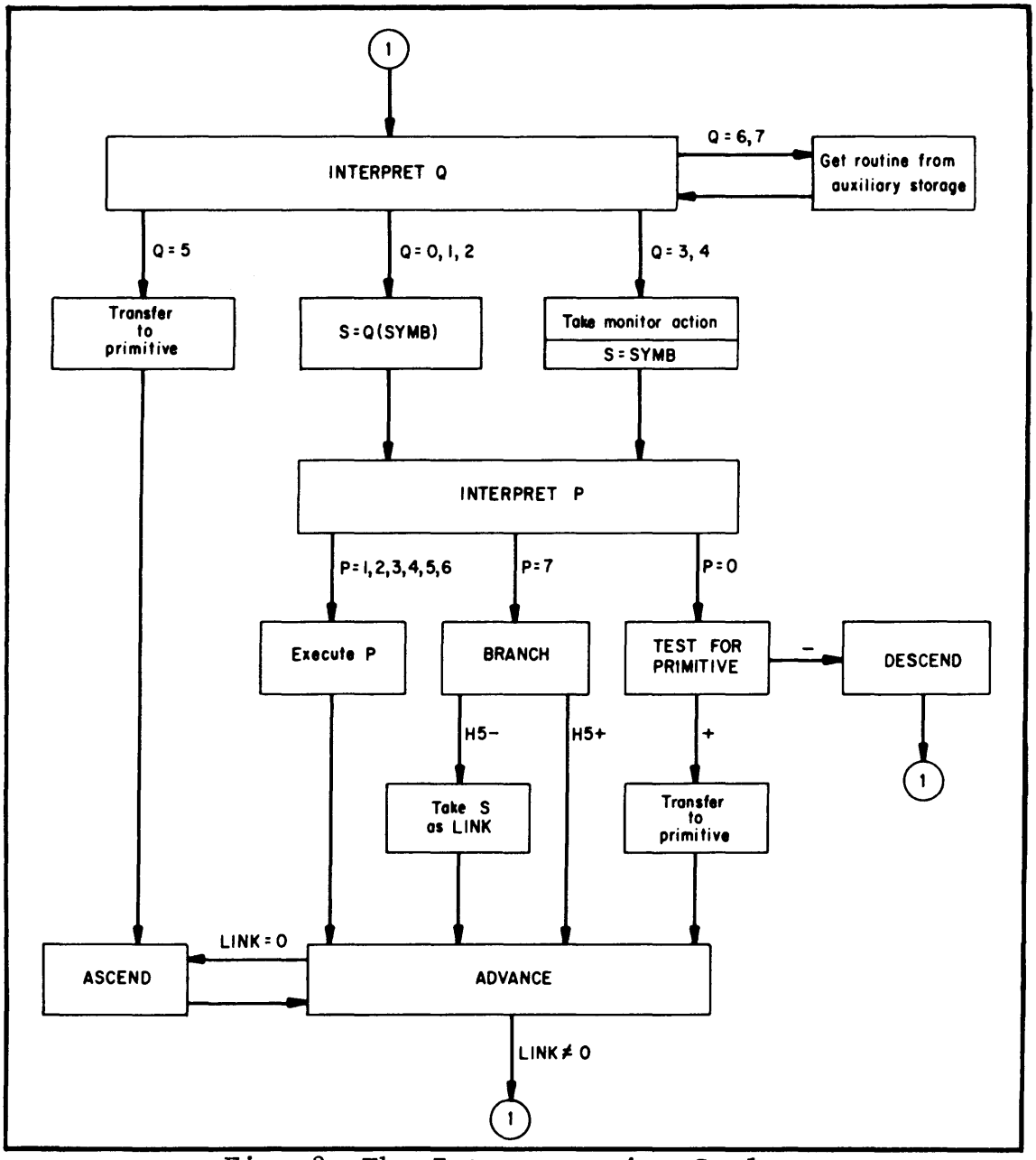


Fig. 3--The Interpretation Cycle

3.16 TALLY OF INTERPRETATION CYCLES, H3

The interpreter counts the number of cycles executed by tallying 1 into H3 every time an ADVANCE occurs. H3 is an integer data term. It is set to zero at the beginning of a run by the loader. It is available to the program during running--that is, it can be copied, reset to 0 at various points in the program, and so on. It provides a useful measure of the amount of processing done.

4.0 BASIC SYSTEM OF PROCESSES

The system of prefixes, P and Q, the interpreter, and the rules for constructing list structures, are essentially the grammar of IPL. In order to construct useful programs, it is necessary to add a set of basic processes for manipulating symbols, lists, description lists, list structures, and special format words. The system provided here is general purpose, in that any process can be accomplished with it. It is focused on list manipulation, however, with the consequence that arithmetical processes are inefficient in comparison with their machine code counterparts. The system consists of a set of storage cells with special functions (some of which have already been described), and a set of basic information processes. Some of the basic processes are primitives; some are elementary IPL routines included to complete the repertoire.

4.1 SYSTEM REGIONS (EXCLUDED FROM OTHER USE)

The regions H, J, and W are used by the system, and no new symbols in these regions may be defined by the programmer.

The $ region is set aside to be used by individual installations for their own system routines and data. The need for this arises because each installation eventually creates a few routines which it makes commonly available to its users. The designation of a single region for these prevents unnecessary conflicts, since users everywhere can avoid using the $ region. Similarly, it is unnecessary for an installation to use J-routine and W-cell names for its unique system routines and data.

4.2 SYSTEM CELLS

The following cells have special functions. They are all storage cells and safe, except H3, W11, and W33, which are integer data terms.

H0 The communication cell.

H1 Current instruction address cell (CIA); never used by programmer.

H2 Available space list; never manipulated by programmer, except to count with J126.

H3 Tally of interpretation cycles executed; an integer data term.

H4 Current auxiliary routine cell; never used by programmer.

H5 Test cell; safe over J's. (See § 3.9, TEST CELL, H5, for definition of safe.)

W0-W9 Ten cells for common working storage (see § 8.0, WORKING STORAGE PROCESSES, and § 7.0, GENERATOR HOUSEKEEPING PROCESSES).

W10 Random number control cell; holds the name of integer data term used to produce random numbers in J129 and J16.

W11 Remainder of integer division; an integer data term (see J113).

(See § 15.0, MONITOR SYSTEM, for W12 through W15, W23, W29.)

W12 Monitor start cell; holds name of routine executed at start of trace (Q = 3).

W13 Monitor end cell; holds name of routine executed at return to Q = 3 point.

W14 External interrupt cell; holds name of routine executed at signaled interruption.

W15 Post mortem routine cell; holds name of routine executed after the post mortem lists have been printed.

(See § 13.0, INPUT-OUTPUT CONVENTIONS, for W16 through W22, W24, W25.)

W16 Input mode cell; holds name of integer determining input mode.

W17 Output mode cell; holds name of integer determining output mode.

W18 Read unit cell; holds name of integer determining unit used by J140, J165, J180.

W19 Write unit cell; holds name of integer determining output unit used by J142 and loading processes.

W20 Print unit cell; holds name of integer determining unit for J150's.

W21 Print column cell; holds name of integer determining print column.

W22 Print spacing cell; holds name of integer determining line and page spacing.

W23 Post mortem list cell; holds name of list determining information to be printed on post mortem dump.

W24 Print line cell; holds name of present print line.

W25 Entry column cell; holds name of integer determining entry position in print line.

(See § 21.0, ERROR TRAP, for W26 through W28.)

W26 Error trap cell; holds name of list, in description list form, of trap symbols and associated processes.

W27 Trap address cell; holds CIA at the time of the trap.

W28 Trap symbol cell; holds symbol indicating cause of trap.

W29 Monitor point address cell; holds name of cell holding instruction with Q = 3.

W30 Field length cell; holds name of integer specifying the number of columns in the current input field for the line read primitives.

W31 Trace mode cell; holds the name of an integer specifying NO TRACE if 0, FULL TRACE if 1, and SELECTIVE TRACE if 2.

W32 Reserved available space cell; holds the name of an integer specifying how many cells of available space will be withheld from H2, to be returned when H2 is exhausted.

W33 Cycle count for trap cell; an integer data term. When H3 equals W33, the trap action routine associated with H3 on W26 is executed.

W34 Current available space cell; holds the name of the available space list used by the loading processes, initially H2.

W35 Slow-auxiliary obsolete structure cell; holds the name of an integer that tallies the number of obsolete data structures occupying space in the slow-auxiliary data system.

W36 Used slow-auxiliary space cell; holds the name of an integer that tallies the total number of data structures, both current and obsolete, occupying space in the slow-auxiliary data system.

W37 Slow-auxiliary storage density cell; holds the name of an integer specifying the percentage of used slow-auxiliary space that may be occupied by obsolete structures.

W38 Slow-auxiliary storage compacting routine cell; holds the name of the routine which tests whether slow-auxiliary storage should be compacted at this time, and compacts if yes.

W39 Fast-auxiliary obsolete structure cell; same as W35, but for fast-auxiliary.

W40 Used fast-auxiliary space cell; same as W36, but for fast-auxiliary.

W41 Fast-auxiliary storage density cell; same as W37, but for fast-auxiliary.

W42 Fast-auxiliary storage compacting routine cell; same as W38, but for fast-auxiliary.

W43 Format cell; holds the name of an integer data term specifying the format for J162.

5.0 GENERAL PROCESSES, J0 to J9

In this and following sections we give the definitions of the basic processes, accompanied by whatever general explanations are appropriate. Note that all outputs are explicitly named, and that only these outputs remain in H0 after completion of a routine. We include definitions of some terms with a circumscribed meaning.

TEST--A test is a process whose only result is to set H5 + or - . Its definition is of the form: "TEST X", where X is any statement. If X is true, then H5 is set + ; if X is false, then H5 is set - . Any number of inputs is permissible.

FIND--A find is a process with a single symbol as output, but where it is uncertain whether the output can be produced (can be found). If the output is produced, it is put in H0, and H5 is set + ; if the output is not produced, there is no output in H0, and H5 is set - . Any number of inputs is permissible.

MOVE--In normal computing one never destroys the information in the originating location when reading it into a new place; i.e., readouts are "non-destructive." In IPL, with the operation of restore, a "destructive" read becomes useful. Thus, move means to put in the newly designated place, but not to leave in the original place. If a symbol is being moved from a storage cell, then the cell is restored; if a list structure is being moved to auxiliary storage, then it is erased in main storage.

J0 NO OPERATION. Proceed to the next instruction.

J1 EXECUTE (0). The process, (0), is removed from H0, H0 is restored (this positions the process's inputs correctly), and the process is executed (as if its name occurred in the instruction instead of J1).

J2 TEST IF (0) ≡ (1). (The identity test is on the SYMB part only; P and Q are ignored.)

J3 SET H5-. The symbol in H5 is replaced by the symbol J3.

J4 SET H5+. The symbol in H5 is replaced by the symbol J4.

J5 REVERSE H5. If H5 is + , it is set - ; if H5 is - , it is set + .

J6 REVERSE (0) and (1). Permutes the symbol in HO with the first symbol down in the HO push down list.

J7 HALT, PROCEED ON GO. The computer stops; if started again, it interprets the next instruction in sequence.

J8 RESTORE HO. (Identical to 30HO, but can be executed as LINK.)

J9 ERASE CELL (0). The cell whose name is (0) is returned to the available space list, without regard to the contents of the cell.

6.0 DESCRIPTION PROCESSES, J10 to J16

As described earlier (§ 2.3, DESCRIBABLE LISTS), there are processes for manipulating descriptions and description lists. For all of them the name of the describable list is input, and not the name of the description list. The name of the description list is found in the head of the describable list, and, whenever created by these processes, is a local symbol. (This allows the description list to be erased automatically whenever the list is erased as a list structure--see J72.)

J10 FIND THE VALUE OF ATTRIBUTE (0) OF (1). If the symbol (0) is on the description list of list (1) as an attribute, then its value--i.e., the symbol following it--is output as (0) and H5 set + ; if not found, or if the description list doesn't exist, there is no output and H5 set - . (J10 is accomplished by a search and test of all attributes on the description list.)

J11 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) OF (2). After J11, the symbol (1) is on the description list of list (2) as the value of attribute (0). If (0) was already on the description list, the old value has been removed, and (1) has taken its place; if the old value was local, it has been erased as a list structure (J72). If (0) is a new attribute, it is placed at the front of the description list. J11 will create the description list (with a local name) if it does not exist (head of (2) empty). There is no output in H0.

J12 ADD (1) AT FRONT OF VALUE LIST OF ATTRIBUTE (0) OF (2). The value of (0) is assumed to be the name of a list. The symbol, (1), is inserted on the front of this list (behind head, as in J64). If the attribute is not on the description list, it is put on and a list is created as its value (with a local name). As in J11, if the description list doesn't exist, it is created.

J13 ADD (1) AT END OF VALUE OF LIST OF ATTRIBUTE (0) OF (2). Identical to J12, except that (1) is inserted at the end of the list, rather than the front.

J14 ERASE ATTRIBUTE (0) OF (1). If the symbol (0) exists on the description list of list (1) as an attribute, both it and its value symbol are removed from the list. If either is local, it is erased as a list structure (J72). If (0) is not an attribute on the description list of (1), nothing is done. (In all cases the description list is left.)

J15 ERASE ALL ATTRIBUTES OF (0). The description list of list (0) is erased as a list structure (J72), and the head of (0) is set empty.

J16 FIND ATTRIBUTE RANDOMLY FROM DESCRIPTION LIST OF (0). All the attributes on the description list of list (0) that have positive numerical data terms as values (integer or floating point) are taken as a population from which a random selection is made with relative weights given by their values. Thus, if there are attributes A_i with values $N_i > 0$:

$$\text{Probability of } A_j \text{ being selected} = \frac{N_j}{\sum_{\text{all } i} N_i}$$

The output (0) is the attribute symbol selected, and H5 is set + . If there are no positive numerical data terms on the description list, there is no output and H5 is set - . The random number used in J16 is generated as in J129, and is therefore controlled by W10.

7.0 GENERATOR HOUSEKEEPING PROCESSES, J17 to J19

7.1 GENERATORS

Repetitive operations can be handled in IPL by means of loops, utilizing the conditional branch, just as in normal programming. They can also be handled by means of generators. A generator is a process that produces a sequence of outputs and applies to each a specified process. The process that the generator applies is called the subprocess of the generator, and is an input. Thus, the generator is associated with the kind of sequence it produces, and will apply any process whatsoever to these outputs. The only thing a generator knows about the subprocess is the name of its routine, plus a convention allowing the subprocess to control whether or not the generator will continue to produce outputs of the sequence. This latter convention is necessary if generators are to be used conditionally--e.g., to search for a member of a sequence with certain properties.

What makes generators different from all the other processes considered so far, is that two contexts of information--that of the generator, and that of the subprocess and superprocess--must coexist in the computer at the same time. 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 routine is always working in its appropriate context. To see this, define the context of a routine to be the set of symbols in the working storages that it is using. We will assume that any routine using n+1 symbols of information, stores these in W0 through Wn, rather than some arbitrary subset of W's. The routine that uses a generator, which we will call the superroutine, has a certain context.

The subprocess is in the same context as the superroutine. The generator is being used to provide a sequence of information to be processed in the routine using the generator, and the subprocess is simply that part of the superroutine that does the processing. In general, it needs access to all the symbols in the context of the superroutine. It is given a name only to communicate to the generator what processing to do. The generator has an entirely different context in order to produce the sequence. The purpose of the generator is to separate the processing that goes into producing a sequence from the processing that is to be done to the sequence. There is an alternation between generator and subprocess which is both an alternation of control and an alternation of context; to produce an element of the sequence, the generator must be in control, and its context should occupy the W's; and to process the element, the subprocess must be in control, and the context of the superroutine should occupy the W's. Thus, whenever the generator fires the subprocess, it is necessary to remove the context of the generator from the W's, thus revealing the prior context, which is that of the superroutine. At the termination of the subprocess, the context of the generator must be returned to the W's (pushing down the W's, of course).

To handle the special housekeeping associated with generators, three routines are provided: J17 is used at the beginning of a generator to set up the housekeeping; J18 is used to fire the subprocess, and shuffles the contexts back and forth; and J19 is used at the end of a generator to clean up the housekeeping structures.

J17 GENERATOR SETUP. Has two inputs:
(0) = Wn, the name of the highest W that will be used for working storage--e.g., (0) = W6, if cells W0 through W6 will be used.
(1) = The name of the subprocess to be executed by generator.
J17 does three things (and has no output):
-Preserves the cells W0 through Wn, thereby preserving the superroutine-subprocess context;
-Stores Wn and the name of the subprocess in storage cells, and preserves a third cell for the output sign of H5 (these three storage cells are called the generator hideout);
-Obtains the trace mode of the superroutine, and records it in one of the hideout cells (see § 15.0, MONITOR SYSTEM).

J18 EXECUTE SUBPROCESS. Has no input. It does six things:
-Removes the symbols in W0 through Wn (generator context), thereby returning the previous context of symbols to the top of the W's (superroutine-subprocess context);
-Stacks the generator context in a hideout cell;
-Sets the trace mode of the subprocess to be that of the superroutine (see § 15.0, MONITOR SYSTEM);
-Executes the subprocess;
-Returns the symbols of the generator's context from the hideout to the W's, pushing the W's down, thereby preserving the superroutine-subprocess context;
-Records H5, the communication of the subprocess to the generator (see J19), in one of the hideout cells.

J19 GENERATOR CLEANUP. Has no input. Does three things:
-Restores W0 through Wn;
-Restores all the cells of the hideout;
-Places in H5 the recorded sign, which will be + if the generator went to completion (last subprocess communicated +), and - if the generator was stopped (last subprocess communicated -).

7.2 CONVENTIONS FOR USING GENERATORS

We can now summarize the conventions for the use of generators.

-A generator is executed like any other routine. Its inputs are placed in H0:
(0) is always the name of the subprocess;
(1), (2), ..., are inputs to the generator.

-The subprocess sets H5 upon termination: + if the generator is to produce the next number of the sequence; - if the generator is to terminate.

-There is no output from the generator to the superroutine except H5, which is + if the generator went to completion--i.e., produced all members of the sequence--and is - if the generator was terminated. J19 sets this output.

7.3 CONVENTIONS FOR CONSTRUCTING GENERATORS

We can now summarize the conventions for the construction of generators.

-Start the generator routine by doing J17: input (1), the subprocess, is already in place; do a 10Wn, where Wn is the highest working cell to be used, for input (0).

-Produce the first member of the sequence, and put it in H0 as input to the subprocess. The member may be given by any number of symbols, (0), (1), ...

-Fire the subprocess by executing J18. At the time of execution, the generator's symbols cannot be stacked up more than one deep in the W's or J18 will fail to clear the context.

-The subprocess operates in the context of the superroutine, taking as input the symbols provided by the generator, above. Thus, the symbols in the W's are the ones placed by the superroutine, or by one of the earlier executions of the subprocess. Likewise, the subprocess can put symbols in the W's (or H0), which are then available to later executions of the subprocess, or to the superroutine after the termination of the generator.

-Within the generator, after executing J18, if H5 is + , produce the next member of the sequence. If there are no more members, clean up and quit with J19, which will pop up the W's and set H5 for output. If H5 is - , then immediately clean up and quit with J19.

- There is no restriction on the nesting or cascading of generators: a generator may use other generators as subroutines; and a generator can be in the form of a subprocess operating on the output of another generator. (The subprocess of a generator is part of its context, so that J18 always fires the subprocess of the generator currently in context.)
- If the generator is in main storage, the subprocess to it may have either a regional or local name. If the generator is in auxiliary storage, the subprocess to it may need a regional name (see § 10.0, AUXILIARY STORAGE PROCESSES).

8.0 WORKING STORAGE PROCESSES, J20 to J59

Storage cells can be created at will by the programmer, and can be used either as permanent or temporary storage for any purpose the programmer desires. The only advantage in using the W's lies in the following forty processes for manipulating them, together with their built-in use in the generator processes.

J2n MOVE (0), (1), ..., (n) INTO W0, W1, ..., Wn, RESPECTIVELY. Ten routines, J20 through J29, that provide block transfers out of H0 into working storage. The symbols currently in W0 to Wn are lost.

J3n RESTORE W0, W1, ..., Wn. Ten routines, J30 through J39.

J4n PRESERVE W0, W1, ..., Wn. Ten routines, J40 through J49.

J5n PRESERVE W0, W1, ..., Wn, THEN MOVE (0), (1), ..., (n) INTO W0, W1, ..., Wn, RESPECTIVELY. Ten routines, J50 through J59, combining J4n and J2n.

9.0 LIST PROCESSES, J60 to J104

9.1 PRESERVE AND RESTORE AS GENERAL LIST OPERATIONS

The preserve and restore operations were defined earlier for storage cells. We describe below the mechanics underlying them. It can be seen that these operations can apply to any list, given the name of a cell in the list: preserve will insert an additional cell with the same PQ SYMB as the given cell, and restore will replace the contents of the given symbol with the contents of the following cell, and remove the following cell from the list, thus performing a deletion.

	NAME	PQ SYMB	LINK
We are given, initially, the available space list, H2, and a cell, W0, with a list proceeding from its LINK:	H2	0	1000
	1000	0	1050
	1050	0	1020
	1020	0	....
	W0	B2	500
	500	C1	505
	505	C2	....
If we preserve W0, then a word is obtained from available space and inserted in the list following W0, with a copy of SYMB of W0:	H2	0	1050
	1050	0	1020
	1020	0	
Notice that all words in the list except W0 remained unchanged, and that all the conditions for preserve are satisfied. Note also the amount of processing is independent of how many items are on the list.	W0	B2	1000
	1000	B2	500
	500	C1	505
	505	C2	....
If we now put into W0 a new SYMB, D1, we get (with no change in the H2 list):	W0	D1	1000
	1000	B2	500
	500	C1	505
	505	C2	....
Restoring W0 reverses the operation, deleting the cell next after W0, putting it back on the available space, but putting its SYMB in W0:	H2	0	1000
	1000	0	1050
	1050	0	1020
	1020	0	....
	W0	B2	500
	500	C1	505
	505	C2	....
Restoring W0 again yields: (Notice that cells are returned on the front of the available space list, H2, so that the amount of processing required is independent of the size of available space.)	H2	0	500
	500	0	1000
	1000	0	1050
	1050	0	1020
	1020	0	....
	W0	C1	505
	505	C2	....

9.2 LOCATE

A locate produces an output which is the name of the cell containing the desired symbol. Since there is no guarantee that the symbol is locatable, H5 is set + if it is, and - if it is not located. In the negative case, an output is still produced; in the locate processes in the basic system, J60, J61, J62, and J200, the output is the name of the last list cell. (A private termination cell is not a list cell.)

9.3 INSERT

In an insert, two symbols are specified, either by the inputs or as the result of preliminary processing by the insert processes: a symbol in a list cell, and a symbol that is to be inserted in the list relative to the first symbol. A new cell from available space is put in the list to hold the new symbol, which is then located in the appropriate relationship to the symbol already in the list. There are no outputs in HO.

Consider the mechanics for two relationships: insert before and insert after. Suppose the symbol to be inserted is A1, the symbol in the list is B1, and its list cell is 1000:

NAME	PQ SYMB	LINK
900	....	1000
1000	B1	910
910	....	....

In both cases we start by preserving 1000:

NAME	PQ SYMB	LINK
900	....	1000
1000	B1	1010
1010	B1	910
910	....	....

For insert before, we put A1 in 1000:

NAME	PQ SYMB	LINK
900	....	1000
1000	A1	1010
1010	B1	910
910	....	....

For insert after, we put A1 in 1010:

NAME	PQ SYMB	LINK
900	....	1000
1000	B1	1010
1010	A1	910
910	....	....

Notice that the symbols bear the appropriate relationship of before and after, but not necessarily the cells. Given the name of a cell, there is no way to insert a cell in front of it, since the cell that links to it is unknown.

9.4 DELETE

In a delete, a symbol in a list is specified, either by the input or as a result of preliminary processing, and it is desired to remove this symbol from the list, reducing the number of list cells by one. H5 is set - for appropriate special cases; e.g., if the symbol designated for deletion does not exist. Otherwise, it is set + .

	NAME	PQ	SYMB	LINK
Suppose the symbol to be removed is A1 and it is in list cell 1000:	900		....	1000
	1000		A1	910
	910		B1	920
	920		....	....
Then deletion is accomplished by restoring 1000:	900		....	1000
	1000		B1	920
	920		....	....
Notice that it is the cell after 1000 that is removed. It is not possible to remove a cell knowing only the name of the cell, since the name of the cell linking to it is unknown.				
Suppose, however, that cell 1000 was the last cell in the list:	900		....	1000
	1000		A1	0
Then, it is not possible to remove the next cell, which is 0, the termination symbol. Instead, 1000 is made into a <u>private termination</u> cell. This is the only way to make cell 900 the last cell in the list. H5 is set - to indicate that we have deleted the last symbol.	900		....	1000
	1000	0	0	0

9.5 POLICY ON PRIVATE TERMINATION CELLS

Private termination cells are introduced by the IPL system to allow deletion of final symbols on lists. They occur in no other way. They can gradually accumulate during processing, using up space. Consequently, J60, the process which locates the next symbol on a list, automatically returns private termination cells to available space, substituting the termination symbol, 0. (J60 can do this, since when it detects a termination cell, it still has available the name of the previous cell.) Any J's that use J60 as a subroutine will also have this feature (see machine system write-ups).

9.6 ERASE

To erase a structure of any kind is to return all the cells comprising it to available space. There is no output in H0.

9.7 COPY

To copy a structure of any kind is to produce a new set of cells from available space and link them together isomorphically to the given structure. All the cells of the new set will contain exactly the same symbols as their correspondents, except those that contain symbols used to link the structure together; e.g., local names in list structures. These contain the names of the copies of the corresponding lists. The name of the new structure is the output, (0).

9.8 LIST PROCESSES

J60 LOCATE NEXT SYMBOL AFTER CELL (0). (0) is the name of a cell. If a next cell exists (LINK of (0) not a termination symbol), then 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 input (0), which is the name of the last cell on the list, and H5 is set - .

If the next cell is a private termination cell, J60 will work as specified above, but in addition, the private termination cell will be returned to available space and the LINK of the input cell (0) will be changed to hold 0.

No test is made to see that (0) is not a data term, and J60 will attempt to interpret a data term as a standard IPL cell.

J61 LOCATE LAST SYMBOL ON LIST (0). (0) is assumed to be the name of a cell in a list (either a head or list cell; it makes no difference). The output (0) is the name of the last cell in the list, and H5 is set + . If there is no cell after (0), then the output (0) is the input (0) and H5 is set - .

J62 LOCATE (0) ON LIST (1). A search of list with name (1) is made, testing each symbol against (0) (starting with cell after cell (1)). If (0) is found, the output (0) is the name of the cell containing it and H5 is set + . Hence, J62 locates the first occurrence of (0) if there are several. If (0) is not found, the output (0) is the name of the last cell on the list, and H5 set - .

J63 INSERT (0) BEFORE SYMBOL IN (1). (1) is assumed to name a cell in a list. A new cell is inserted in the list behind (1). The symbol in (1) is moved into the new cell, and (0) is put into (1). The end result is that (0) occurs in the list before the symbol that was originally in cell (1).

J64 INSERT (0) AFTER SYMBOL IN (1). Identical with J63, except the symbol in (1) is left in (1), and (0) is put into the new cell, thus occurring after the symbol in (1). (If (1) is a private termination symbol, (0) is put in cell (1), which agrees with the definition of insert after.)

J65 INSERT (0) AT END OF LIST (1). Identical with J64, except that the location of the last symbol is obtained first, prior to inserting.

J66 INSERT (0) AT END OF LIST (1), IF NOT ALREADY ON IT. A search of list (1) is made, testing each symbol against (0) (starting with the cell after cell (1)). If (0) is found, J66 does nothing further. If (0) is not found, it is inserted at the end of the list, as in J65.

J67 REPLACE (1) BY (0) ON LIST (2) (FIRST OCCURRENCE ONLY). A search of list (2) is made, testing each symbol against (1) (starting with the cell after cell (2)). If (1) is found, (0) is placed in that cell. If (1) is not found, J67 does nothing.

J68 DELETE SYMBOL IN CELL (0). (0) names a cell in a list. The symbol in it is deleted by replacing it with the next symbol down the list (the next cell is removed from the list and returned to available space, so that the list is now one cell shorter). H5 is set + unless (0) is the last cell in the list or a termination cell. Then H5 is set - . Thus, H5- means that after J68, the input (0) (which is no longer in H0) is a termination cell (see discussion in § 9.4, DELETE).

J69 DELETE SYMBOL (0) FROM LIST (1) (FIRST OCCURRENCE ONLY). A search of list (1) is made, testing each symbol against (0) (starting with the cell after cell (1)). If (0) is found, it is deleted, as in J68, and H5 is set + . If (0) is not found, H5 is set - .

J70 DELETE LAST SYMBOL FROM LIST (0). The last symbol on list (0) is located. If a last symbol is found, it is deleted and H5 is set + . If no last symbol exists (list (0) is empty at input), H5 is set - .

J71 ERASE LIST (0). (0) is assumed to name a list. All cells of the list--both head and list cells--are returned to available space. (Nothing else is returned, not even the description list of (0), if it exists.) There is no output in H0. If (0) names a list cell, the cell linking to it will be linking to available space after J71, a dangerous but not always fatal situation.

J72 ERASE LIST STRUCTURE (0). (0) is assumed to name a list structure or a sublist structure. List (0) is erased, as are all lists with local names on list (0), and all lists with local names on them, and so on. Thus, description lists get erased, if they have local names. If the list is on auxiliary storage (Q of (0) = 6 or 7), then the list structure is erased from auxiliary, and the head, (0), is also erased.

J73 COPY LIST (0). The output (0) names a new list, with the identical symbols in the cells as are in the corresponding cells of list (0), including the head. If (0) is the name of a list cell, rather that a head, the output (0) will be a copy of the remainder of the list from (0) on. (Nothing else is copied, not even the description list of (0), if it exists.) The name is local if the input (0) is local; otherwise, it is internal.

J74 COPY LIST STRUCTURE (0). A new list structure is produced, the cells of which are in one-to-one correspondence with the cells of list structure (0). All the regional and internal symbols in the cells will be identical to the symbols in the corresponding cells of (0), as will the contents of data terms. There will be new local symbols, since these are the names of the sublists of the new structure. Description lists will be copied, if their names are local. If (0) is in auxiliary storage (Q of (0) = 6 or 7), the copy will be produced in main storage. In all cases, list structure (0) remains unaffected. The output (0) names the new list structure. It is local if the input (0) is local; it is internal otherwise.

J75 DIVIDE LIST AFTER LOCATION (0). (0) is assumed to be the name of a cell on a list. A termination symbol is put for LINK of (0), thus making (0) the last cell on the list. The output (0) names the remainder list: a new blank head followed by the string of list cells that occurred after cell (0).

J76 INSERT LIST (0) AFTER CELL (1), AND LOCATE LAST SYMBOL. List (0) is assumed to be describable. Its head is erased (if local, the symbol in the head is erased as a list structure). The string of list cells is inserted after cell (1): LINK of cell (1) is the name of the first list cell, and LINK of the last cell of the string is the name of the cell originally occurring after cell (1). The output (0) is the name of the last cell in the inserted string and H5 is set + . If list (0) has no list cells, then the output (0) is the input (1) and H5 is set - .

J77 TEST IF (0) IS ON LIST (1). Assume (1) is the name of a cell on a list. A search is done of all cells after (1); H5 is set + if (0) is found, and set - if not.

J78 TEST IF LIST (0) IS NOT EMPTY. H5 is set - if LINK of (0) is a termination symbol, and set + if not.

J79 TEST IF CELL (0) IS NOT EMPTY. H5 is set - if SYMB of (0) is 0, and set + otherwise. (Q of (0) is ignored; thus, both cells holding internal zero and termination cells give H5-.)

J8n FIND THE nth SYMBOL ON LIST (0), $0 \leq n \leq 9$. (Ten routines, J80-J89.) Set H5 + if the nth symbol exists, - if not. Assume list (0) describable, so that J81 finds symbol in first list cell, etc. J80 finds symbol in head; and sets H5- if (0) is a termination symbol.

J9n CREATE A LIST OF THE n SYMBOLS (n-1), (n-2), ..., (1), (0), $0 \leq n \leq 9$. The order is (n-1) first, (n-2) second, ..., (0) last. The output (0) is the name (internal) of the new list; it is describable. J90 creates an empty list (also used to create empty storage cells, and empty data terms).

J100 GENERATE SYMBOLS FROM LIST (1) FOR SUBPROCESS (0). The subprocess named (0) is performed successively with each of the symbols of list named (1) as input. The order is the order on the list, starting with the first list cell. H5 is always set + at the start of the subprocess. J100 will move in list (1) if it is on auxiliary.

J101 GENERATE CELLS OF LIST STRUCTURE (1) FOR SUBPROCESS (0). The subprocess named (0) is performed successively with each of the names of the cells of list structure named (1) as input. The order (called print order) is as follows:

1. List (0) is generated first.
2. All cells of a list are generated in contiguous sequence, starting with the head.
3. After a list has been generated, the sublists of the list structure that occur on the list are generated in the order they occur on the list.
4. Lower-level sequences of sublists occur after the higher-level sequence is finished, and are not interpolated.
5. Each list is generated only once, at the first opportunity.

The name of the cell is output to the subprocess as (0). H5 is set + if the cell is the head of a list (so that J101 is starting to generate a new sublist). In this case, J101 has already marked the sublist processed (J137), so that the head contains the processed mark and an internal zero. The original contents of the head are one-down in the list, and will occur as the next cell to be generated. In case the cell output to the subprocess is a list cell, H5 is set - .

J101 has available the name of the next cell to be generated prior to executing the subprocess (which determines how manipulations of the list structure by the subprocess will affect generation).

J101 cleans up the processing marks that it puts in the list structure, returning the list structure to its original state (except as modified by the subprocess). Structures whose names have been put by the subprocess in the empty heads created by marking processed, are not erased by the generator. (Note that J101 cannot be used in a subprocess to itself on the same list, because of the process marks.)

J101 will move in list structure (1) if it is on auxiliary.

J102 GENERATE CELLS OF TREE (1) FOR SUBPROCESS (0). The subprocess named (0) is performed successively with each of the names of the cells of the tree named (1) as input. A tree is a data list structure in which each sublist appears once and only once. The cells of each sublist are generated before going on with the superlist; the cell containing the name of the sublist occurs immediately before the sublist and all its sublists are generated. H5 is set + to the subprocess if input (0) is the head of a new sublist, and is set - otherwise. (Nothing is marked processed, since there is no need to keep track of multiple occurrences.) The name of the next cell to be generated is found before the cell is presented to the subprocess--i.e., it is possible to erase a tree with J102.

J102 will move in list structure (1) if it is on auxiliary.

J103 GENERATE CELLS OF BLOCK (1) FOR SUBPROCESS (0). (1) is assumed to be a block control word. The subprocess named (0) is performed successively with each of the names of the cells of the block (1) as input, generated in ascending order. H5 is always set + at the start of the subprocess. (See § 17.0, BLOCK HANDLING PROCESSES.)

10.0 AUXILIARY STORAGE PROCESSES, J105 to J109

There are two types of auxiliary storage--fast and slow--and two separate auxiliary storage systems--one for data list structures and the other for routines.

10.1 AUXILIARY STORAGE FOR DATA LIST STRUCTURES

The system for data list structures is patterned after a file drawer. The file holds data list structures. A list structure can be filed in auxiliary storage (it is the programmer's decision whether in fast or slow storage). When filed, the structure is no longer in main storage, and all the space it used is made available (except the head--see below). The programmer must be aware that he has filed a list structure in auxiliary, since most of the processes do not check for this. Thus, doing a J60, which locates the next symbol, on the name of filed list structure can only lead to chaos. The system determines where a list structure shall be filed, and records this information in the head of the list structure, which acts as a control word for the filed structure. The head remains in main memory. Thus, a list structure has the same name throughout a run, no matter how often it is shuffled between main and auxiliary storage: when it is in auxiliary, the head of the filed structure holds the control information to get the list structure back.

A filed list structure may be moved back into main storage, in which case it is no longer filed; its image, still occupying space in the auxiliary system, is considered an obsolete structure. A move can be done any time the name of the filed list structure is encountered, since the head holds the control information that locates it in auxiliary. It is also possible to copy or erase list structures in auxiliary using the regular list processes,

J74 and J72. Thus, the repertoire of processes for handling auxiliary storage of data list structures consists of the following processes:

J72 ERASE LIST STRUCTURE (0). (See definition in § 9.8, LIST PROCESSES.) J72 leaves an obsolete structure occupying auxiliary storage.

J74 COPY LIST STRUCTURE (0). (See definition in § 9.8, LIST PROCESSES.)

(See also J101 and J102.)

J105 MOVE LIST STRUCTURE (0) IN FROM AUXILIARY. The control word in cell (0) determines the location of the list structure, including whether it is in fast (Q = 6) or slow (Q = 7) storage. The list structure is returned to main storage, using words from available space, and the head replaced by the head of the list structure, so that the list structure is identical to itself prior to filing (except that different list cells are used). H5 is set + . If the list structure (0) was already in main storage (Q ≠ 6 or 7), J105 does nothing and H5 is set - . The output (0) is the input (0). J105 leaves an obsolete structure image occupying space in auxiliary storage.

J106 FILE LIST STRUCTURE (0) IN FAST-AUXILIARY STORAGE. Creates a copy of list structure (0) in a unit of the fast storage (the system selects unit and the space within the unit). Erases the list structure in main storage, except for head. Creates control word (Q = 6) and places it in the head. There is no output. (If there is no space in the fast-auxiliary, it is filed in the slow-auxiliary.)

J107 FILE LIST STRUCTURE (0) IN SLOW-AUXILIARY STORAGE. Identical to J106 except uses slow storage (Q = 7). (If there is no space in the slow-auxiliary, an error signal occurs; see § 21.0, ERROR TRAP.)

J108 TEST IF LIST STRUCTURE (0) IS ON AUXILIARY. Sets H5 + if (0) is on either fast- or slow-auxiliary, and H5 - in all other cases.

J109 COMPACT THE AUXILIARY DATA STORAGE SYSTEM (0). J109 purges the obsolete data structures from the auxiliary data storage system specified by the integer data term (0). The slow-auxiliary system is compacted if (0) = 0, the fast-auxiliary system if (0) = 1.

The system will automatically compact both fast- and slow-auxiliary data storage when they become full or inefficient. However, since compacting may become a time-consuming operation in some applications, the programmer has the option of assuming partial or complete responsibility for specifying when and how frequently it shall occur. The following system cells are relevant to compacting slow-auxiliary storage. Cells W39 through W42 perform the same function for the fast-auxiliary data system.

-W35 Holds the name of an integer data term which gives the number of obsolete structures currently occupying space in the slow-auxiliary data storage system.

-W36 Holds the name of an integer data term which gives the total number of structures, current and obsolete, occupying space in the slow-auxiliary data storage system.

-W37 Holds the name of an integer data term which the system interprets as the numerator of a fraction whose denominator is 100. When the ratio of obsolete to total structures in slow-auxiliary exceeds the above fraction, compacting will occur and all obsolete structures will be eliminated. 1W37 is initially 25, so compacting will occur when the number of obsolete structures is greater than 25 per cent of the total number of structures on slow-auxiliary storage.

-W38 Holds the name of the routine which compacts when necessary. W38 initially names a system routine which performs the test described above under W37, but may be replaced by the name of a programmer's IPL routine which determines when compacting should occur. 1W38 compacts by executing J109. 1W38 is executed automatically after every execution of J105 (Move List Structure (0) in from Auxiliary).

10.2 AUXILIARY STORAGE FOR ROUTINES

The auxiliary system for routines is used by the interpreter to bring routines into main storage for execution. It uses an auxiliary buffer into which all routines

stored in auxiliary (either fast or slow) are copied, and executed. All routines to be stored in auxiliary are assembled into this buffer during loading, so that no further assembly is needed to execute them once they have been brought in (see § 3.12, INTERPRETATION). Since all auxiliary routines use the same buffer, if an auxiliary routine uses an auxiliary routine, the copy of the higher one in main storage is destroyed when the lower one is called in. It is necessary to bring the higher auxiliary routine back into main storage again when the lower is finished. This leads to a "two call" system, in which every routine requires two reads from auxiliary storage: one to bring the routine in, and one to bring its predecessor in the auxiliary buffer back in. It is necessary to use a storage cell, the current auxiliary routine cell, H4, to keep track of the routines in the auxiliary buffer, since the nesting of auxiliary routines is unlimited. The symbols stacked in H4 are names of the control words, so the routines can be called back. When the routines in auxiliary storage are highly interdependent, the "two call" system is quite inefficient during execution. Much of this inefficiency can be eliminated by grouping those auxiliary routines which call on one another frequently into the same buffer-load. A buffer-load of auxiliary routines is created at loading time by preceding a set of routines with a single header card (TYPE = 6 or 7). The entire set of routines is loaded into consecutive cells of the buffer and written to auxiliary storage as a unit. The first call on any one routine in the set causes the entire buffer-load to be brought into main memory. Mutual calls between the routines in this buffer-load do not result in accesses to auxiliary storage; a call on a routine in a different buffer-load does. Any number of buffer-loads can be created while loading. A routine or group of routines too large for the buffer overflows into main memory via H2,

with no ill effects other than the expenditure of cells in main memory. The above considerations lead to the following restrictions:

- -No auxiliary routine shall modify itself in any way during execution. If it did, the call back from auxiliary would not be the same as the initial--and now modified--copy read in from auxiliary. (There are other reasons for not allowing self-modification--e.g., recursions.)
- -When a subprocess and its generator reside in different buffer loads, the auxiliary subprocess must be an independent routine--i.e., have a regional name--so that every time the generator executes the subprocess it can be brought in from auxiliary. If the subprocess were a sub-list-structure of the superroutine (with a local name), then when the buffer load containing the generator was brought in from auxiliary, it would destroy the copy of the superroutine--and with it, the subprocess--and chaos would result when the generator tried to execute the subprocess (see § 7.1, GENERATORS).

11.0 ARITHMETIC PROCESSES, J110 to J129

All the input and output symbols in this section are the names of data terms. Most operations admit only integers (P = 0, Q = 1) or floating point numbers (P = 1, Q = 1), but some admit any data term. In the arithmetic operations, if both factors are integers, then the result will be an integer. If either factor is floating point, the result will be a floating point number. Note that the prior nature of the cell holding the answer is immaterial. Thus, for example, J90 is used to create new result cells, even though it does not create data terms. None of the factors are affected by the operations, unless they are also named as the result. Any illegal operation--overflow, divide check, etc.--produces an error condition (see § 21.0, ERROR TRAP).

J110 (1) + (2)⟶(0). The number named (0) is set equal to the algebraic sum of the numbers named (1) and (2). The output (0) is the input (0); i.e., the result.

J111 (1) - (2)⟶(0). The number (0) is set equal to the algebraic difference between numbers (1) and (2). The output (0) is the input (0).

J112 (1) x (2)⟶(0). The number (0) is set equal to the low-order digits of the product of the numbers (1) and (2). The output (0) is the input (0).

J113 (1) / (2)⟶(0). The number (0) is set equal to the quotient of the number (1) divided by the number (2). The output (0) is the input (0). If division is integer division, then the remainder is the data term, W11 (consequently, the remainder is unsafe over divisions).

J114 TEST IF (0) = (1). Tests identity, including prefixes, of any two data terms, named (0) and (1). Hence will always give H5- if an integer is tested against a floating point.

J115 TEST IF (0) > (1).

J116 TEST IF (0) < (1).

J117 TEST IF (0) = 0.

J118 TEST IF (0) > 0.

J119 TEST IF (0) < 0.

J120 COPY (0). The output (0) names a new cell containing the identical contents to (0). The name is local if the input (0) is local; otherwise, it is internal.

J121 SET (0) IDENTICAL TO (1). The contents of the cell named (1) is placed in the cell (0). The output (0) is the input (0).

J122 TAKE ABSOLUTE VALUE OF (0). The number (0) is modified by setting its sign + . It is left as the output (0).

J123 TAKE NEGATIVE OF (0). The number (0) is modified by changing its sign--i.e., by multiplication by -1. It is left as the output (0). (Zero is signed; J123 takes zero into minus zero.)

J124 CLEAR (0). The number (0) is set to be 0. If the cell is not a data term, it is made an integer data term = 0. If a number, its type, integer, or floating point, is unaffected. It is left as the output (0).

J125 TALLY 1 IN (0). An integer 1 is added to the number (0). The type of the result is the same as the type of (0). It is left as the output (0).

J126 COUNT LIST (0). The output (0) is an integer data term, whose value is the number of list cells in list (0) (i.e., it doesn't count the head). If (0) = H2, J126 will count the available space list. This is the only place where H2 can be used safely by the programmer.

J127 TEST IF DATA TYPE (0) = DATA TYPE (1). Tests if P of cell (0) is the same as the P of cell (1). (Assumes (0) and (1) are data terms; hence, uses P of data term representation, which is not the same as P of instructions--see machine system write-ups.)

J128 TRANSLATE (0) TO BE DATA TYPE OF (1). The output (0) is the input (0), translated according to the data type of data term (1). This translation is not defined for all data terms. It will float integers (P = 0 to P = 1) and fix floating point numbers (P = 1 to P = 0). It can be expanded to include other P's (see machine system write-ups).

J129 PRODUCE RANDOM NUMBER IN RANGE 0 TO (0). The output (0) is a new number chosen from the uniform distribution over the interval 0 up to number (0) (the endpoint (0) is excluded). It is an integer or floating point number according to (0). It is produced by first generating a random number in the interval 0 up to 1, and then multiplying this number by (0). The random fraction is generated by multiplying the number named in storage cell W10 by a fixed number and taking the low-order digits. This new number is returned to W10 to become the factor in the next random number generated. Thus, starting W10 with a specified integer leads to a fixed sequence with random properties, which can be repeated. Different random sequences, such as are needed in statistical replication, are generated by starting W10 with different initial numbers.

Note that if the input is the integer n, the selection is from the n integers, 0, 1, ..., n-1, each with probability 1/n.

12.0 DATA PREFIX PROCESSES, J130 to J139

The reason for defining the data list structure as a unit of information is to allow processes that work for the list structure as a whole. We have processes like J72, erase a list structure; J74, copy a list structure; and J140, read a list structure into the computer. One erase process is sufficient to cover almost all possible types of data. It is desirable to be able to construct additional higher IPL routines that also work for list structures. To do this requires the ability to detect and manipulate the three kinds of symbols: regional, internal, and local. This is possible (for data only) since the Q prefix is used internally to encode the symbol with each occurrence. Upon loading data list structures (see § 13.0, INPUT-OUTPUT CONVENTIONS), the following coding takes place:

Q = 0	SYMB is regional.
Q = 1	Word is data term.
Q = 2	SYMB is local
Q = 3	Unassigned.
Q = 4	SYMB is internal.
Q = 5	Word is data term (same as Q = 1).
Q = 6	P = 1: List structure is in fast-auxiliary storage.
Q = 7	P = 1: List structure in is slow-auxiliary storage.
P = 0	For all standard IPL words, and as assigned for data terms.

The only values of Q and P that appear externally are those connected with data terms. We give the others here to make it clear what processes are being performed with the data prefix processes; details can be found in the machine system write-ups.

12.1 RECURSIONS

Besides the processes mentioned above, it is necessary

to be able to work on all parts of the list structure--e.g., in an erase, every cell must be erased. The basic technique in processing list structures is recursion. Since a list structure is recursively defined, the kind of operations that can be defined for a list structure involve defining what is to be done to each list of the structure and then recursing through the structure. That is, the total process has the form:

-Do what you have to to this list;
-Find all the local names on this list;
-Do the total process to each sub-list-structure defined by these local names.

Eventually, all the lists in the list structure get processed and the recursion will stop; the recursive character of the routine and the fact that all connections in the structure are marked by local names assures this. Since, however, the name of a list can occur in many places in a list structure, there must be some device for avoiding multiple processing of the same list if this is not desired (and it must not be allowed for list structures which allow the name of a list to appear on one of its sublists).

For example, in erasing a list of lists which consists of three occurrences of the same sublist--e.g., L1: 9-1, 9-1, 9-1--the sublist, 9-1, must be erased only once, not just as a matter of efficiency, but because chaos will result if an erased list is erased.

12.2 MARKING A LIST PROCESSED

The solution provided in the basic system to keep track of multiple processing is a technique for marking a list "processed": J137 (taking the name of a list as input) preserves the list, makes the head empty (Q = 4, SYMB = 0), and marks it with P = 1. Since throughout the rest of the data P = 0, it is possible to detect if the sublist

has already been processed by testing whether P = 1 (J133). The mark can be removed and the list returned to its initial condition by a restore. The empty head can hold temporary information relevant to each sublist during a list structure process. For example, a new temporary description list could be put in the head. It would not get mixed up with the normal description list, which is one-down in the push down list. Of course, this temporary description list must be cleaned up at the end, say by J15.

It is possible to avoid some of the problems of keeping track of list structures by using J101, the generator of the cells of a data list structure. J101 uses the device of marking processed--every sublist is marked processed when first presented--but much of the mechanics is buried in J101, and need not be repeated by the subprocess that uses it.

J130 TEST IF (0) IS REGIONAL SYMBOL. Tests if Q = 0 in HO.

J131 TEST IF (0) NAMES DATA TERM. Tests if Q = 1 or 5 in the cell whose name is (0).

J132 TEST IF (0) IS LOCAL SYMBOL. Tests if Q = 2 in HO.

J133 TEST IF LIST (0) HAS BEEN MARKED PROCESSED. Tests if P = 1 (and Q ≠ 1 or 5) in the cell whose name is (0). It will only be 1 if list (0) has been preserved and P = 1 put in its head by J137. This means list (0) has been marked processed.

J134 TEST IF (0) IS INTERNAL SYMBOL. Tests if Q = 4 in HO.

J136 MAKE SYMBOL (0) LOCAL. The output (0) is the input (0) with Q = 2. Since all copies of this symbol carry along the Q value, if a symbol is made local when created, it will be local in all its occurrences.

J137 MARK LIST (0) PROCESSED. List (0) is preserved, its head made empty (Q = 4, SYMB = 0), and P set to be 1. Restoring (0) will return (0) to its initial state. This will work even with data terms. The output (0) is the input (0).

J138 MAKE SYMBOL (0) INTERNAL. The output (0) is the input (0) with Q = 4. Best considered as "unmake local symbol."

13.0 INPUT-OUTPUT CONVENTIONS

Input and output comprise several pieces: initial loading, translation from one representation to another; reading data list structures during running; writing data list structures created during running so they can be reloaded; printing; and monitoring the running program. All of these utilize common conventions about format and designation of units.

13.1 EXTERNAL TAPES

It is possible to use tapes for input and output, rather than the on-line card readers, punches, and printers. Such tapes are called external tapes to distinguish them from the tapes used for auxiliary storage. An external tape is functionally identical with a deck of cards outside the IPL computer. It consists of a sequence of independent list structures. External tapes can be generated in one run and used in a different run. External tapes are not generally compatible across different types of machines (but see machine system write-ups for details). Tapes can be used as intermediate storage, since tapes written by the write processes can be read back in by the read processes. An external tape can hold information in any of the representations defined below. (External tapes are also used as intermediate storage of blocks of information; see § 17.0, BLOCK HANDLING PROCESSES.)

13.2 INPUT-OUTPUT UNIT CODE

The units used for input and output are named by small integers as follows:

0 The "normal" value for an installation. This will depend on the operating system being used at the installation and the kind of machine. It will include on-line card read and punch for some signal from the console.

1-10 External tapes. The connection between these names and physical units is again dependent on the machine and the installation.

The machine system write-ups should be consulted for more information.

13.3 INPUT-OUTPUT REPRESENTATION MODE

The information being input and output is in one of several modes, each of which has an integer code:

0 = IPL standard (one IPL word per card, as represented on the coding sheet).

1 = IPL compressed (about 7 IPL words per card).

2 = IPL binary (about 20 IPL words per card).

3 = Machine code.

4 = Restart mode (see § 20.0, SAVE FOR RESTART).

5, 6, 7 = Machine language for various object machines. See machine system write-ups for further details.

13.4 IPL COMPRESSED REPRESENTATION

See machine system write-ups for information.

13.5 IPL BINARY REPRESENTATION

(See machine system write-ups for further information.)

The information is put on the card in column binary, although the notation used is as if it were row binary--e.g., 9L is the 36-bit word in the left half of the 9 row of the card. The 9 row is special:

9LP = 6 (= 7 if wish to ignore checksum).

9LD = v + 500_8, where v = word count and is, at most, 22.

9LA = sequence number of card in deck.

9R = checksum = (9L) + (8L) + ... + (v^{th} information word).

All the v information words, starting with 8L and working back, are considered one long string of bits. The string is divided up into units by the following heading code and convention:

Heading code (bits)

0 = End of list.

10 = IPL word: followed by Q LINK P SYMB NAME.

11 = Data term: followed by Q P DATA NAME.

P and Q each coded into 3 bits.

NAME, SYMB, LINK, each coded into 1 bit (= 0) if blank; or into 6-bit region plus 15-bit relative number if not blank.

DATA is coded into 30 bits.

14.0 READ AND WRITE PROCESSES, J140 to J146

These are processes that allow the input and output of data list structures during running, under the control of the program. Only data list structures, not routines, can be input or output by these processes. The form of the data list structures is identical to that of initial loading, and may be in any of the three modes of representation: IPL standard, IPL compressed, or IPL binary (if possible for the object machine). A safe storage cell, W16 for reading and W17 for writing, determines the mode. The symbol in the cell is the name of the integer data term giving the code stated earlier. The list structures are handled independently, and not as sets (as in initial loading), and no header cards are used. No translation, assembly listing, or direct input to auxiliary (all inputs being to main storage) is possible. A structure may be loaded into a specific block of main storage, however, (see § 18.5, TYPE = 5, 6, 7, 8: HEADER CARDS). The unit to be used must be selected, and safe storage cells, W18 for read and W19 for write, are used for this. The symbol in the cell names the integer data term giving the unit (see § 13.2, INPUT-OUTPUT UNIT CODE).

J140 READ LIST STRUCTURE. A list structure on cards (or external tape) in any of the admissible forms (IPL, compressed, binary) is read into the main storage cells taken from the available space list 1W34, its name input to (0), and H5 set + . Blank records are treated as end-of-list-structure marks. (End-of-list-structure is also signaled by an input end-of-file condition or by the start of a new list structure, with a regional or internal name.) If the first record read by J140 is blank, it is ignored. If there is no list structure (card hopper empty or end-of-file) then there is no input and H5 is set - . Internal symbols are assumed to already exist in the IPL computer: internal symbol 1345 is assigned address 1345.

J141 READ A SYMBOL FROM CONSOLE. Inputs a symbol or data term from the console into H0. Sets H5 + if there is an input, and - if there is not. An input data term is put in a new cell and given an internal name.

The console conventions depend on the particular machine, and the machine system write-ups should be consulted for the exact definition of J141.

J142 WRITE LIST STRUCTURE (0). (0) is assumed to name a list structure. It is punched (or written on external tape) in any of the admissible forms (IPL, compressed, IPL binary). Regional symbols are converted back to external form, adddd; internal symbols are converted directly--address 1345 to symbol 1345; and local symbols are expressed as 9dddd, where the dddd are small integers. The order of writing is that of J101, so that all the symbols of a list are written consecutively. Thus, there is no need for local names for list cells--i.e., no link is needed except for 0, the termination symbol.

J143 REWIND TAPE (0). The external tape named by the data term (0) is rewound.

J144 SKIP TO NEXT TAPE FILE. The external tape named in W18 is positioned past the next end-of-file mark.

J145 WRITE END OF FILE. The end-of-file mark is written on the external tape named in W19.

J146 WRITE END OF SET. A blank record (appropriate to mode 1W17) is written on the external tape named in W19. (See § 18.0, INITIAL LOADING, for use of blank records.)

15.0 MONITOR SYSTEM, J147 to J149

Three kinds of facilities are available for monitoring the running program and controlling it. First, it is possible to take a "snapshot" of the program to see what it is doing. Second, it is possible to get "post mortem" information after a program has stopped. Third, it is possible to trace the program, printing information on each instruction as it is executed. The machine system write-ups should be consulted on the conventions for using the console to accomplish the features described below.

15.1 MONITOR POINT, Q = 3

Any instruction with Q = 3 is called a monitor point in the program. As far as execution of the program is concerned, it is treated as Q = 0. However, when it is encountered, the interpreter takes the following monitoring action:

- -It turns the trace "on," also marking that a monitor point has occurred.
- -It pushes down the safe storage cell W29 and stores the current instruction address (the name of the cell holding the instruction with Q = 3) as 1W29.
- -It checks whether the number of cells of reserved available space is equal to 1W32. If unequal, it adjusts the supply of cells to equal 1W32.
- -It checks the console for the following signals:
  - -External interrupt: if the external interrupt signal is present, the routine named in the safe storage cell W14 is executed and the program continues.
  - -External trace mode: no trace, selective trace, full trace. (If there is no external trace signal from the console, the external trace mode is set according to 1W31.)
- -Finally, it executes the routine named in the safe storage cell, W12, and then continues the program.

-When the program list in which Q = 3 occurred is finished--i.e., when the marked routine is finished--it executes the routine named in the safe storage cell, W13.

-It then pops up W29 and continues with the program.

It is normal to mark a routine by putting the monitor mark in the head.

15.2 SNAPSHOTS

W12 and W13 hold snapshot routines. As seen above, they will be executed under various conditions associated with the monitor points, Q = 3. There is no restriction on the routine that may be executed, although the normal use is to print out various lists to see how the program is progressing.

The snapshot mechanism is operative at monitor points regardless of the trace mode or external trace conditions. The snapshot cells (W12 and W13) initially contain J0, meaning "no operation."

15.3 EXTERNAL INTERRUPT

The system checks for the presence of an external interrupt signal at each monitor point. If the signal is present, the routine named in the safe storage cell W14 is executed and the program continues. Setting a console switch manually is the normal way of providing an external interrupt signal, but see the machine system write-ups for additional or alternative methods.

There is no restriction on the nature of the routine 1W14. In particular, terminating 1W14 with J166 will save for restart and continue with the program. Terminating 1W14 with J7 will terminate the program without providing for restart. To terminate the program and provide for restart, see the example in § 20.0, SAVE FOR RESTART.

15.4 POST MORTEM

In the event the system detects some internal error while executing a program, it automatically prints out information about the terminating condition of the machine via J202 and then stops. J202 may also be executed directly by the programmer any number of times during a run. W23 holds the name of the list specifying information to be printed by J202. This list may be modified by the programmer. W15 holds the name of a routine that J202 executes after the other information has been printed. Any routine may be executed except J202. Its primary use is to select and print debugging information that cannot be specified on the 1W23 list. W15 initially holds JO.

J202 PRINT POST MORTEM AND CONTINUE. Print as defined for the particular machine system.

15.5 TRACING

There are two internal trace modes, "on" and "off." In addition, there are three externally imposed conditions: no trace, in which the trace mode is "off" no matter what is indicated internally; selective trace, in which the trace mode is as indicated internally; and full trace, in which the trace mode is "on" no matter what is indicated internally.

The three externally imposed trace conditions (no trace, full trace, and selective trace), may also be imposed internally by the integer data term named by W31. The code for W31 is:

0 = No trace;
1 = Full trace;
2 = Selective trace.

1W31 is set for selective trace initially. The programmer may change 1W31 anytime. The change becomes effective when the next monitor point is encountered. If the trace mode is on, then for each instruction the following information is printed:

-Level number, counting down from the initial routine as level 1.

-CIA, the current instruction address (the symbol in H1).

-Test signal, the contents of H5 (+ or -) prior to execution.

-Instruction being executed, PQ SYMB LINK (the contents of CIA).

-S, the designated symbol.

-(0), the symbol in H0 prior to execution.

-The contents of cell (0), printed in appropriate form (data term or PQ SYMB LINK).

-H3, the number of interpretation cycles since H3 was last reset. (H3 will include one count for each line of trace that would have printed had full trace been on.)

The format is as follows:

←——Level CIA——→H5 P Q SYMB LINK S (0) CONTENTS H3

The level and CIA are indented according to the level, modulo the printing interval available. The symbols are translated back into IPL representation (this is not possible on all machines). The Q of (0) is printed, indicating whether the symbol is internal or local.

15.6 TRACE MARKS

The trace mode is carried by a mark in H1. This mark encodes whether the trace mode is on or off, and also whether a monitor point occurred. On selective trace, the interpreter consults this mark each cycle (after INTERPRET Q but before INTERPRET P) and if it reads on, prints the trace information. This mark is governed by the occurrence of Q = 3, and Q = 4, in the instructions of the program. Both of these are treated as Q = 0 in determining the designated symbol. The following rules describe their function:

-If a Q = 3 is encountered, set trace on.

-If the trace is on, it remains on as we advance along a program list (always at the same level)--i.e., the trace mark propagates down a list.

-When the program descends a level, the trace is always off, a priori--i.e., the trace mark does not propagate down levels.

-If a Q = 4 is encountered, the trace mark is set to equal the trace mark one level up--i.e., the trace is propagated down a level by Q = 4.

-In ascending, H1 is restored and the trace mark of the higher level again becomes operative.

These rules mean the following: putting Q = 3 in the head of a program list will cause that list to be traced. Putting Q = 4 in the head of a program list will cause that list to be traced, if the program list calling upon it is tracing. Hence, putting Q = 4 in the heads of all local sublists of a routine, makes the routine a tracing unit: all instructions of the routine will trace if Q = 3 in the head of the routine; the whole routine will trace conditionally if Q = 4 is put in the head; and none will trace if Q $\neq$ 3 or 4 in any instruction.

Where generators are involved, the superroutine and subprocess are on the same level; the subprocess will trace without being marked, provided the superroutine is tracing. The generator is down one level from the superroutine; hence, if marked with Q = 4, the generator will trace when the superroutine is tracing.

The Q's can be written in the routines at the time of coding by the programmer. Since Q = 3 and 4 are equivalent to Q = 0, they can often be put in without adding space to the system. If the head of a routine does not have Q = 0, then an additional instruction, say with SYMB = JO, is necessary. Since the routines that are traced are changed often, it is desirable to specify the Q's at the beginning of each run, without permanently marking the routines. This can be done by means of three IPL processes:

J147 MARK ROUTINE (0) TO TRACE. If Q = 0, 3, or 4 in cell (0), changes Q to be 3. If not, preserves (0), and places the instruction 03 J0 in cell (0).

J148 MARK ROUTINE (0) TO PROPAGATE TRACE. Identical to J147, except uses Q = 4.

J149 MARK ROUTINE (0) NOT TO TRACE. If Q = 3 or 4 in cell (0), puts Q = 0, unless SYMB is also J0 and P = 0, in which case J149 restores (0). If Q $\neq$ 3 or 4, does nothing.

16.0 PRINT PROCESSES, J150 to J162

Two classes of printing processes are provided, those for printing IPL units of data (symbols, lists, list structures, data terms) and those for composing and printing a line of information. Each of the printing processes is relative to:

- -The unit that will print, given by the integer data term named in the safe storage cell W20. (See § 13.2, INPUT-OUTPUT UNIT CODE.)
- -The column in which the leftmost character of the format will print, given by the integer data term named in the safe storage cell W21. The columns run from 1, at the far left of the page, to 120 at the right.
- -The line spacing that will occur between a line and the previous printing, given by the integer data term named in the safe storage cell W22. The spacing code is the following:
  - 0 If spacing is suppressed--i.e., print on the same line;
  - 1 If start printing on the next line;
  - 2 If skip one line before starting to print;
  - 3 If skip to next page, and start printing at the top.

Not all the object machines have the full flexibility, so the machine system write-ups should be consulted.

16.1 PRINTING IPL UNITS OF DATA

J150 PRINT LIST STRUCTURE (0). The contents of all the cells of the data list structure named (0) are printed. Regional symbols are translated to the form adddd; internals are printed as the decimal integer corresponding to the address; and local symbols are translated to the form 9dddd, where dddd are small integers. All data terms are translated to their external form. If input (0) is a block control word or the head of a structure on auxiliary, only the word (0) itself is printed. Each list of the list structure is printed in an uninterrupted vertical column, so that neither LINK nor the NAME of any list cell is ever printed. If the SYMB names a

data term, then this data term is printed to the right on the same line. If the NAME is a local name (which can occur only in printing the head of a sublist), its corresponding address is printed to the left. The local name, 9dddd, bears no relation to this address. The full format is shown below. (Column 1 corresponds to the column specified by the integer data term named in W21.)

-column:	12345	67	89111 012	111 345	11112 67890	2222 1234	22 56	22233333333 78901234567
	addr. of NAME if local		NAME	PQ	SYMB		PQ	DATA if SYMB names data term

The lists of the list structure are printed in the order of J101.

J151 PRINT LIST (0). The contents of all the cells of the list named (0) are printed in an uninterrupted vertical column. The format is the same as that of J150, except that local symbols are not translated to form 9dddd; but instead, their addresses are printed, and the Q = 2 identifies them as locals. If input (0) is a block control word or the head of a structure on auxiliary, only the word (0) itself is printed.

J152 PRINT SYMBOL (0). The symbol (0) is printed. The format is the same as J150, where (0) is placed at SYMB, and if it names a data term, this is printed to the right. Locals are handled as in J151.

J153 PRINT DATA TERM (0) WITHOUT NAME OR TYPE. (0) is assumed to name a data term (if not, nothing is printed and the designated spacing occurs). The DATA part of the data term is printed in its location in the format of J150, but neither (0) nor the PQ of the data term is printed.

16.2 LINE PRINTING

In addition to the output unit, left margin, and line spacing controls given previously, line printing is controlled by:

-The current print line, named by the symbol in the safe storage cell W24. Print lines are reserved during loading (see § 18.3, TYPE = 3: BLOCK RESERVATION CARDS), when the symbol naming the line and the size of the line are specified. All print lines start with column 1; the specified line size determines the right margin of the line.

-The current column at which information will be entered in the current print line, given by the integer data term named in the safe storage cell W25. Information can be entered either left-justified--1W25 specifying the position of the leftmost character of the field being entered--or right-justified--1W25 specifying the position of the rightmost character of the field. After an entry, 1W25 is set to the next column following the rightmost character of the field entered, and H5 is set + . If the entire field cannot be entered because it would exceed the line size, no information is entered, 1W25 is left unchanged, H5 is set - , and H0 no longer holds the input.

Symbols are entered in the print line compactly; i.e., as A1, B10, etc. (A0 is entered as A). Data terms are entered as follows:

Integers: Leading zeros are eliminated. Plus signs are not entered, but minus signs are. Examples: "00273" entered as "273" (3 cols.); "-01050" entered as "-1050" (5 cols.).

Floating Point: The entire number is entered, signed value followed by signed exponent. Only minus signs are entered. Examples: ".505135x10^{5}" entered as "505135 05" (9 cols.); ".14x10^{-16}" entered as "140000 -16" (10 cols.).

Alphanumeric: Trailing blanks--that is, blanks that follow some non-blank character and are not followed by some non-blank character--are eliminated. Example: "_A_F_" entered as "_A_F" (4 characters); "_____" entered as "_____" (5 characters).

All Other: The entire value of the data term is entered as a ten-digit octal integer. Example: "0000567234" entered as "0000567234".

J154 CLEAR PRINT LINE. Print line 1W24 is cleared and the current entry column, 1W25, is set equal to the left margin, 1W21.

J155 PRINT LINE. Line 1W24 is printed, according to spacing control 1W22. The print line is not cleared.

J156 ENTER SYMBOL (0) LEFT-JUSTIFIED. Symbol (0) is entered in the current print line with its leftmost character in print position 1W25, 1W25 is advanced to the next column after those in which (0) is entered, and H5 is set + . If (0) exceeds the remaining space, no entry is made and H5 is set - .

J157 ENTER DATA TERM (0) LEFT-JUSTIFIED. Data term (0) is entered in the current print line with its leftmost character in print position 1W25, 1W25 is advanced, and H5 is set + . If (0) exceeds the remaining space, no entry is made and H5 is set - .

J158 ENTER SYMBOL (0) RIGHT-JUSTIFIED. Symbol (0) is entered as in J156, except that 1W25 names the print position of the rightmost character of the field. If entry is possible, 1W25 is advanced and H5 is set + ; if not, H5 is set - .

J159 ENTER DATA TERM (0) RIGHT-JUSTIFIED. Data term (0) is entered as in J157, except that 1W25 names the print position of the rightmost character of the field. If entry is possible, 1W25 is advanced and H5 is set + ; if not, H5 is set - .

J160 TAB TO COLUMN (0). (0) is taken as the name of an integer data term. Current entry column, 1W25, is set equal to 1W21 + (0).

J161 INCREMENT COLUMN BY (0). (0) is taken as the name of an integer data term. Current entry column, 1W25, is set equal to 1W25 + (0).

J162 ENTER (0) ACCORDING TO FORMAT 1W43. The name and contents of cell (0) are entered after having been converted to an appropriate external representation (e.g., octal), specified by the data term in W43. 1W25 and H5 are treated as in J156. J162 is intended primarily to provide dumps of blocks when used with J103. (See machine system write-ups for the various formats and conversion schemes available.)

In addition to lines composed using these primitives, complete headings and partial lines can be specified at loading (see § 18.3, TYPE = 3: BLOCK RESERVATION CARDS).

17.0 BLOCK HANDLING PROCESSES, J171 to J179

In order to deal effectively with programs which exceed the main storage capacity of the computer several times over, it is necessary to have techniques for dealing with blocks of main storage. A block control word is a cell with P = 7 and Q = 7, whose SYMB specifies the origin of a continuous block of memory cells, and whose LINK specifies the number of cells in the block.

Since a region is represented in the computer by a block of cells, there is a block control word for each of the 36 possible regions definable by the IPL-V programmer. We will refer to a block control word for a region as a region control word hereafter. The 36 region control words are a permanent part of the system; their names are only obtainable via J175. They are used by the system to translate the external representation of regional symbols (e.g., R15) into computer addresses during loading, and to translate regional cell names back into their external regional representation during output. The programmer may copy region control words, but should never modify them; changing the contents of a region control word effectively redefines the region it controls.

The programmer may define and name blocks of space, other than regions, with Type-3 cards; the symbol which names the block is made the control word for the block in this case.

A block of space, including a region, may be turned into a list, which may be used by the loading processes as an available space list. This ability to load into specific blocks, coupled with fast processes to read and write the contents of blocks on tape, allows overlay techniques for problems too large to be performed economically in a single phase.

J171 RETURN UNUSED REGIONAL CELLS TO H2. J171 scans all region blocks for unused cells and returns them to the end of H2. "Unused" means that the regional symbol has not appeared in any NAME, SYMB, or LINK field of the input deck and the cell does not lie within a block reserved by any Type-3 card. J171 modifies the region control word so that the highest symbol used becomes the last cell of the region after J171; unused symbols higher than this symbol lose their regional status; unused cells lower than this symbol retain their regional status for the input and output processes and for J175 and J201.

J172 MAKE BLOCK (0) INTO A LIST. Input (0) is assumed to be a block control word. Output (0) names the head of the list and is the name of the first cell of the block. The cells of the block are linked in ascending order; their P, Q, and SYMB are unchanged. (J172 provides a way to turn a block into a list. The list may then be added to H2 by J71, or used as special available space for loading, by putting its name in W34.)

J173 READ NEXT BLOCK FROM TAPE 1W19 INTO BLOCK (0). Sets H5+ if successful. If first word fails to match the contents of (0), J173 traps on trap attribute 'J173', with first word of block named in HO. Sets H5-, gives message, and traps on attribute 'HO' if (0) is not a block control work; i.e., P and Q are not equal to 7.

J174 WRITE BLOCK (0) ONTO TAPE 1W19. Sets H5+ if successful. Same procedure as J173 if (0) is not a block control word. The first word written on tape is not the first word of the block, but is the contents of the input block control word (0). This is used by J173 to detect reading of information into a block other than that from which it was written.

J175 FIND REGION CONTROL WORD OF REGIONAL SYMBOL (0). If (0) is a regional symbol, H5 is set + , and output (0) is the name of the block control word for the region. H5 is set - and there is no output if input (0) is not a regional symbol. A symbol is a regional symbol to J175 if the cell which it names lies within one of the blocks defined by the 36 region control words.

J176 SPACE (0) BLOCKS ON UNIT 1W19. (0) is assumed to be a signed integer data term. Tape 1W19 is spaced (0) blocks in the direction indicated by the sign of (0). Plus indicates forward spacing, minus specifies backspacing.

18.0 INITIAL LOADING

To use IPL, the computer must first be turned into an IPL computer by loading the IPL interpretive system, either from cards or tape. Then the IPL computer must load the user's program into the total available space. This requires a deck of cards (or external tape) containing the IPL words, as well as some special cards to identify the program and to define the regional symbols that are used in the program. These special cards are called type cards, and they are identified by a non-zero digit in the TYPE column (column 41). The cards that have been described up till now have all been Type-0 cards (TYPE may be left blank on Type-0 cards). The following additional types are recognized.

18.1 TYPE = 1: COMMENT CARDS

All columns (except 41) are available for anything the programmer wishes to write. Comment cards are listed on the assembly listing, but have no other effect on the loading process.

18.2 TYPE = 2: REGION CARDS

All the regional symbols with the same initial letter constitute a region. Each region is represented in the computer by a block of consecutive cells. For example, the R-region might correspond to the block of cells 1000 to 1018: then R0 would correspond to 1000, R1 to 1001, and R18 to 1018. The size of each region must be specified at loading time by a Type-2 card. One Type-2 card is used for each region. The first symbol of the region--e.g., R or R0--is put in the NAME field, SYMB is left blank, and the number of cells in the block is put in LINK. The initial loader assigns the next available block of

contiguous cells to this region and records the origin and size of the block in the region control word. Thus, the origin of a region block is assigned arbitrarily. There is normally no need to know the origin, since all regional symbols are translated back into the letter-number form for output. However, for some purposes it may be desirable to specify the origin. This is done by placing the absolute address of the origin in SYMB. The origin can also be specified symbolically in terms of another region, provided the other region is first defined. (See machine system write-ups for further details.)

Examples:	TYPE	NAME	PQ	SYMB	LINK
Ten symbols for the M region M0 to M9:	2	M0		1000	10
Starting the M region at address 1000:	2	M0		1000	10
Making M0 synonymous with B37:	2	M0		B37	10

There are 36 possible regions:

A B C D E F G H I J K L M N O P Q R
S T U V W X Y Z + - / = . , $ *) (

Three regions, H, J, and W, have already been permanently specified for the basic system. Also, the $ region is to be used for system routines unique to particular installations (see § 4.1, SYSTEM REGIONS). The first symbol of all those regions which the programmer does not define with Type-2 cards is automatically defined and reserved by the initial loader when the first header card (TYPE = 5, 6, 7, or 8) is encountered. This allows the programmer to use the line read primitives on English text without having to define all 36 regions explicitly (see § 22.0, LINE READ PROCESSES). All the regional symbols that are not actually used during loading--i.e., do not occur as some NAME, SYMB, or LINK on the coding sheet and have not been read by J181 (Input Line Symbol) during processing--may be made

part of the available space for the IPL computer by executing J171. All regional symbols mentioned (in SYMB or LINK) but not defined (in NAME) are used but empty. If the exact limits of regions are specified, then the blocks of cells corresponding to different regions may overlap and need not be contiguous. If origins are assigned by the IPL computer, the region blocks are adjacent and disjoint. The block control word for a region may be obtained by J175.

18.3 TYPE = 3: BLOCK RESERVATION CARDS

It is necessary to create blocks of space for various purposes, and sometimes desirable to set a number of regional symbols to be empty without mentioning them. Type-3 cards are used to accomplish this. As in Type-2 cards, SYMB indicates the base, if appropriate, and LINK indicates the size of the block. The initial loader creates a block control word in the cell mentioned in the NAME field of all Type-3 cards. Q is used to indicate the purpose of the block, according to the following table:

Q = 0 RESERVE REGIONAL SYMBOLS. If SYMB is A5 and LINK is 10, then A5 through A14, inclusive, are set empty, and will not be put back on available space by J171. If NAME is B20, then B20 is a block control word for the block A5-A14. The symbols reserved must have previously been covered by a Type-2 card.

Q = 1 RESERVE PRINT LINE. NAME is the regional symbol naming the line (i.e., the block control word). LINK is the number of words to be set aside for the print line. (These words are taken from available storage, not from the region. See machine system write-ups for details of how many characters are stored per word in a particular machine.) If P is not 0 or blank, the immediately following record is a BCD record to be loaded into the block starting with column 1, into the first character position, and continuing to the end of the block.

Q = 2 RESERVE BLOCK. The regional symbol appearing in the NAME field is the name of the block, (i.e., the block control word). LINK is an integer specifying the number of cells in the block. The size of any one block is limited by the size of the machine. Blocks may overlap one or several other blocks, completely or partially, including blocks of regional cells or even blocks of code which make up the IPL-V Loader, Interpreter, or Monitor, if this is useful. Any number of blocks may be reserved. In general, a block control word should not lie within the block which it controls.

Q = 3 RESERVE AUXILIARY ROUTINES BUFFER. This reserves a block of size LINK (starting at SYMB, if given). LINK is the number of cells, and SYMB, if given, specifies the origin of the block. NAME is optional. Only one such buffer may be reserved. This buffer is used by all the routines on auxiliary storage. Its size limits the maximum size of a routine on auxiliary (but see § 10.0 AUXILIARY STORAGE PROCESSES).

Q = 4 SPECIFY AVAILABLE SPACE. If this card is absent from the loading deck, or if it is present with LINK blank, all the available space possible will be assigned to H2. This includes all interstices between blocks, if any, which would go at the end of available space. LINK, when present, specifies the number of cells that will be provided, in one continuous block if possible. NAME, specifying a block control word, is optional. If a large enough block is not available, a message is given, but the block control word is not modified.

18.4 TYPE = 4: LISTING CARDS

Type-4 cards represent printed output from computers which must output via cards and therefore require a way of distinguishing printed output (J150's) from punched output (J142). They are generated by the computer, and not by the programmer. If input, they are listed on the assembly listing, but have no other effect on loading.

18.5 TYPE = 5, 6, 7, 8: HEADER CARDS

Data or routines are loaded in a series of separate sets, each of which is preceded by a header card that governs the loading process. The input set may be in one of several modes: IPL standard (one word per card); IPL compressed; IPL binary; or one of the machine codes. It may also come from one of several input units: tapes or the card header. It is possible to specify an output during initial loading, which serves the purpose of translating from one form, such as IPL standard, to another, such as IPL binary, for subsequent use. An assembly listing is usually produced during loading, to indicate the machine location assigned to each IPL word in order to facilitate debugging. This may be suppressed, if desired.

The set may contain either routines or data, and it is necessary to specify which, as the P and Q codes are treated differently. Also, the set may go into main storage (TYPE = 5), may go to one of the auxiliary storages (TYPE = 6 for fast, TYPE = 7 for slow), or may be skipped (TYPE = 8). Structures going into main storage may be loaded into cells from the standard available space list, H2, or may be loaded into a specific block. Data list structures are loaded to auxiliary in relocatable form; auxiliary routines are assembled into the single auxiliary routines buffer and written to secondary storage in non-relocatable form.

Loading into a specific block of main storage is accomplished by the use of the safe storage cell W34. W34 holds the name of the available space list used by the loading processes (initial loading, J140 and J165) and initially holds H2. To load into a specific block it is necessary to make the cells of the block into a list with J172 and put the name of this list into the safe storage cell W34. Sets of data or routines preceded by

Type-5 cards with NAME = blank will then be loaded into the cells of the block. The first cell of the block is never loaded into since, like H2, it is the head of the available space list. If the list 1W34 becomes exhausted, H2 is placed in W34 without push down, an error message is given, and loading continues from H2.

The loader will automatically set the name of the desired available space list into W34 if it encounters a Type-5 card with NAME = name of block. It preserves W34 and places the name of the first cell of the block into W34; the associated set of routines or data is loaded and W34 is restored when the next header card (TYPE = 5, 6, 7, or 8) is encountered. The block mentioned in the NAME field must have been made into a list (J172) at some previous time.

A Type-8 editing header inhibits the loading of its associated set of routines or data, but allows the listing and output options. It is intended for use on the controlling unit to skip over unwanted sets on an alternate unit. (See § 18.7, CONTROLLING AND ALTERNATE INPUT UNITS.)

Finally, a Type-5 card is used to specify that loading has finished, and to indicate where the program starts.

The codes for these various items of information are given in the following table:

TYPE: Type of storage to be used:
5 = Main storage
6 = Fast-auxiliary storage
7 = Slow-auxiliary storage
8 = Inhibit loading--permits listing and output options.

NAME: Name of storage block:
NAME = Blank, TYPE = 5: Load into the main memory cells taken from the current available space list, 1W34. 1W34 is initially H2.

NAME = Regional symbol, TYPE = 5: Name is assumed to be a block control word whose SYMB names a previously constructed available space list. Preserve and set W34 = SYMB, and load the set into the main memory cells taken from the list named SYMB, and restore W34 when the next header is encountered.

NAME = Anything, TYPE = 6: (NAME is ignored.) Load each data list structure to fast- or slow-auxiliary in relocatable form. Load routines to fast- or slow-auxiliary in non-relocatable form, each routine origined one cell beyond the end of the immediately preceding routine. The first routine in the set is origined at the first cell of the auxiliary routines buffer. (See § 10.2, AUXILIARY STORAGE FOR ROUTINES.)

P: Input Mode:
0 = IPL standard (1 word per card)
1 = IPL compressed
2 = IPL binary
3 = Machine code
4 = Restart mode
5, 6, 7 = Machine dependent modes for various object machines. See machine system write-ups for details

Q: Type of Input:
0 = Routines. Internal symbols are considered pure symbolics. Undefined internal symbols (internal symbols not in the internal symbol table) are assigned equivalents from available space (0-9 are always defined and absolute).
1 = Data list structures. Internal symbols are considered pure symbolics. Undefined internal symbols are assigned equivalents from available space.
2 = Routines. Internal symbols are considered pure symbolics. The internal symbol table is reset (thus undefining all internal symbols) and undefined internal symbols are assigned equivalents from available space.
3 = Data list structures. Internal symbols are considered pure symbolics. The internal symbol table is to be reset and undefined internal symbols are to be assigned equivalents from available space.
4 = Routines. Internal symbols are considered machine addresses (and so no equivalent need be assigned). Such internal symbols do <u>not</u> start a new list structure.

5 = Data list structures. Internal symbols are considered machine addresses and do not start new list structures.

P or Q blank are interpreted as P or Q = 0.

SYMB: Input unit:
0 = "Normal" for installation; may be left blank.
1-10 for external tapes (see § 18.7, CONTROLLING AND ALTERNATE INPUT UNITS).
If SYMB of a Type-5 card contains a regional symbol, this start card terminates loading and the program begins at the routine named in SYMB.

LINK: Output mode: of form bbbcd
b = Output unit: blank = unit 1W19; 1-10 means unit 1-10.
c = 0 or blank if assembly listing desired
= 1 or any other character, if assembly listing to be suppressed.
d = 0 or blank if no output desired.
= 1 if output in IPL compressed.
= 2 if output in IPL binary.
= 3 if output in machine code.
= 9 if output in IPL standard.
Each set of IPL compressed or IPL binary output ends with a blank record appropriate to that mode (see § 18.7, CONTROLLING AND ALTERNATE INPUT UNITS).

18.6 TYPE = 9: FIRST CARD

The very first card of each program to be loaded must be a Type-9 card. The use of Type-9 cards allows several programs to be stacked on an external tape for batch execution. SYMB of the Type-9 card specifies the controlling unit for initial loading. If SYMB is blank or 0, the standard input unit is the controlling unit.

18.7 CONTROLLING AND ALTERNATE INPUT UNITS

Generally all sets exist in sequence on a single input unit. However, it is possible to have more complex arrangements. In any case, there will be a single

controlling input unit which contains the header cards of all sets in order. (This unit is specified by SYMB of the first Type-9 card.) If SYMB of a particular header card is blank, then the associated set follows immediately on the controlling unit. If SYMB of the header card refers to an alternate input unit, then the set associated with the header card is read from the alternate unit. The header card on the controlling unit completely specifies the input mode, type of input, destination in storage, output mode and unit; header cards on the alternate unit are ignored. Discrepancy between the header card on the controlling unit and the actual information on the alternate input unit causes a loading error. The set on the alternate unit is terminated by a blank record or by a header card, at which time the next header on the controlling unit is read. Any non-Type-0 cards on the alternate unit are printed like Type-1 cards.

18.8 ASSEMBLY LISTING

It is possible to obtain an assembly listing of the program being loaded when specified by LINK of the Type-5, 6, 7, or 8 header card. This consists of a replica of the cards being input alongside the machine locations they correspond to with the assembled contents in decimal. The assembly listing of Type-0 and Type-1 cards can be suppressed for any set by a signal in the LINK of the header card. Other Type cards are printed under all conditions.

18.9 LOADING DECK

The IPL deck for initial loading consists of the following parts in order:

1. One Type-9 card.
2. All Type-2 cards with exact limits, if any, in any order.

3. All Type-3 cards with exact limits, if any, in any order.
4. All Type-2 cards giving only region size, if any, in any order.
5. All Type-3 cards giving only block size, if any, in any order.

   Only regions and blocks defined by these cards (plus the H, J, W, and $ regions) exist for the IPL computer this run. The Type-2 and 3 cards with exact limits must go first to insure that their cells will be available.

6. Sets of data and routines, in any order.

   Each set is preceded by an appropriate Type-5, 6, 7, or 8 card. For IPL standard and IPL compressed cards, the end of the set is signaled by the next Type-5, 6, 7, or 8 card. For binary and machine modes, a special termination signal is required in the last card (see machine system write-ups for details).

   The input unit named by SYMB of the Type-9 card is the controlling unit for initial loading. If a Type-5, 6, 7, or 8 card on the controlling unit indicates in SYMB that a set is to come from an alternate input unit, then after that set is loaded from the alternate unit, the next header card is picked up from the controlling unit.

7. The start card: A final Type-5 card on the controlling unit with a regional symbol for SYMB to start the program at SYMB.

   Any violation of this order will result in an on-line printed error message.

(It may be noted that the process of loading an IPL program is a one-pass symbolic assembly, hence the need to declare regions at the beginning.)

In loading Type-0 cards, the IPL computer assigns locations from available space to represent local symbols. A list of local symbol definitions is kept. The list is cleared whenever a regional or internal symbol is encountered in NAME (the start of a new list structure).

When internal symbols are treated as pure symbolics rather than as absolute machine locations, they are likewise represented by locations assigned from available space

and thus redefined. A list of internal symbol definitions is kept. This list is cleared upon the appropriate signal from a header card (see Q of Type-5, 6, 7, 8 cards). The programmer knows the correspondence of input symbols and their redefinitions only by means of the assembly listing. Any subsequent output of internal symbols will be in terms of their redefinitions. (Internals 0 through 9 are always defined and absolute, however.)

Regional cells may be defined more than once in the loading sequence. The latest occurring definition is the effective one. (This is often useful in making corrections.)

19.0 IN-PROCESS LOADING

More routines and data can be loaded during interpretation of an IPL program. All options as to mode, unit, etc., available during initial loading are present during in-process loading. No new regions or blocks can be specified during in-process loading. (Not all object machines have full flexibility, so the machine system write-ups should be consulted.)

J165 LOAD ROUTINES AND DATA. More routines and data are read, with the input unit specified by 1W18 as the controlling unit. The load deck consists of header cards (Type-5, 6, 7, or 8), each followed by a set of routines or data (except when the headers specify a set from an alternate input unit), and terminated by a start card (a Type-5 card with a regional SYMB). The routine named as SYMB on the start card is taken as the next routine to be interpreted. If there are no routines or data, or if there is no start card following the sets, then interpretation continues with the instruction following J165.

20.0 SAVE FOR RESTART

A primitive process is provided that allows a running program to be terminated at any point, read out on tape or cards, and restarted again by reading the tape or cards back into the machine. This process may be initiated externally at a monitor point (see § 15.0, MONITOR SYSTEM) or may be put in the program at any point.

J166 SAVE ON UNIT (0) FOR RESTART. The entire contents of main and auxiliary storage are written onto a single external tape (or punched on cards, according to the unit named by data term (0)). Identification of the auxiliary units and external tapes being used by the IPL computer are printed out. Then H5 is set + and the program continues. If the specified auxiliary units and external tapes are provided, and the tape (deck) is loaded under control of a Type-5 card with P = 4 (restart mode), subsequent runs will commence at the instruction following J166, with H5 set - .

Since J166 sets H5+ and the restart process sets H5-, the instruction following J166 can take different action depending on whether this is the original run (H5+) or a restart run (H5-). For example, if the external interrupt cell, W14, named the routine X1, below, and the console signaled an external interrupt, then the run would save for restart and terminate when the next monitor point occurred. Restart runs, since H5 is set - , would restore the original sign of H5 and resume execution at the monitor point.

```
NAME   SIGN   PQ   SYMB   LINK          COMMENTS
X1            40   H5                   Save current sign of H5.
              10   9-1
                   J166                 Save for restart on unit 3,
              70             J7         then terminate this run.
              30   H5        0         Restore H5 on restart runs.
9-1    +      01                   3    Integer 3.
```

J166 does not save external tapes. The programmer saving for restart must provide routines to record the

position of external tapes before executing J166 and to reposition those tapes when continuing after restart. An additional primitive is provided for use in repositioning external tapes:

J167 SKIP LIST STRUCTURE. A single list structure on cards or external tape (as specified by 1W18) in any of the admissible forms--IPL, compressed, binary--(as specified by 1W16) is skipped over, and H5 set + . A blank record is treated as an end-of-list-structure mark. Immediately subsequent blank records are ignored. If there is no list structure (card hopper empty or end-of-file), then H5 is set - . J167 behaves as does J140, except that the structure is not entered into storage.

Save for restart is used to provide a fast-loading version of checked-out routines, to which additional routines to be debugged can be added by J165.

21.0 ERROR TRAP, J170

Many different error conditions can occur during processing by the IPL computer--for example: available space exhausted; specifying other than a data term as operand for an arithmetic process; etc. These conditions cause a system error trap to occur. The action taken upon trapping depends on the routine currently associated with the particular error condition. When an error condition occurs, the following steps take place:

- The safe storage cell W27 is preserved and the CIA at the time of the trap is stored as 1W27. This is the name of the instruction word designating the trapped process, except for primitives executed as links, when it is the name of the primitive.
- The safe storage cell W28 is preserved and the symbol associated with the trapping condition, the trap attribute, is stored as 1W28.
- The description list of W26 (that is, the list 1W26) is searched (as by J10) for the trap attribute. If the trap attribute exists as an attribute of W26, its value names the routine to be executed as the trapping action. That routine is executed. If no value is associated with the trap attribute, the routine associated with the attribute 'internal zero' (the symbol 0) is executed as the trapping action. If no value is associated with 'internal zero', no trapping action is taken.

  The trapping action is executed as a subprocess of the trapped process--that is, as though it were designated directly in the trapped process. Because H0, H5, and the W's are not disturbed by the error trap mechanism, the trapping action can repeat the trapped process under its own control, if desired. If the trapping action is marked with Q = 4, it will trace conditionally.
- When the trapping action terminates, W27 and W28 are restored and interpretation continues with the process following the trapped process.

The machine system write-ups should be consulted for the normal error condition and trap actions. However, the traps described below are standard for all machine systems.

TRAP ON AVAILABLE SPACE EXHAUSTED: 'H2'.

W32 holds the name of an integer data term which specifies the number of cells to be removed from H2 before execution of the program begins. If available space becomes exhausted during execution, these cells are returned to H2 to enable trapping on the attribute H2. The trap action routine to regain space must be provided by the programmer, and may include erasing data structures or routines (J72, J201), or filing data on auxiliary space (J106 or J107). Upon return from the trap, 1W32 cells are again removed from H2 and the program continues. 1W32 initially specifies ten cells, but may be changed by the programmer at any time. The change becomes effective at the next monitor point, after the next H2 trap has been executed, or whenever a start card is encountered by the loader.

TRAP ON INTERPRETATION CYCLE COUNT: 'H3'.

Traps when H3 (cycle count) is equal to W33. W33 is an integer data term that is compared to H3 each interpretation cycle, after H3 has been incremented. When H3 is equal to W33, the action associated with the symbol H3 on 1W26 is executed and the program continues. W33 is initially zero, so no trapping will occur until the programmer sets W33 to a non-zero value.

The standard description list form of W26 allows any trapping action to be modified or disabled by assigning a different value to the trap attribute. Also, additional trap attributes and associated actions can be added. A primitive process is provided to take trapping action at any point in the program.

J170 TRAP ON (0). J170 preserves W27 and W28, stores the appropriate CIA in W27 and (0) in W28, searches the description list of W26 for the attribute (0), and executes as a subprocess of the process designating J170 the routine named by the associated value. If (0) is not an attribute of W26, the routine associated with 'internal zero' is executed. If 'internal zero' is not an attribute of W26, no trapping action is taken. J170 then restores W27 and W28 and terminates.

22.0 LINE READ PROCESSES, J180 to J189

The line read primitives provide a means of reading a BCD card under control of an IPL-V program and translating selected fields into IPL symbols or data terms.

Control Cells:

- 1W18 names the input unit for J180. 1W18 = 0 means the normal input tape.
- 1W24 names the current read line. ("Read lines" and "print lines" are identical and interchangeable. Lines for either or both purposes are specified by Type-3 cards with Q = 1.)
- 1W25 is a decimal integer data term specifying the left column of the current input field.
- 1W30 is a positive decimal integer data term specifying the size (number of columns) of the current input field.

J180 READ LINE. The next record on unit 1W18 is read to line 1W24. (The record is assumed to be BCD, 80 cols.) Column 1 of the record is read into column 1 of the read line, and so forth. H5 is set + . If no record can be read (end-of-file condition), the line is not changed and H5 is set - .

J181 INPUT LINE SYMBOL. The IPL symbol in the field starting in column 1W25, of size 1W30, in line 1W24, is input to H0 and H5 is set + . The symbol is regional if the first (leftmost) column holds a regional character; otherwise, it is absolute internal. All non-numerical characters except in the first column are ignored. If the field is entirely blank, or ignored, there is no input to H0, and H5 is set - . In either case, 1W25 is incremented by the amount 1W30. (J181 turns unused regional symbols into empty but used symbols.)

J182 INPUT LINE DATA TERM (0). The field specified as in J181 is taken as the value of a data term. Input data term (0) is set to that value and left as output (0). H5 is set + . The data type of input (0) determines the data type of the output. If the input (0) is a decimal or octal integer, or BCD, the read line field is interpreted as that type. Any other data type is treated as BCD. In composing BCD data terms,

the field is left-justified and the full data term completed with blanks on the right, if necessary. If the specified field exceeds five columns, the rightmost five columns are taken as the field. In composing decimal and octal integer data terms, non-numerical characters are ignored. If the resulting information exceeds the capacity of the data term, the rightmost digits are retained. If the read line field is entirely blank (or non-numerical, for integer data types), (0) is cleared (to blanks for BCD; to zero for integer) and H5 is set - . In either case, 1W25 is incremented by the amount 1W30.

J183 SET (0) TO NEXT BLANK. (0) is taken as a decimal integer data term. Line 1W24 is scanned, left to right, starting with column 1W25+1, for a blank. One is added to (0) for each column scanned, including that in which the scanned-for character ('blank' in J183) is found. (0) is left as output (0). H5 is set + if the character is found in the line, and - if it is not. (Thus, if input (0) = 1W25, after scanning, output (0) will specify the column holding the scanned-for character. If input (0) = decimal integer 0, after scanning, output (0) will be the size of a field beginning in column 1W25 and delimited on the right by the next occurrence of the scanned-for character.)

J184 SET (0) TO NEXT NON-BLANK. Same as J183, except scans for any non-blank character.

J185 SET (1) TO NEXT OCCURRENCE OF CHARACTER (0). Same as J183, except scans for character (0), counting into decimal integer data term (1). Input (1) is left as output (0). If input (0) is a regional symbol, its region character is the character scanned for, if input (0) is internal, its last (low-order) digit is the character scanned for.

J186 INPUT LINE CHARACTER. The character in column 1W25 of line 1W24 is input to H0, H5 is set + . If the character is numerical, that internal symbol is input; if the character is non-numerical, the zeroth symbol in the region designated by that character is input; i.e., A → A0, 3 → 3. If the character is a blank, there is no input and H5 is set - . In either case, 1W25 is not advanced.

J189 TRANSFER FIELD. The field in line 1W24, starting in column 1W25, and of size 1W30, is transferred to line (0), starting in column 1W21. H5 is set + . If the entire field cannot be transferred (line (0) is too short), as much is transferred as can be, and H5 is set - . In either case, 1W25 is set to the last column transferred plus one.

23.0 PARTIAL WORD PROCESSES, J190 to J197

These primitives allow manipulation and testing of the P, Q, SYMB, or LINK of IPL words. The words are assumed to be standard words, not data terms. The P, Q, SYMB, or LINK is input to, or output from, the symbol portion of HO, and may be treated as any other IPL symbol.

J190 INPUT P OF CELL (0) TO HO. After J190, the symbol in HO will be an absolute internal symbol between zero and seven.

J191 INPUT Q OF CELL (0) TO HO. After J191, the symbol in HO will be an absolute internal symbol between zero and seven.

J192 INPUT SYMB OF CELL (0) TO HO. The symbol input will be regional if covered by a region control word; otherwise, it will be internal. That is, the Q of the cell (0) is not used to determine the type of symbol.

J193 INPUT LINK OF CELL (0) TO HO. The symbol input will be regional if covered by a region control word; otherwise, it will be internal.

J194 SET (1) TO BE THE P OF CELL (0). (1) is an absolute internal symbol between zero and seven.

J195 SET (1) TO BE THE Q OF CELL (0). (1) is an absolute internal symbol between zero and seven.

J196 SET (1) TO BE THE SYMB OF CELL (0). Q of cell (0) is unchanged.

J197 SET (1) TO BE THE LINK OF CELL (0).

24.0 MISCELLANEOUS PROCESSES, J200 to J209

J200 LOCATE THE (0)th SYMBOL ON LIST (1). (0) is an integer data term whose sign is ignored, and whose value, n, specifies that the name of the nth list cell of list (1) be output in H0, with H5 set + . Output (0) names the last cell if H5 is set - , indicating that less than n symbols exist on list (1). (Note that private termination cells are not list cells.)

J201 ERASE ROUTINE (0). Return the space to the available space list, 1W34. (0) is assumed to be a regional cell and is set empty rather than being returned to available space. If (0) contains Q = 6 or 7, it is assumed to be an auxiliary routine and J201 does nothing.

For J201 all non-regional symbols appearing in the SYMB of a routine are treated as sublists to be erased. Thus, mentioning local or internal data terms, working cells, or data lists in the routine will cause unpredictable erasure. A regional LINK is equivalent to LINK = 0, signaling the end of the sublist. If a routine is loaded after J171 has been executed, an unused regional cell from the middle of a regional block may be used in its construction. Since J201 considers this cell to be regional and hence the termination of a sublist, a portion of the routine may not be returned to available space.

J202 PRINT POST MORTEM AND CONTINUE. (See § 15.4, POST MORTEM, for complete definition of J202.)

25.0 CHANGES AND EXTENSIONS

The modifications described in this section have originated from users' experience with IPL-V in the two years since publication of the first edition of the Manual. Sections 25.1 through 25.3 describe changes to previously defined features of the system; they are reported separately because in some cases they may impose minor modifications to previously checked out programs. The extensions of IPL-V are described in Sections 25.4 through 25.8; they impose no modifications to existing programs. The modifications are not described in full in this section, but are simply listed with references to the appropriate sections of the Manual.

25.1 SYSTEM CELL CHANGES (See § 4.2)

W14 External interrupt cell; holds name of routine executed at return to Q = 3 point. (See § 15.3, EXTERNAL INTERRUPT.)

W15 Post mortem routine cell; holds name of routine executed after the post mortem lists have been printed. (See § 15.4, POST MORTEM.)

25.2 PRIMITIVE PROCESS CHANGES

J166 SAVE ON UNIT (0) FOR RESTART. The program does not terminate when J166 is executed. J166 sets H5+, and restarting causes H5 to be set - . (See § 20.0, SAVE FOR RESTART.)

25.3 CHANGES IN LOADING CONVENTIONS

TYPE = 3: BLOCK RESERVATION CARDS (See § 18.3)

NAME is the regional symbol naming the line for Q = 1. (The earlier edition of the Manual erroneously stated that SYMB specified the name.)

TYPE = 6 or 7: HEADER CARDS (See § 10.2)

When a single Type-6 or Type-7 header precedes several routines, the entire set of routines is loaded into consecutive cells of the buffer and written to auxiliary as a single unit when the next header is encountered. A set of routines too large for the buffer overflows into main memory, using cells from H2. The entire set of routines is brought into main memory when any one of them is executed. Mutual calls between routines in the same set do not result in accesses to auxiliary.

TYPE = 9: FIRST CARD (See §§ 18.6, 18.7)

SYMB of the first Type-9 card specifies the controlling unit; comments on Type-9 cards are restricted to the COMMENTS field of the coding form.

J171 RETURN UNUSED REGIONAL CELLS TO H2. (See § 17.0, BLOCK HANDLING PROCESSES.)

Unused regional cells are not automatically returned to available space at the end of initial loading; they are returned only when J171 is executed.

25.4 EXTENSIONS TO LIST OF SYSTEM CELLS

The cells W30 through W43 have been assigned system functions as described in § 4.2.

25.5 EXTENSIONS TO THE LIST OF BASIC PROCESSES

The following processes have been added; their full descriptions are found in the indicated sections:

LIST PROCESSES (§ 9.8)
*J103 Gen cells of block (1) for (0).

AUXILIARY STORAGE PROCESSES (§ 10.1)
J109 Compact auxiliary data storage system (0).

PRINT PROCESSES (§ 16.2)
*J162 Enter (0) according to format W43.

```
BLOCK HANDLING PROCESSES (§ 17.0)
     J171  Return unused regionals to H2.
     J172  Make block (0) into a list.
    *J173  Read into block (0).
    *J174  Write block (0).
    *J175  FIND region control word of regional symbol (0).
     J176  Space (0) blocks on unit 1W19.

LINE READ PROCESSES (§ 22.0)
    *J180  Read line.
    *J181  Input line symbol.
    *J182  Input line data term (0).
    *J183  Set (0) to next blank, leave (0).
    *J184  Set (0) to next non-blank, leave (0).
    *J185  Set (1) to next occurrence of character (0),
           leave (0).
    *J186  Input line character.
    *J189  Transfer field to line (0).

PARTIAL WORD PROCESSES (§ 23.0)
     J190  Input P of cell (0).
     J191  Input Q of cell (0).
     J192  Input SYMB of cell (0).
     J193  Input LINK of cell (0).
     J194  Set (1) to be P of cell (0).
     J195  Set (1) to be Q of cell (0).
     J196  Set (1) to be SYMB of cell (0).
     J197  Set (1) to be LINK of cell (0).

MISCELLANEOUS PROCESSES (§ 24.0)
    *J200  LOCATE (0)th symbol on list (1).
     J201  ERASE routine (0).
     J202  Print post mortem and continue.
```

25.6 EXTENSIONS TO THE LOADER

TYPE = 2: REGION CARDS (See §§ 18.2, 17.0)

A block control word for a region is created by a Type-2 card, and this region control word is accessible by J175. The loader defines the first symbol of those regions the programmer did not define. The $ region is reserved for system routines and data unique to local installations.

TYPE = 3: BLOCK RESERVATION CARDS (See §§ 18.3, 17.0)

The loader creates a block control word in the cell appearing in NAME of all Type-3 cards.

TYPE = 5, 6, 7, 8: HEADER CARDS (See §§ 18.5, 19.0 14.0, 17.0)

The loading processes load into the available space list 1W34. Sets of data or routines going into main storage may be loaded into cells from the standard available space list H2 (by NAME = blank, 1W34 = H2) or into specific blocks of cells (by NAME = name of block).

A Type-8 editing header inhibits loading of its associated set of routines or data but allows output and listing options; it is intended for skipping sets on an alternate input unit.

INPUT MODE (See §§ 18.5, 20.0, 13.3)

Header cards with P = 3 indicate machine code; headers with P = 4 indicate restart mode.

OUTPUT MODE (See §§ 18.5, 13.3)

The integer 3 indicates machine code output; the integer 9 indicates output in IPL standard form.

25.7 EXTENSIONS TO THE MONITOR SYSTEM

The three externally imposed trace conditions may also be imposed internally by setting the data term 1W31 appropriately. (See § 15.5, TRACING.)

A post mortem may be printed at any point in the processing by J202, without terminating the program. A terminal post mortem is still given automatically. (See § 15.4, POST MORTEM.)

25.8 EXTENSIONS TO THE INTERPRETIVE SYSTEM

The interpretation cycle count in H3 is compared each cycle to the number set by the programmer in cell W33. Trapping on the attribute H3 occurs on equality. (See § 21.0, ERROR TRAP.)

When available space is exhausted, a number of cells of reserved space is added to H2 and trapping on the attribute H2 occurs. (See § 21.0, ERROR TRAP.)

LIST OF IPL-V BASIC PROCESSES

* Indicates processes which set H5

General Processes (§ 5.0)
- J0 No operation
- J1 Execute (0) after restoring H0
- *J2 TEST (0) ≠ (1)
- *J3 Set H5-
- *J4 Set H5+
- *J5 Reverse sense of H5
- J6 Reverse (0) and (1)
- J7 Halt, proceed on GO
- J8 Restore H0
- J9 ERASE cell (0)

Description Processes (§ 6.0)
- *J10 FIND value of attribute (0) of (1)
- J11 Assign (1) as value of attribute (0) of (2)
- J12 Add (1) at front of value list of attribute (0) of (2)
- J13 Add (1) at end of value list of attribute (0) of (2)
- J14 ERASE attribute (0) of (1)
- J15 ERASE all attributes of (0)
- *J16 FIND attribute of (0) randomly

Generator Housekeeping Processes (§ 7.1)
- J17 Gen set up: context (0), subprocess (1)
- *J18 Execute subprocess of Gen
- *J19 Gen clean up

Working Storage Processes (§ 8.0)
- J2n MOVE (0)-(n) into W0-Wn
- J3n Restore W0-Wn
- J4n Preserve W0-Wn
- J5n Preserve W0-Wn; MOVE (0)-(n) into W0-Wn

List Processes (§ 9.8)
- *J60 LOCATE next symbol after cell (0)
- *J61 LOCATE last symbol on list (0)
- *J62 LOCATE (0) on list (1) (1st occurrence)
- J63 INSERT (0) before symbol in cell (1)
- J64 INSERT (0) after symbol in cell (1)
- J65 INSERT (0) at end of list (1)
- J66 INSERT (0) at end if not on list (1)
- J67 Replace (1) by (0) on list (2) (1st occur.)
- *J68 DELETE symbol in cell (0)
- *J69 DELETE (0) from list (1) (1st occurrence)
- *J70 DELETE last symbol from list (0)
- J71 ERASE list (0)
- J72 ERASE list structure (0)
- J73 COPY list (0)
- J74 COPY list structure (0)
- J75 Divide list after location (0); name of remainder is output (0)
- *J76 INSERT list (0) after (1), locate last symbol
- *J77 TEST if (0) is on list (1)
- *J78 TEST if list (0) is not empty
- *J79 TEST if cell (0) is not empty
- *J8n FIND the nth symbol on list (0)
- J9n Create list of n symbols, (n-1) to (0)
- *J100 Gen symbols on list (1) for (0)
- *J101 Gen cells of list structure (1) for (0)
- *J102 Gen cells of tree (1) for (0)
- *J103 Gen cells of block (1) for (0)
- J104

Auxiliary Storage Processes (§ 10.1)
- *J105 MOVE list structure (0) in from auxiliary
- J106 File list structure (0) in fast-auxiliary
- J107 File list structure (0) in slow-auxiliary
- *J108 TEST if list structure (0) is on auxiliary
- J109 Compact auxiliary data storage system (0)

Arithmetic Processes (§ 11.0)
- J110 (1) + (2) → (0), leave (0)
- J111 (1) - (2) → (0), leave (0)
- J112 (1) x (2) → (0), leave (0)
- J113 (1) / (2) → (0), leave (0)
- *J114 TEST if (0) = (1)
- *J115 TEST if (0) > (1)
- *J116 TEST if (0) < (1)
- *J117 TEST if (0) = 0
- *J118 TEST if (0) > 0
- *J119 TEST if (0) < 0
- J120 COPY (0)
- J121 Set (0) identical to (1), leave (0)
- J122 Take absolute value of (0), leave (0)
- J123 Take negative of (0), leave (0)
- J124 Clear (0), leave (0)
- J125 Tally 1 in (0), leave (0)
- J126 Count list (0)
- *J127 TEST if data type (0) = data type (1)
- J128 Translate (0) to be data type of (1), leave (0)
- J129 Produce random number between 0 and (0)

Data Prefix Processes (§ 12.2)
- *J130 TEST if (0) is regional symbol
- *J131 TEST if (0) names data term
- *J132 TEST if (0) is local symbol
- *J133 TEST if list (0) has been marked processed
- *J134 TEST if (0) is internal symbol
- J135
- J136 Make (0) local, leave (0)
- J137 Mark list (0) processed, leave (0)
- J138 Make (0) internal, leave (0)
- J139

Read and Write Processes (§ 14.0)
- *J140 Read list structure
- *J141 Read symbol from console
- J142 Write list structure (0)
- J143 Rewind tape (0)
- J144 Skip to next tape file
- J145 Write end-of-file
- J146 Write end-of-set

Monitor System (§ 15.6)
- J147 Mark routine (0) to trace
- J148 Mark routine (0) to propagate trace
- J149 Mark routine (0) to not trace

Print Processes (§ 16.1, 16.2)
- J150 Print list structure (0)
- J151 Print list (0)
- J152 Print symbol (0)
- J153 Print data term (0) w/o name or type
- J154 Clear print line
- J155 Print line
- *J156 Enter symbol (0) left-justified
- *J157 Enter data term (0) left-justified
- *J158 Enter symbol (0) right-justified
- *J159 Enter data term (0) right-justified
- J160 Tab to column (0)
- J161 Increment column by (0)
- *J162 Enter (0) according to format W43
- J163
- J164

In-process Loading (§ 19.0)
- J165 Load routines and data

Save for Restart (§ 20.0)
- *J166 Save on unit (0) for restart
- *J167 Skip list structure
- J168
- J169

Error Trap (§ 21.0)
- J170 Trap on (0)

Block Handling Processes (§ 17.0)
- J171 Return unused regionals to H2
- J172 Make block (0) into a list
- *J173 Read into block (0)
- *J174 Write block (0)
- *J175 FIND region control word of regional symbol (0)
- J176 Space (0) blocks on unit 1W19
- J177
- J178
- J179

Line Read Processes (§ 22.0)
- *J180 Read line
- *J181 Input line symbol
- *J182 Input line data term (0)
- *J183 Set (0) to next blank, leave (0)
- *J184 Set (0) to next non-blank, leave (0)
- *J185 Set (1) to next occurrence of character (0), leave (0)
- *J186 Input line character
- J187
- J188
- *J189 Transfer field to line (0)

Partial Word Processes (§ 23.0)
- J190 Input P of cell (0)
- J191 Input Q of cell (0)
- J192 Input SYMB of cell (0)
- J193 Input LINK of cell (0)
- J194 Set (1) to be P of cell (0)
- J195 Set (1) to be Q of cell (0)
- J196 Set (1) to be SYMB of cell (0)
- J197 Set (1) to be LINK of cell (0)
- J198
- J199

Miscellaneous Processes (§ 24.0)
- *J200 LOCATE (0)th symbol on list (1)
- J201 ERASE routine (0)
- J202 Print post mortem and continue

IPL INSTRUCTION: PQ SYMB LINK

P is operation code
- P = 0 Execute S
- P = 1 Input S (after preserving H0)
- P = 2 Output to S (then restore H0)
- P = 3 Restore (pop up) S
- P = 4 Preserve (push down) S
- P = 5 Replace (0) by S
- P = 6 Copy (0) in S
- P = 7 Branch to S if H5-

Q is designation code
- Q = 0 S = SYMB
- Q = 1 S = symbol in cell named SYMB
- Q = 2 S = symbol in cell named in cell named SYMB
- Q = 3 S = SYMB; start selective trace
- Q = 4 S = SYMB; continue selective trace
- Q = 5 Machine language routine
- Q = 6 Routine in fast-aux. storage
- Q = 7 Routine in slow-aux. storage

SYMB is symbol operated on by Q

LINK is address of next instruction (0 for end of routine)

SYSTEM STORAGE CELLS

H0 Communication cell
H1 Current instruction address cell
H2 Available space list
H3 Tally of interpretation cycles
H4 Current auxiliary routine cell
H5 Test cell

W0-W9 Common working storage
W10 Random number control cell
W11 Integer division remainder
W12 Monitor start cell (Q = 3)
W13 Monitor end cell (Q = 3)
W14 External interrupt cell
W15 Post mortem routine cell
W16 Input mode cell
W17 Output mode cell
W18 Read unit cell
W19 Write unit cell
W20 Print unit cell
W21 Print column cell
W22 Print spacing cell
W23 Post mortem list cell
W24 Print line cell
W25 Print entry column cell
W26 Error trap cell
W27 Trap address cell
W28 Trap symbol cell
W29 Monitor point address cell
W30 Field length cell
W31 Trace mode cell
W32 Reserved available space cell
W33 Cycle count for trap cell
W34 Current available space cell
W35 Slow-aux. obsolete structure cell
W36 Used slow-auxiliary space cell
W37 Slow-auxiliary storage density cell
W38 Slow-auxiliary storage compacting routine cell
W39 Fast-aux. obsolete structure cell
W40 Used fast-auxiliary space cell
W41 Fast-auxiliary storage density cell
W42 Fast-auxiliary storage compacting routine cell
W43 Format cell

IPL DATA: PQ SYMB LINK

Q = 0 Standard list cell:
- P is irrelevant
- SYMB is symbol
- LINK is address of next list cell (0 for end of list)

Q = 1 Data term:

	±PQ	SYMB	LINK
Decimal integer	1	dddd	dddd
Floating point	11	ddddd	d ±ee
Alphanumeric	21	aaaaa	
Octal	31	ddddd	ddddd

TYPE CARDS

0 (blank) Routines and data
1 Comments
2 Region definition
- NAME = Regional symbol
- SYMB = Origin (if given)
- LINK = Size

3 Block reservation
- NAME = Block control word (if given)
- SYMB = Origin (if given)
- LINK = Size
- Q = 0 Reserve regional symbols
- Q = 1 Reserve print line
- Q = 2 Reserve block
- Q = 3 Reserve auxiliary buffer
- Q = 4 Specify available space

4 Listing cards
5 Main storage header
6 Fast-auxiliary storage header
7 Slow-auxiliary storage header
8 Editing header; inhibits loading
- NAME = Name of storage block
- P = Input mode
  - P = 0 IPL standard
  - P = 1 IPL compressed
  - P = 2 IPL binary
  - P = 3 Machine code
  - P = 4 Restart mode
- Q = Type of input
  - Q = 0 Routines; internals symbolic
  - Q = 1 Data; internals symbolic
  - Q = 2 Routines; internals symbolic; reset internal symbol table
  - Q = 3 Data; internals symbolic; reset internal symbol table
  - Q = 4 Routines; internals absolute
  - Q = 5 Data; internals absolute
- SYMB = Alternate input unit
  - 0 (blank) = controlling unit
  - 1-10 = Internal tapes
  - Regional SYMB names first routine (terminate loading)
- LINK = Output mode: of form bbbcd
  - b = Output unit: blank = unit 1W19; 1-10 = unit 1-10
  - c = 0 (blank) if assembly listing
    - = 1 or any character if no assembly listing
  - d = 0 (blank) if no output
    - = 1 IPL compressed output
    - = 2 IPL binary output
    - = 3 Machine code output
    - = 9 IPL standard output

9 First card
- SYMB = Controlling unit (0 or blank = normal input unit)

INDEX

(Index references are to the section numbers of Part Two, except where noted by the letter "p", in which case the reference is to a particular page. The Index is for Part Two only.)

LIST OF IPL-V BASIC PROCESSES

* Indicates processes which set H5

General Processes ($ 5.0)
- J0 No operation
- J1 Execute (0) after restoring H0
- *J2 TEST (0) = (1)
- *J3 Set H5-
- *J4 Set H5+
- *J5 Reverse sense of H5
- J6 Reverse (0) and (1)
- J7 Halt, proceed on GO
- J8 Restore H0
- J9 ERASE cell (0)

Description Processes ($ 6.0)
- *J10 FIND value of attribute (0) of (1)
- J11 Assign (1) as value of attribute (0) of (2)
- J12 Add (1) at front of value list of attribute (0) of (2)
- J13 Add (1) at end of value list of attribute (0) of (2)
- J14 ERASE attribute (0) of (1)
- J15 ERASE all attributes of (0)
- *J16 FIND attribute of (0) randomly

Generator Housekeeping Processes ($ 7.1)
- J17 Gen set up: context (0), subprocess (1)
- *J18 Execute subprocess of Gen
- *J19 Gen clean up

Working Storage Processes ($ 8.0)
- J2n MOVE (0)-(n) into W0-Wn
- J3n Restore W0-Wn
- J4n Preserve W0-Wn
- J5n Preserve W0-Wn; MOVE (0)-(n) into W0-Wn

List Processes ($ 9.8)
- *J60 LOCATE next symbol after cell (0)
- *J61 LOCATE last symbol on list (0)
- *J62 LOCATE (0) on list (1) (1st occurrence)
- J63 INSERT (0) before symbol in cell (1)
- J64 INSERT (0) after symbol in cell (1)
- J65 INSERT (0) at end of list (1)
- J66 INSERT (0) at end if not on list (1)
- J67 Replace (1) by (0) on list (2) (1st occur.)
- *J68 DELETE symbol in cell (0)
- *J69 DELETE (0) from list (1) (1st occurrence)
- *J70 DELETE last symbol from list (0)
- J71 ERASE list (0)
- J72 ERASE list structure (0)
- J73 COPY list (0)
- J74 COPY list structure (0)
- J75 Divide list after location (0); name of remainder is output (0)
- *J76 INSERT list (0) after (1), locate last symbol
- *J77 TEST if (0) is on list (1)
- *J78 TEST if list (0) is not empty
- *J79 TEST if cell (0) is not empty
- *J8n FIND the nth symbol on list (0)
- J9n Create list of n symbols, (n-1) to (0)
- *J100 Gen symbols on list (1) for (0)
- *J101 Gen cells of list structure (1) for (0)
- *J102 Gen cells of tree (1) for (0)
- *J103 Gen cells of block (1) for (0)
- J104

Auxiliary Storage Processes ($ 10.1)
- *J105 MOVE list structure (0) in from auxiliary
- J106 File list structure (0) in fast-auxiliary
- J107 File list structure (0) in slow-auxiliary
- *J108 TEST if list structure (0) is on auxiliary
- J109 Compact auxiliary data storage system (0)

Arithmetic Processes ($ 11.0)
- J110 (1) + (2) → (0), leave (0)
- J111 (1) - (2) → (0), leave (0)
- J112 (1) x (2) → (0), leave (0)
- J113 (1) / (2) → (0), leave (0)
- *J114 TEST if (0) = (1)
- *J115 TEST if (0) > (1)
- *J116 TEST if (0) < (1)
- *J117 TEST if (0) = 0
- *J118 TEST if (0) > 0
- *J119 TEST if (0) < 0
- J120 COPY (0)
- J121 Set (0) identical to (1), leave (0)
- J122 Take absolute value of (0), leave (0)
- J123 Take negative of (0), leave (0)
- J124 Clear (0), leave (0)
- J125 Tally 1 in (0), leave (0)
- J126 Count list (0)
- *J127 TEST if data type (0) = data type (1)
- J128 Translate (0) to be data type of (1), leave (0)
- J129 Produce random number between 0 and (0)

Data Prefix Processes ($ 12.2)
- *J130 TEST if (0) is regional symbol
- *J131 TEST if (0) names data term
- *J132 TEST if (0) is local symbol
- *J133 TEST if list (0) has been marked processed
- *J134 TEST if (0) is internal symbol
- J135
- J136 Make (0) local, leave (0)
- J137 Mark list (0) processed, leave (0)
- J138 Make (0) internal, leave (0)
- J139

Read and Write Processes ($ 14.0)
- *J140 Read list structure
- *J141 Read symbol from console
- J142 Write list structure (0)
- J143 Rewind tape (0)
- J144 Skip to next tape file
- J145 Write end-of-file
- J146 Write end-of-set

Monitor System ($ 15.6)
- J147 Mark routine (0) to trace
- J148 Mark routine (0) to propagate trace
- J149 Mark routine (0) to not trace

Print Processes ($ 16.1, 16.2)
- J150 Print list structure (0)
- J151 Print list (0)
- J152 Print symbol (0)
- J153 Print data term (0) w/o name or type
- J154 Clear print line
- J155 Print line
- *J156 Enter symbol (0) left-justified
- *J157 Enter data term (0) left-justified
- *J158 Enter symbol (0) right-justified
- *J159 Enter data term (0) right-justified
- J160 Tab to column (0)
- J161 Increment column by (0)
- *J162 Enter (0) according to format W43
- J163
- J164

In-process Loading ($ 19.0)
- J165 Load routines and data

Save for Restart ($ 20.0)
- *J166 Save on unit (0) for restart
- *J167 Skip list structure
- J168
- J169

Error Trap ($ 21.0)
- J170 Trap on (0)

Block Handling Processes ($ 17.0)
- J171 Return unused regionals to H2
- J172 Make block (0) into a list
- *J173 Read into block (0)
- *J174 Write block (0)
- *J175 FIND region control word of regional symbol (0)
- J176 Space (0) blocks on unit 1W19
- J177
- J178
- J179

Line Read Processes ($ 22.0)
- *J180 Read line
- *J181 Input line symbol
- *J182 Input line data term (0)
- *J183 Set (0) to next blank, leave (0)
- *J184 Set (0) to next non-blank, leave (0)
- *J185 Set (1) to next occurrence of character (0), leave (0)
- *J186 Input line character
- J187
- J188
- *J189 Transfer field to line (0)

Partial Word Processes ($ 23.0)
- J190 Input P of cell (0)
- J191 Input Q of cell (0)
- J192 Input SYMB of cell (0)
- J193 Input LINK of cell (0)
- J194 Set (1) to be P of cell (0)
- J195 Set (1) to be Q of cell (0)
- J196 Set (1) to be SYMB of cell (0)
- J197 Set (1) to be LINK of cell (0)
- J198
- J199

Miscellaneous Processes ($ 24.0)
- *J200 LOCATE (0)th symbol on list (1)
- J201 ERASE routine (0)
- J202 Print post mortem and continue

IPL INSTRUCTION: PQ SYMB LINK

P is operation code
- P = 0 Execute S
- P = 1 Input S (after preserving H0)
- P = 2 Output to S (then restore H0)
- P = 3 Restore (pop up) S
- P = 4 Preserve (push down) S
- P = 5 Replace (0) by S
- P = 6 Copy (0) in S
- P = 7 Branch to S if H5-

Q is designation code
- Q = 0 S = SYMB
- Q = 1 S = symbol in cell named SYMB
- Q = 2 S = symbol in cell named in cell named SYMB
- Q = 3 S = SYMB; start selective trace
- Q = 4 S = SYMB; continue selective trace
- Q = 5 Machine language routine
- Q = 6 Routine in fast-aux. storage
- Q = 7 Routine in slow-aux. storage

SYMB is symbol operated on by Q

LINK is address of next instruction (0 for end of routine)

SYSTEM STORAGE CELLS

H0 Communication cell
H1 Current instruction address cell
H2 Available space list
H3 Tally of interpretation cycles
H4 Current auxiliary routine cell
H5 Test cell

W0-W9 Common working storage
W10 Random number control cell
W11 Integer division remainder
W12 Monitor start cell (Q = 3)
W13 Monitor end cell (Q = 3)
W14 External interrupt cell
W15 Post mortem routine cell
W16 Input mode cell
W17 Output mode cell
W18 Read unit cell
W19 Write unit cell
W20 Print unit cell
W21 Print column cell
W22 Print spacing cell
W23 Post mortem list cell
W24 Print line cell
W25 Print entry column cell
W26 Error trap cell
W27 Trap address cell
W28 Trap symbol cell
W29 Monitor point address cell
W30 Field length cell
W31 Trace mode cell
W32 Reserved available space cell
W33 Cycle count for trap cell
W34 Current available space cell
W35 Slow-aux. obsolete structure cell
W36 Used slow-auxiliary space cell
W37 Slow-auxiliary storage density cell
W38 Slow-auxiliary storage compacting routine cell
W39 Fast-aux. obsolete structure cell
W40 Used fast-auxiliary space cell
W41 Fast-auxiliary storage density cell
W42 Fast-auxiliary storage compacting routine cell
W43 Format cell

IPL DATA: PQ SYMB LINK

Q = 0 Standard list cell:
- P is irrelevant
- SYMB is symbol
- LINK is address of next list cell (0 for end of list)

Q = 1 Data term:

	±PQ	SYMB	LINK
Decimal integer	1	dddd	dddd
Floating point	11	ddddd	d ±ee
Alphanumeric	21	aaaaa	
Octal	31	ddddd	ddddd

TYPE CARDS

0 (blank) Routines and data
1 Comments
2 Region definition
- NAME = Regional symbol
- SYMB = Origin (if given)
- LINK = Size

3 Block reservation
- NAME = Block control word (if given)
- SYMB = Origin (if given)
- LINK = Size
- Q = 0 Reserve regional symbols
- Q = 1 Reserve print line
- Q = 2 Reserve block
- Q = 3 Reserve auxiliary buffer
- Q = 4 Specify available space

4 Listing cards
5 Main storage header
6 Fast-auxiliary storage header
7 Slow-auxiliary storage header
8 Editing header; inhibits loading
- NAME = Name of storage block
- P = Input mode
  - P = 0 IPL standard
  - P = 1 IPL compressed
  - P = 2 IPL binary
  - P = 3 Machine code
  - P = 4 Restart mode
- Q = Type of input
  - Q = 0 Routines; internals symbolic
  - Q = 1 Data; internals symbolic
  - Q = 2 Routines; internals symbolic; reset internal symbol table
  - Q = 3 Data; internals symbolic; reset internal symbol table
  - Q = 4 Routines; internals absolute
  - Q = 5 Data; internals absolute
- SYMB = Alternate input unit
  - 0 (blank) = controlling unit
  - 1-10 = Internal tapes
  - Regional SYMB names first routine (terminate loading)
- LINK = Output mode: of form bbbcd
  - b = Output unit: blank = unit 1W19; 1-10 = unit 1-10
  - c = 0 (blank) if assembly listing
    - = 1 or any character if no assembly listing
  - d = 0 (blank) if no output
    - = 1 IPL compressed output
    - = 2 IPL binary output
    - = 3 Machine code output
    - = 9 IPL standard output

9 First card
- SYMB = Controlling unit (0 or blank = normal input unit)

LIST OF IPL-V BASIC PROCESSES

* Indicates processes which set H5

General Processes (§ 5.0)
- J0 No operation
- J1 Execute (0) after restoring H0
- *J2 TEST (0) = (1)
- *J3 Set H5-
- *J4 Set H5+
- *J5 Reverse sense of H5
- J6 Reverse (0) and (1)
- J7 Halt, proceed on GO
- J8 Restore H0
- J9 ERASE cell (0)

Description Processes (§ 6.0)
- *J10 FIND value of attribute (0) of (1)
- J11 Assign (1) as value of attribute (0) of (2)
- J12 Add (1) at front of value list of attribute (0) of (2)
- J13 Add (1) at end of value list of attribute (0) of (2)
- J14 ERASE attribute (0) of (1)
- J15 ERASE all attributes of (0)
- *J16 FIND attribute of (0) randomly

Generator Housekeeping Processes (§ 7.1)
- J17 Gen set up: context (0), subprocess (1)
- *J18 Execute subprocess of Gen
- *J19 Gen clean up

Working Storage Processes (§ 8.0)
- J2n MOVE (0)-(n) into W0-Wn
- J3n Restore W0-Wn
- J4n Preserve W0-Wn
- J5n Preserve W0-Wn; MOVE (0)-(n) into W0-Wn

List Processes (§ 9.8)
- *J60 LOCATE next symbol after cell (0)
- *J61 LOCATE last symbol on list (0)
- *J62 LOCATE (0) on list (1) (1st occurrence)
- J63 INSERT (0) before symbol in cell (1)
- J64 INSERT (0) after symbol in cell (1)
- J65 INSERT (0) at end of list (1)
- J66 INSERT (0) at end if not on list (1)
- J67 Replace (1) by (0) on list (2) (1st occur.)
- *J68 DELETE symbol in cell (0)
- *J69 DELETE (0) from list (1) (1st occurrence)
- *J70 DELETE last symbol from list (0)
- J71 ERASE list (0)
- J72 ERASE list structure (0)
- J73 COPY list (0)
- J74 COPY list structure (0)
- J75 Divide list after location (0); name of remainder is output (0)
- *J76 INSERT list (0) after (1), locate last symbol
- *J77 TEST if (0) is on list (1)
- *J78 TEST if list (0) is not empty
- *J79 TEST if cell (0) is not empty
- *J8n FIND the nth symbol on list (0)
- J9n Create list of n symbols, (n-1) to (0)
- *J100 Gen symbols on list (1) for (0)
- *J101 Gen cells of list structure (1) for (0)
- *J102 Gen cells of tree (1) for (0)
- *J103 Gen cells of block (1) for (0)
- J104

Auxiliary Storage Processes (§ 10.1)
- *J105 MOVE list structure (0) in from auxiliary
- J106 File list structure (0) in fast-auxiliary
- J107 File list structure (0) in slow-auxiliary
- *J108 TEST if list structure (0) is on auxiliary
- J109 Compact auxiliary data storage system (0)

Arithmetic Processes (§ 11.0)
- J110 (1) + (2) → (0), leave (0)
- J111 (1) - (2) → (0), leave (0)
- J112 (1) x (2) → (0), leave (0)
- J113 (1) / (2) → (0), leave (0)
- *J114 TEST if (0) = (1)
- *J115 TEST if (0) > (1)
- *J116 TEST if (0) < (1)
- *J117 TEST if (0) = 0
- *J118 TEST if (0) > 0
- *J119 TEST if (0) < 0
- J120 COPY (0)
- J121 Set (0) identical to (1), leave (0)
- J122 Take absolute value of (0), leave (0)
- J123 Take negative of (0), leave (0)
- J124 Clear (0), leave (0)
- J125 Tally 1 in (0), leave (0)
- J126 Count list (0)
- *J127 TEST if data type (0) = data type (1)
- J128 Translate (0) to be data type of (1), leave (0)
- J129 Produce random number between 0 and (0)

Data Prefix Processes (§ 12.2)
- *J130 TEST if (0) is regional symbol
- *J131 TEST if (0) names data term
- *J132 TEST if (0) is local symbol
- *J133 TEST if list (0) has been marked processed
- *J134 TEST if (0) is internal symbol
- J135
- J136 Make (0) local, leave (0)
- J137 Mark list (0) processed, leave (0)
- J138 Make (0) internal, leave (0)
- J139

Read and Write Processes (§ 14.0)
- *J140 Read list structure
- *J141 Read symbol from console
- J142 Write list structure (0)
- J143 Rewind tape (0)
- J144 Skip to next tape file
- J145 Write end-of-file
- J146 Write end-of-set

Monitor System (§ 15.6)
- J147 Mark routine (0) to trace
- J148 Mark routine (0) to propagate trace
- J149 Mark routine (0) to not trace

Print Processes (§ 16.1, 16.2)
- J150 Print list structure (0)
- J151 Print list (0)
- J152 Print symbol (0)
- J153 Print data term (0) w/o name or type
- J154 Clear print line
- J155 Print line
- *J156 Enter symbol (0) left-justified
- *J157 Enter data term (0) left-justified
- *J158 Enter symbol (0) right-justified
- *J159 Enter data term (0) right-justified
- J160 Tab to column (0)
- J161 Increment column by (0)
- *J162 Enter (0) according to format W43
- J163
- J164

In-process Loading (§ 19.0)
- J165 Load routines and data

Save for Restart (§ 20.0)
- *J166 Save on unit (0) for restart
- *J167 Skip list structure
- J168
- J169

Error Trap (§ 21.0)
- J170 Trap on (0)

Block Handling Processes (§ 17.0)
- J171 Return unused regionals to H2
- J172 Make block (0) into a list
- *J173 Read into block (0)
- *J174 Write block (0)
- *J175 FIND region control word of regional symbol (0)
- J176 Space (0) blocks on unit 1W19
- J177
- J178
- J179

Line Read Processes (§ 22.0)
- *J180 Read line
- *J181 Input line symbol
- *J182 Input line data term (0)
- *J183 Set (0) to next blank, leave (0)
- *J184 Set (0) to next non-blank, leave (0)
- *J185 Set (1) to next occurrence of character (0), leave (0)
- *J186 Input line character
- J187
- J188
- *J189 Transfer field to line (0)

Partial Word Processes (§ 23.0)
- J190 Input P of cell (0)
- J191 Input Q of cell (0)
- J192 Input SYMB of cell (0)
- J193 Input LINK of cell (0)
- J194 Set (1) to be P of cell (0)
- J195 Set (1) to be Q of cell (0)
- J196 Set (1) to be SYMB of cell (0)
- J197 Set (1) to be LINK of cell (0)
- J198
- J199

Miscellaneous Processes (§ 24.0)
- *J200 LOCATE (0)th symbol on list (1)
- J201 ERASE routine (0)
- J202 Print post mortem and continue

IPL INSTRUCTION: PQ SYMB LINK

P is operation code
- P = 0 Execute S
- P = 1 Input S (after preserving H0)
- P = 2 Output to S (then restore H0)
- P = 3 Restore (pop up) S
- P = 4 Preserve (push down) S
- P = 5 Replace (0) by S
- P = 6 Copy (0) in S
- P = 7 Branch to S if H5-

Q is designation code
- Q = 0 S = SYMB
- Q = 1 S = symbol in cell named SYMB
- Q = 2 S = symbol in cell named in cell named SYMB
- Q = 3 S = SYMB; start selective trace
- Q = 4 S = SYMB; continue selective trace
- Q = 5 Machine language routine
- Q = 6 Routine in fast-aux. storage
- Q = 7 Routine in slow-aux. storage

SYMB is symbol operated on by Q
LINK is address of next instruction (0 for end of routine)

SYSTEM STORAGE CELLS

H0 Communication cell
H1 Current instruction address cell
H2 Available space list
H3 Tally of interpretation cycles
H4 Current auxiliary routine cell
H5 Test cell

W0-W9 Common working storage
W10 Random number control cell
W11 Integer division remainder
W12 Monitor start cell (Q = 3)
W13 Monitor end cell (Q = 3)
W14 External interrupt cell
W15 Post mortem routine cell
W16 Input mode cell
W17 Output mode cell
W18 Read unit cell
W19 Write unit cell
W20 Print unit cell
W21 Print column cell
W22 Print spacing cell
W23 Post mortem list cell
W24 Print line cell
W25 Print entry column cell
W26 Error trap cell
W27 Trap address cell
W28 Trap symbol cell
W29 Monitor point address cell
W30 Field length cell
W31 Trace mode cell
W32 Reserved available space cell
W33 Cycle count for trap cell
W34 Current available space cell
W35 Slow-aux. obsolete structure cell
W36 Used slow-auxiliary space cell
W37 Slow-auxiliary storage density cell
W38 Slow-auxiliary storage compacting routine cell
W39 Fast-aux. obsolete structure cell
W40 Used fast-auxiliary space cell
W41 Fast-auxiliary storage density cell
W42 Fast-auxiliary storage compacting routine cell
W43 Format cell

IPL DATA: PQ SYMB LINK

Q = 0 Standard list cell:
- P is irrelevant
- SYMB is symbol
- LINK is address of next list cell (0 for end of list)

Q = 1 Data term:

	±PQ	SYMB	LINK
Decimal integer	1	dddd	dddd
Floating point	11	ddddd	d ±ee
Alphanumeric	21	aaaaa	
Octal	31	ddddd	ddddd

TYPE CARDS

0 (blank) Routines and data
1 Comments
2 Region definition
- NAME = Regional symbol
- SYMB = Origin (if given)
- LINK = Size

3 Block reservation
- NAME = Block control word (if given)
- SYMB = Origin (if given)
- LINK = Size
- Q = 0 Reserve regional symbols
- Q = 1 Reserve print line
- Q = 2 Reserve block
- Q = 3 Reserve auxiliary buffer
- Q = 4 Specify available space

4 Listing cards
5 Main storage header
6 Fast-auxiliary storage header
7 Slow-auxiliary storage header
8 Editing header; inhibits loading
- NAME = Name of storage block
- P = Input mode
  - P = 0 IPL standard
  - P = 1 IPL compressed
  - P = 2 IPL binary
  - P = 3 Machine code
  - P = 4 Restart mode
- Q = Type of input
  - Q = 0 Routines; internals symbolic
  - Q = 1 Data; internals symbolic
  - Q = 2 Routines; internals symbolic; reset internal symbol table
  - Q = 3 Data; internals symbolic; reset internal symbol table
  - Q = 4 Routines; internals absolute
  - Q = 5 Data; internals absolute
- SYMB = Alternate input unit
  - 0 (blank) = controlling unit
  - 1-10 = Internal tapes
  - Regional SYMB names first routine (terminate loading)
- LINK = Output mode: of form bbbcd
  - b = Output unit: blank = unit 1W19; 1-10 = unit 1-10
  - c = 0 (blank) if assembly listing
    - = 1 or any character if no assembly listing
  - d = 0 (blank) if no output
    - = 1 IPL compressed output
    - = 2 IPL binary output
    - = 3 Machine code output
    - = 9 IPL standard output

9 First card
- SYMB = Controlling unit (0 or blank = normal input unit)

IPL-V CODING SHEET

Problem No. _____ Programmer _____ Date _____ Page ____ of ____

COMMENTS	TYPE		NAME	SIGN	PO	SYMB		LINK		COMMENTS	I.D
06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40	41	42	43 44 45 46 47	48	49 50	51 52 53 54 55	56	57 58 59 60 61	62	63 64 65 66 67 68 69 70 71 72	73 74 75 76 77 78 79 80
6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	1	2	3 4 5 6 7	8	9 0	1 2 3 4 5	6	7 8 9 0 1	2	3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	