MEMORANDUM
RM-5270-PR
AUGUST 1967

JOSS: CENTRAL PROCESSING ROUTINES

J. W. Smith

PREPARED FOR:
UNITED STATES AIR FORCE PROJECT RAND

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-5270-PR
AUGUST 1967

JOSS: CENTRAL PROCESSING ROUTINES

J. W. Smith

This research is supported by the United States Air Force under Project RAND—Contract No. F44620-67-C-0045—monitored by the Directorate of Operational Requirements and Development Plans, Deputy Chief of Staff, Research and Development, Hq USAF. Views or conclusions contained in this Memorandum should not be interpreted as representing the official opinion or policy of the United States Air Force.

The RAND Corporation
1700 MAIN ST. • SANTA MONICA • CALIFORNIA • 90406

Published by The RAND Corporation

PREFACE

JOSS,† RAND's on-line, time-shared computing system, is designed to give the individual scientist or engineer an easy, direct way of solving numerical problems without recourse to professional computer programmers or to extensive, ad hoc programming education. To each user, JOSS appears to be a personal computing aide and file clerk, responding promptly to instructions couched in a simple language and transmitted over communication lines from the user's remote, electric-typewriter console.

Much of the ease and directness of the system may be attributed to the simplicity and readability of the language and to a collection of machine-language routines (in the system's central computer) for interpreting and responding to requests expressed in the language. The routines, when viewed collectively as an active agent, correspond to the "central processing unit" of the JOSS "computer."

This memorandum concentrates on the design of the language and of the central-processing routines. Design details are eschewed in favor of design considerations and decisions, and the general implementation of these decisions. The material is presented in a narrative style, augmented by flowchart representations of some of the principal routines, and is intended to provide students, evaluators, reproducers, and maintainers of the system with a source book and reference guide. It is also designed to serve as prolegomena to

---

†JOSS is the trademark and service mark of The RAND Corporation for its computer program and services using that program.

the annotated, machine-language listings of the routines (copies of which are obtainable from RAND) and can be read as an independent introduction to the system, the language, and the central-processing routines.

SUMMARY

JOSS is a multiuser, single-server computing system consisting of a collection of individual users' consoles connected to a central computing center by dual communication lines. The consoles are electric typewriters augmented by special electronics. The center consists of a general-purpose, digital computer,† with ancillary devices for storage and input/output, and a collection of machine-language routines "permanently resident" in the high-speed core storage of the computer. There are three sets of routines: (1) input/output routines for communicating with the users' consoles and their long-term (magnetic-disc) files; (2) central-processing routines for interpreting and responding to requests typed by the users and for interpreting the users' stored programs; and (3) supervisory routines for general management and accounting, and for giving each user a fair and proportionate share of processing. The center may be considered a single, active agent that serves the consoles and the users by time-sharing its activities; that is, it turns its attention from user to user so rapidly and smoothly as to give individual users the illusion of a single-user, single-server system. The three collections of routines may be viewed as active subagents, operating concurrently and synchronized by a common purpose. With this viewpoint as an introduction to the system, the single, active agent, JOSS, is described in terms of (1) the actions that can be requested of JOSS; (2) the language for requesting the actions; and (3) the parts played by the three

†The Digital Equipment Corporation PDP-6.

sets of routines in servicing the users and carrying out the requests.

Associated with each user is a collection of dynamically changing information: programs, data, information about actions initiated by the user, and other pertinent data. The central-processing routines service a user by processing his block of information. Although list structures are used extensively for storing information in the user's block, neither a general structure nor a general processor for lists is used. Instead, information is maintained in several distinct and simple list structures, whose representations are attuned both to the organization of the system's central computer and to the special nature of JOSS. These and other considerations concerning the storage and handling of information are discussed in detail, as are the structure and content of the information.

The organization of the central-processing routines mirrors the responsibilities of a central processing unit serving more than one user: communication with the users and with the supervisory routines; command interpretation and execution; intercommand sequencing and control; error diagnosis, control, and commentary. For purposes of storage and execution, commands typed by the users are represented as almost direct copies of the line typed by the user. The operation is interpretive: Although the execution of commands often requires that information be compiled and retained, no compilation in the usual programmatic sense is done. An extra level of interpretation controlled by a tabular representation of the rules for forming commands is not used. Instead, each type of command is handled by a distinct routine that "reads" like the rules for forming instances of the command type.

These routines are composed, in the main, of direct examinations of the primitive tokens entering into the command, mixed with uses of reentrant subroutines for interpreting more complex expressions. The major routines are described in terms of requests that can be made of JOSS, and in terms of JOSS's interpretation of the requests and responses to valid and invalid ones. Many design points of both the language and the responses to requests in the language are discussed in detail, as are major points of implementation of the routines.

ACKNOWLEDGMENTS

There are three facets to the construction of large software systems: (1) programming the system's components, (2) marrying the components, and (3) debugging the components and the entire schmeer.

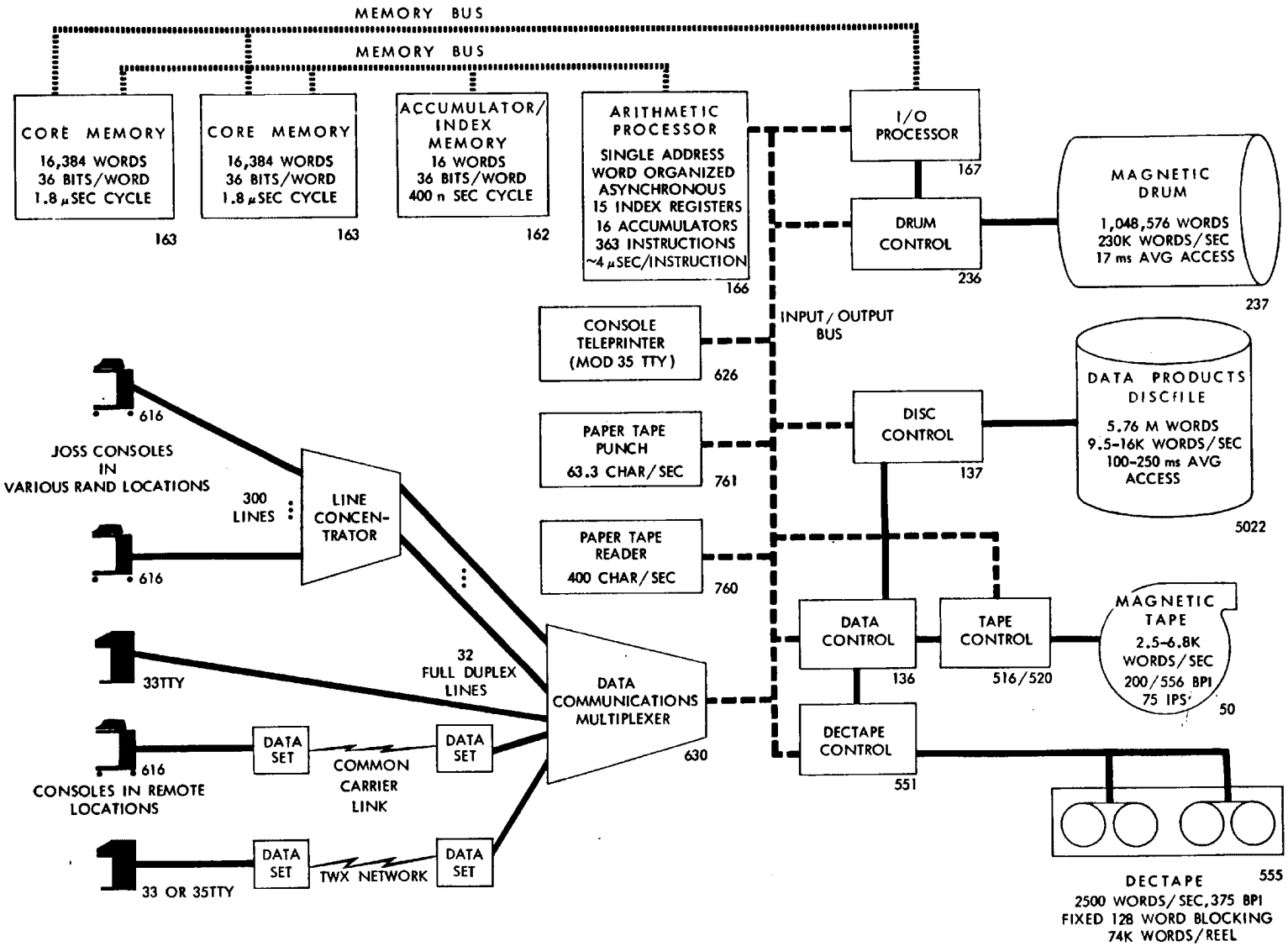
C. L. Baker, who supervised the effort, did yeoman's work in resolving design antinomies. The time-shared, software facilities provided by the Digital Equipment Corporation for their computer contributed significantly to all phases of the implementation effort. G. E. Bryan, who built the supervisory unit, and I. D. Greenwald, who built the input/output unit and the arithmetic/elementary function subroutine package, made the marriage painless: There was no interface problem.

The bulk of the acknowledgments must, however, go to O. A. Gross of RAND's Mathematics Department for his dedicated and marvelously ingenious probing of JOSS. The bugs and idiosyncrasies he uncovered, his suggestions for modifications and extensions, and the humor and subtlety he used when presenting them were a continual source of delight, enlightenment, embarrassment, and irritation. He must be classed as a national resource for system testing.

CONTENTS

FLOWCHARTS

TABLES



JOSS PDP-6 system

I. INTRODUCTION

JOSS† is an on-line, remote-console, time-shared computing service of The RAND Corporation designed to give the individual scientist or engineer an easy, direct way of solving numerical problems without recourse to professional computer programmers or to extensive, ad hoc programming education.

Functionally, JOSS is a multiuser, single-server computing service consisting of a collection of individual users' consoles connected to a central computer by full-duplex communication lines. The consoles are electric typewriters augmented by special electronics. The center consists of a general-purpose, digital computer,†† with ancillary devices for storage and input/output and a collection of machine-language routines "permanently resident" in the high-speed core storage of the computer. The center serves the consoles and users by time-sharing its activities; that is, the computer turns its attention so rapidly and smoothly among users as to give individuals the illusion of a single-user, single-server system.

The entire system is dedicated to a single task: providing a uniform, round-the-clock computing service to users, so that JOSS appears to be a personal "computing aide," privately interacting with the user and responding promptly and precisely to instructions couched in a simple language and transmitted from his electric-typewriter console.

---

† JOSS is the trademark and service mark of The RAND Corporation for its computer program and services using that program.

†† The Digital Equipment Corporation PDP-6. See facing page.

APERÇU

JOSS is an on-line, time-shared computing service of The RAND Corporation designed to appear to each user as a personal computing aide and file clerk, interacting privately with the users by means of their remote electric-typewriter consoles. Control of each typewriter is proprietary: Either the user has control for input purposes, or JOSS has control for output purposes.

Users request actions of JOSS by typing single-line commands.† A numeric label prefixed to the command is an implied directive to JOSS to retain the command as a step of a stored program, rather than to carry it out directly. JOSS retains steps in sequence, according to the numeric value of the label or step number. Thus the step number determines if an addition, insertion, or replacement is required.

Steps are organized into parts according to the integer parts of the step numbers. Steps and parts are units that may be edited, deleted, typed out, or filed in long-term storage. In addition, they are natural stored-program units for specifying, in a hierarchical manner, procedures to be carried out by JOSS.

Decimal and logical values may be assigned to any of the 52 upper- and lower-case letters admitted as identifiers. Values may be organized into vectors and arrays by using indexed letters, and the letters may be used to refer to entire arrays for deleting, typing, filing in long-term storage, and as actual parameters of formulas (see below).

In addition to values, arbitrarily complex expressions for values and letters may be assigned to a letter, which may then be used as an

†See Appendix for a summary of the language.

abbreviation for the expression; expressions so assigned are called formulas. Formulas involving formal parameters may also be assigned to a letter. The letter, accompanied by expressions for actual parameters, may then be used as an abbreviation for the formula with the actual parameters substituted for the formal ones. The letter itself may be used to refer to the formula for purposes of deleting, typing, filing, and as an actual parameter of formulas.

Expressions for the sum, product, largest and smallest of a set of decimal values, and for the first in a range of decimal values for which a condition holds, can be written succinctly and used as expressions for values. For example:

$$sum[i = 1(1)n: A(i)] \qquad max(x, y, z/3)$$

$$min[i = 1(1)n: A(i)] \qquad prod(x, y, z/3)$$

$$first[i = 1(1)n: A(i) \neq B(i)]$$

Except for the function *first*, either of the two notational styles may be used.† The conjunction or disjunction of a set of logical values can also be expressed in either of the two styles and used as expressions for logical values.

Short "programs" for choosing expressions for values differentially on the basis of a set of conditions can be expressed succinctly and used as expressions for values. For example, phrases such as

if $a = b$, use $x + y$;

if $a > b$, use x;

otherwise, use y

†Parentheses and brackets may be used interchangeably in pairs.

are expressed as

$$(a = b: x + y; a > b: x; y).$$

Such iterative functions and conditional expressions, together with formulas, lead to powerful, direct expressions for complex procedures, even recursive ones.

JOSS represents decimal numbers in scientific notation: nine digits of significance and a base-ten scale factor with exponent in the range -99 through +99. Addition, subtraction, multiplication, division, and square root are carried out to give true results rounded to nine significant digits; zero is substituted on underflow, while overflow yields an error message. In other elementary functions, care is taken to provide acceptable significance, to minimize discontinuities, to factor out error conditions, and to hit certain "magic" values on the nose.

The six numerical relations together with *and*, *or*, *not*, and a set of ad hoc logical functions may be used to express conditions, which may be attached to any step, and to express logical values.

A general rule governs the formation and use of expressions for values: With the exception of step labels, which must be decimal numerals, wherever a decimal (logical) numeral is allowed in a command, an arbitrarily complex expression for a decimal (logical) value may be substituted.

JOSS types answers one-per-line, identifying answers by the expression used in the step calling for the output; in the event of conditional expressions, JOSS uses only the chosen subexpression for identification. Decimal points and equal signs are lined up, and

fixed-point notation is used when reasonable. For more formal output, the user can define full-line forms to specify literal information and blank fields to be filled in with answers. A string of underscores with an optional decimal point is used to specify fixed-point fields; a string of periods specifies a tabular form of scientific notation.

Users can request JOSS to file, in long-term storage, identifiable units and collections of units--steps, parts, forms, formulas, and values. Users may then request JOSS to recall such filed items, discard them from the files, or type out a list of items in a file.

Users start JOSS off on the task of carrying out a stored program by directing JOSS to *Do* a step or part--repeatedly (for a range of values or a specified number of times), if desired. JOSS cancels all outstanding tasks before starting out on a direct (initiated from the console) task, begins the interpretation of a part at the first step of the part, and then interprets each step in sequence. Each subsequent indirect (initiated by a step of a stored program) *Do* causes JOSS to retain the status of the current task, drop a level to carry out the new task, and then return to pick up the suspended one. If the user wishes JOSS to act in the same manner for a directly initiated task, the command must be enclosed in parentheses.

JOSS modifies this general behavior whenever encountering

1. An error.
2. A branching command.
3. A stopping command.
4. A command for terminating a task or a portion of a task.
5. An interrupt signal from the user.

The deep and involved hierarchy of tasks and formulas that can occur (recursion is allowed) make it mandatory that JOSS's status be perfectly clear each time control is returned to the user, for any reason. In addition to error messages, interrupt messages, and stopping messages, JOSS types status messages on completion of parenthetical tasks to distinguish this state from the state of having finished a direct, nonparenthetical task. JOSS is able to proceed in every situation; in the event of errors, the user can take corrective action and then direct JOSS to continue with a *Go* command.

THE SYSTEM

For the system to function, the central hardware and the associated routines must be able to provide

1. Input/output, for communicating with the consoles.
2. Central processing, for interpreting and executing commands directed to the system by the users.
3. Supervision, for coordinating input, output, and processing, in order to provide users with a smooth and unbiased service, thus maintaining the illusion of a single-user system.

To perform efficiently, it is essential that the overhead functions of input, output, and supervision proceed asynchronously and concurrently with processing. This concurrence is realized by having the corresponding collections of routines time-share the central processing unit of the computer, in the same way that users time-share

the central processing "unit" of the JOSS system. The computer's hardware provides mechanisms for such time-sharing and for coordination and simultaneity of input, output, and processing.

An effective concurrence of JOSS functions is realized because the central processor of the computer spends a minimum of time interpreting and executing the input, output, and supervisory routines. The corresponding collections of routines may be viewed as independent units, operating asynchronously and concurrently, and the terms "Input/Output Unit," "Supervisory Unit," "Central Processing Unit" together with the abbreviations "IOU," "SU," "CPU" may be used.

REMOTE CONSOLE

The user's only visible link to JOSS is his remote console, which consists of an IBM Selectric typewriter (with a special character set) and a small, local-control box equipped with lights and switches. Control of the typewriter is proprietary: Either the user has control for purposes of input, or JOSS has control for purposes of output. Which of these situations holds is indicated by a set of audible, visible, and tactile signals.

JOSS is in control when

1. The red light is on.
2. The keyboard is locked.
3. Output is typed in black.

JOSS has returned control to the user when

1. A green light goes on, and the red light goes off.

2. The keyboard is unlocked.
3. Ribbon control shifts to green.
4. An audible tone is heard.

A power on/power off switch is provided on the console to request JOSS service and to disconnect the terminal when service is no longer required. Once connected to JOSS, three white status lights indicate whether

1. The console is working.
2. The JOSS system is working.
3. The typewriter is working and ready.

When the console is disconnected from JOSS, it may be used as a stand-alone typewriter, controlled by the typewriter's on/off switch. When the console is connected to JOSS, the typewriter on/off switch serves as a ready/hold switch to allow the user to suspend output typing (for example, if the paper supply becomes exhausted or the paper jams). Output can be continued after remedial action with no loss of information. When JOSS has control of the typewriter, an interrupt button and light allow the user to request that control be returned to him. JOSS turns on the interrupt light to indicate that the request has been sensed and honors the request as soon as possible.

Letters, digits, and punctuation marks occupy their customary positions on the keyboard. Left and right brackets and the absolute value bar are included to improve the readability of expressions. Parentheses and brackets are interchangeable in pairs for purposes of grouping. The six numerical relation symbols are added, together

with a centered dot for multiplication. The slightly elevated asterisk is used for exponentiation.

Four of the symbols have other applications:

1. The space sign (#) is used as a strike-out character, causing JOSS to replace the struck-out character by a space. JOSS also allows the user to backspace and strikeover characters, for purposes of correction.
2. The asterisk (*) at either end of a command input line causes JOSS to ignore the line and return control to the user without comment. This provides the user with a device for annotating his work and for canceling lines.
3. The dollar sign ($) may be used in any expression for a decimal value and carries a value equal to the line number of the typewriter's position on the page. This number (from 1 to 54) is updated by JOSS to allow the user to control output formatting. JOSS generally leaves a one-inch margin at the top and bottom of each page, although a 55th line will sometimes be typed on the page to prevent splitting two-line outputs over two pages.
4. The underscore (_) may be used to specify lines to be left blank by JOSS or to specify that JOSS is to leave fields empty (in user-defined forms for formal output).

INPUT/OUTPUT UNIT (IOU)

Communication between the center and the consoles is on a line-by-line basis. Typewritten lines from the users are terminated by a

carrier-return signal or a *page* signal. The IOU must accept and test incoming characters and signals, convert characters to an appropriate 7-bit encoding, collect these in a buffer area in core storage, and signal the SU whenever a line is terminated and whenever other signals--*on*, *off*, *in*--are received from the console. The IOU has the responsibility for maintaining exact images of typewritten lines, so that fixed-length buffers may be used.

On output, the IOU accepts outbound characters and signals from a buffer, converts these to the necessary encoding, transmits them to the desired console, and signals the SU when a line has been transmitted.

The IOU time-shares its activities among the various consoles under the control of hardware signals from a scanning and transmitting device that connects the communication lines to the computer.

CENTRAL PROCESSING UNIT (CPU)

Associated with each user is a collection of dynamically changing information: stored sequences of commands, data, information about actions initiated by the user, and other pertinent data. The CPU services a user--edits, displays, interprets, and executes direct (from the console) and indirect (stored-program) commands--by processing his block of information.

The CPU requires that a user's block of information be available in a continuous stretch of core storage for processing. Because core storage is not large enough to accommodate simultaneously all possible blocks, a secondary (magnetic-drum) storage is used to cache those not of immediate concern. The SU must see to it that the appropriate blocks

are available in core storage as required and must respond properly to CPU requests for more storage to handle an expanding user's block.

Two independently addressed, 16,384-word, core-storage units are available.† The JOSS machine-language routines reside in the lower addressed unit; one or more users' blocks reside in the second unit, which holds the blocks of immediate interest to the system.

To ensure that processing of the user's block be independent of the absolute core-storage locations used, the address-interpretation hardware of the PDP-6 was modified to distinguish address numbers greater than $2^{15} - 1$. Such addresses are reduced modulo 2^{15} and then added to the address number of the first word of the user's block (stored in a special "relocation" register) to obtain the required absolute address number. Thus, all addresses referring to the user's block are, in effect, relative addresses. Further, the scheme limits the size of users' blocks to 2^{15} words.

Some of the CPU's responsibilities and features are

1. Extensive error detection and alert monitoring of user actions.
2. Precise point-of-error reporting and careful error commentary.
3. The ability to clean up the user's block and back off to the beginning of a command or to other reasonable stopping positions to report an error.

In particular, the CPU makes no irreversible changes in a user's block until it is certain that the interpretation of a command or an editing

---

† The PDP-6 has an 18-bit address field.

action will be completed or reach a reasonable stopping position without an error.

In the time-sharing regimen of JOSS, processing of a user's block may be suspended for many reasons. Whenever a suspension occurs, the CPU must cache enough information in the user's block to allow the unit to later pick up the thread of processing on that block. Many suspensions of processing are the direct result of the processing itself: for example, errors occurring during command interpretation, execution, or interstep sequencing; input or output actions; insufficient buffers for transmitting lines to users; or insufficient core storage for an expanding user's block. Some suspensions are triggered by the SU when a user signals that he wishes to interrupt processing (by sending an *in* signal), and whenever the SU wishes to reassign the CPU to a different user's block. These signals are stored in the appropriate user's block by the SU. The CPU looks for such signals and acts on them with sufficient frequency to prevent processing jams.

SUPERVISORY UNIT (SU)

The SU uses a simple priority discipline to schedule the activities of the CPU. First priority goes to the servicing of signals from the IOU: *on, off, in, carrier return, page,* and *end of transmission.* Second priority is given to the resumption of output-limited tasks that have been set aside until the typewriters can catch up. Third priority is given to users with unfinished actions; these are handled on a round-robin basis.

The rationale behind this discipline is equally simple. Conceptually short, direct actions, such as editing of program and/or data and requests for "modest" computations, should be completed well before the typewriter carrier has physically returned to its rest point. Since most lines typed by a user result in such short actions, *carrier-return* and *page* signals are assigned a high priority. Output-limited tasks should produce a rhythmic pattern of output typing without stuttering. This is achieved by assigning a high priority to *end-of-transmission* signals and to output-limited users. Finally, high priority should be given to servicing users whose current behavior portends a short burst of processing followed by a long period of withdrawal, when no processing will be required (e.g., users who are interrupting JOSS, or are turning on or off, and output-limited users).

Users may ask JOSS to file specified collections of program and data in long-term (magnetic-disc) storage. The SU properly queues such requests as they are interpreted by the CPU, and the IOU effects the required transmissions of information between disc and core storage.

BACKGROUND

The JOHNNIAC Open Shop System (JOSS) was made available on a daily, round-the-clock basis to RAND personnel and associates in January 1964.[†] Implemented on the RAND-built JOHNNIAC computer, the system provided concurrent service to eight of ten available consoles. The incipient decline of JOHNNIAC and users' demands for more JOSS time (i.e., more consoles and a more reliable central computer), for

[†]A preliminary version saw limited use during most of 1963.

more storage space for programs and data, and for long-term storage and retrieval of programs and data led to a decision in 1964 to expand and retool the system using modern hardware. Changeover to the current system, implemented on the Digital Equipment Corporation PDP-6, took place in February 1966.

Many linguistic and processing capabilities above those required to satisfy the users' requests were incorporated in the new system, but never in such a way as to seriously stray from the intent of the parent system: to provide an easily learned, easily used, fail-safe computational service for the individual scientist or engineer.

The conversion of JOSS from the austere, 40-bit-word JOHNNIAC to the 36-bit-word PDP-6 had a profound influence on the internal design of the new system, particularly on the structure of the CPU and users' blocks. JOSS numbers that had been neatly represented in a single 40-bit word could not be represented in a single 36-bit word. Strings of characters for steps, forms, and formulas took up more 36-bit words than 40-bit words. The only alternative was a packing that would have made a mockery of processing-speed commitments. The preservation of almost nine-digit accuracy for the basic functions without sacrificing speed became even more of a coding challenge than it had been on the JOHNNIAC, despite the richer instruction repertoire of the PDP-6.

A complete redesign of the console and a decision to build the IOU in software (rather than hardware as in the original system[†]) added

---

[†] The JOHNNIAC had no interrupt logic.

to the conversion burden. Normal delays in delivery and debugging of the new consoles necessitated that the IOU be designed to accept TTY's as well as JOSS consoles, adding further to the cost of conversion (though resulting in a useful fallout).

Obviously, users could not be penalized by the conversion to the new system. Neither the number of users that could be accommodated nor the amount of storage available for a user's block could be less than that admitted by the original system. In effect, this meant that no great departure from the original block design could be made, particularly in view of the 10 percent degradation in storage capacity caused by the 40-bit to 36-bit change. The decision to abide by this precept paid off handsomely--the conversion was made long before the magnetic drum arrived.

The excellent, time-shared software facilities provided by the Digital Equipment Corporation for the PDP-6 permitted the concurrent implementation of the IOU, SU, and CPU via the console TTY and two additional TTY's obtained specifically for that purpose. Concurrent debugging of the different units was made possible by the construction of a version of the CPU (with a surrogate SU) that operated within the DEC time-sharing system. This version of the CPU is still used, as a matter of strict policy, to validate modifications and additions to JOSS before they are presented to the users.

II. THE USER'S BLOCK OF INFORMATION

When a user turns on his console, the SU leads him through a log-on ritual to obtain information for accounting purposes, and then places him in the hands of the CPU. Thereafter, each line typed by the user is interpreted by the CPU, and each line typed by JOSS is either composed by the CPU or produced under its direction. In a real sense, the user interacts with the CPU, although the speed with which JOSS responds to the user's requests is determined not only by the CPU's internal processing speed but also by the SU's ability to give each user a fair share of the CPU's time. Therefore, it is appropriate to describe the CPU's gross structure in terms of JOSS's interpretation of lines typed by the user and JOSS's responses to them.

Apart from information required by the SU for scheduling, and volatile information generated by the CPU during processing, all information relevant to the user is kept in a piece. During suspensions of processing, the user's block of information may be cached on a magnetic drum by the SU; during processing, it is maintained by the CPU in a consecutive stretch of core storage, whose precise location may change after any suspension of processing. Much of the CPU's fine structure can best be demonstrated by examination of the user's information block.

The only detailed description of the CPU's implementation on the PDP-6 is furnished by the annotated, machine-language listing of the central-processing routines, which is mainly of value to maintainers

of that particular implementation. A thorough knowledge of JOSS's behavior and of the structure of the user's block is a prerequisite to reading the listings and to reproducing the CPU on a different computer. Accordingly, the user's block will be examined in detail, and the central-processing routines will be described behavioristically. Major points of the implementation that were strongly influenced by the PDP-6 computer will be discussed when appropriate; otherwise, only obscure or controversial points of the implementation will be taken into consideration.

NOTATION AND TERMINOLOGY

It is difficult to describe the fine encoding of some of the information in the user's block without recourse to the 36-bit, word-oriented central processing unit of the PDP-6. In particular, because a 36-bit word in core storage may often represent more than one item of information, a notation for word fragments must be introduced.

The notation

$$(i, j - k)$$

is used indifferently to refer to the quantity represented by bits j through k of the i*th* 36-bit word of the set of words being described, or to the storage fragment associated with the $(k - j + 1)$ bits. To conform to PDP-6 machine-code requirements, the most significant bit of a word is called the "zeroth" bit; the least, the "35th" bit. Thus,

$$(2, 0-35)$$

would refer to the complete second word of the set of words being described.

A cell is two consecutive words.

A link is a relative (to the beginning of the user's block) address greater than zero, or it is zero (indicating the last element of a list).

A *left (right)* linked cell list is a collection of cells, simply ordered, with each cell containing a link to its successor in *(2, 0-17)*, *(2, 18-35)*, respectively.

An object is simply an element of discourse of the language or of the CPU; for example, an array, a decimal number, or a range of values for iteration.

A descriptor is a one-word set of information that, in the proper context, uniquely represents the descriptee.

An object descriptor describes the direct representation of an object in the user's block.

The assignment table is a set of 52 contiguous cells, one for each letter of the double-barreled alphabet, each being the first on a push-down list of object descriptors and ancillary information.

An object's use count is a count of the number of descriptors for that object that happen to be in the assignment table at any instant (multiple assignments can occur during formula evaluations).

An object header is a cell containing information about complex objects, such as arrays, formulas, parts, steps, and forms.

A list header is a single word containing a link to the first cell of a cell list.

A line number is a nonnegative integer used as a relative address in a table of items.

A pointer is a 36-bit quantity describing a specific word fragment or byte, used by all PDP-6 byte-handling instructions as a description of the byte to be manipulated.

GENERAL STRUCTURE

Information in the user's block is organized into

1. SU-maintained identification and accounting information.
2. CPU-maintained console information: type of console, page number, and line counter.
3. CPU working storage for preserving major context during processing and over suspensions of processing.
4. PARTS: a list of the representations of the literal strings of characters for steps of the user's stored programs, ordered by step number and aggregated by part number.
5. FORMS: a list of the representations of the literal strings of characters for the user's forms for formal output, ordered by form number.
6. A table of 52 entries for keeping track of assignments to letters.
7. JPDL: a push-down list for keeping track of hierarchical tasks.
8. DS: a push-down list for keeping track of "operands" during interpretation.

9. PS: a push-down list for keeping track of "operators" during interpretation.
10. FPDL: a push-down list for keeping track of place markers during the interpretation of formulas and iterative functions.
11. ACL: a list of available space units.

Direct representations of objects are never stored on the operand stack, DS, or in the assignment table. Instead, all such bookkeeping is done in terms of descriptors that represent objects indirectly. Whenever available space in the user's block is used for storage of objects, a record of the transaction is stored in either the part list, the form list, the assignment table, or on the operand stack. The operator stack, PS, and the formula stack, FPDL, contain only direct representations of operators, place holders, and other storageless information; therefore, cleaning up and backing off after errors requires only that PS, FPDL, DS, and storage for the latter's objects be transferred back to the list of available space units.

LIST STRUCTURES AND LIST PROCESSING

Because of the nature of JOSS--as a language and as a system--list structures and list processing are integral to the structure of users' blocks and the CPU. The representation of lists and their elements, and the strategy used in list processing (in particular, the handling of available space), depend on many factors, including

1. The kinds of elements and structures to be processed.
2. The kinds of processing to be done on elements and structures.

3. The particular computer to be used.
4. System specifications.
5. Desired balances between storage economy and processing efficiency.
6. The personal taste of the implementer.

The special-purpose nature of JOSS led to a fairly simple choice of representation and strategy. It is always difficult to justify such a choice, but an attempt must be made.

First, available space is partitioned into two-word cells, which are the atomic storage elements of all list structures in the user's block. The use of standard techniques for handling variable-size cells (usually multiples of two words per cell) was ruled out on at least two counts. Most elements of the user's block fit neatly into two-word cells, and activities involving such cells far outweigh those involving larger cells. Consequently, any general scheme for handling variable-size cells would have caused a significant decrease in the CPU's processing speed and a very slight loss in storage capacity. Indeed, with the relatively few types of list structures required in the user's block, it would have been far simpler and more economical of both storage and processing time to take advantage of consecutive cells only during high-payoff situations--when storing steps, forms, and formulas.

However, even such an easily implemented and highly effective strategy was ruled out by the insistence on a certain consistency of system behavior: Equipment willing, programs and data that have once been accommodated must always be accommodated. The combination of

finite core storage and variable-size cells could have caused a configuration of program and data arrived at by different user actions, or by the same actions in a different sequence, to fit in some cases and not in others. Thus, variable-size cells were discarded in favor of the two-word cells, despite the fact that nonaccommodation of once-accommodated users' blocks would occur so infrequently that the CPU could well afford to reorganize storage on such occasions. This is a straightforward coding job, but, unfortunately, one that requires temporarily assigned, additional core storage. In the author's opinion, it would have been far better to rely on the user's sense of civic responsibility and in extreme cases to allow him to request a little more than the normally allotted maximum storage.

Other compromises for handling storage were considered and rejected. Volatile information used during command interpretation (in particular, during expression evaluation) could have been stored outside the user's block to be cached in the user's block only on suspensions of processing. Or, the user's block could have been partitioned into "buckets" of varying capacity, one for volatile information, the others for slowly changing information such as steps, forms, and assignments (to be maintained in a compact form by rearrangement as required). Such techniques would have achieved gains in processing speed by permitting the use of stacking operations rather than list-processing operations on push-down lists. For JOSS, these gains were small and worth neither the complexity nor the effort required to achieve them. It must be recalled that JOSS is a computer and not a list processor, although list structures and list processing are necessary to fulfill

that role. In fact, less than 10 percent of the CPU's time is devoted to list processing, and this rate would increase to no more than 12 percent were the arithmetic and the basic function calculations to be done "in hardware" rather than by routines. (These are conservative estimates.) Thus, the entire question becomes an academic one. The handling of available space, in particular the question of what to do with released cells, is a standard problem. Should the cells be returned to the available space list when released or should they be collected only in extremis? In the latter case, should they be marked to facilitate subsequent collection, or not? Marking released lists either restricts the range of representations that may be used or costs a little in storage capacity. Moreover, for the simple lists used in JOSS, marking takes as much time as returning a cell to the available space list. The simplest technique would have been to collect only when necessary, using a "storage map" outside the user's block for marking during collection.

The decision to eschew collection in favor of returning cells when released was influenced by the fact that all list processing (with the exception of storing and releasing steps, forms, and formulas) could be accomplished by a collection of short, fast macro's, most of which are composed of a few machine instructions involving transfers between core storage and high-speed registers. Since macro's are used extensively, it would be a simple matter to change to a strategy of collection and then compare speeds.

In summary, available space is partitioned into two-word cells, which are returned to the available space list when released. Neither

a general structure for lists nor a general list processor is required or used. Instead, each of the few types of list structures is handled by machine-language routines or macro's that "know" the structure of the specific list being processed.

REPRESENTATION OF STRINGS OF CHARACTERS

The IOU receives and transmits characters in a 6-bit, typewriter-dictated, "tilt-rotate" code with parity bit. These are translated within the IOU into a 7-bit encoding, and again translated within the CPU into a CPU-specific, 8-bit encoding. The CPU does little more than strip off trailing spaces and step numbers, note the position of conditional clauses, and translate words in JOSS's vocabulary (and strings of consecutive spaces) into single, 8-bit codes.

The decision to represent commands and formulas in essentially their original form requires some comment. The original system made no syntactic checks of steps before storing them away, because of JOHNNIAC's limited core storage. Users were not overly penalized since they could write only small programs. Duplicating this behavior in the new system was probably a mistake, despite the fact that many users now use JOSS to build files of textual information ordered into steps and parts. Nevertheless, there are a variety of representations of partially transformed steps and formulas, amenable to efficient interpretation and execution, that could be used without demanding error commentary during the required preprocessing. (It is important that an equivalent of the original typewritten form of steps and other elements be retrievable from such representations,

for display to the user.) These representations are usually strung out or tabularized, postfix equivalents of the original strings, with enough ad hoc "operators" provided to clarify ambiguities. Such transformations speed up subsequent processing by stripping off parentheses, reordering operations, and replacing the usual infix notation by an equivalent postfix representation.

In word-organized machines such as the PDP-6, the tabularized representation leads to more rapid processing by allowing the coder to take advantage of the natural syllabic structure and/or instruction format of the computer, with a consequent loss of storage capacity. An analysis of such representations in the light of JOSS's language and of the PDP-6 showed that the increase in processing speed would be marginal in terms of the effort involved. Tabularized representations took up too much storage space, while packed or strung-out representations were no more economical of storage than the one-to-one representation chosen. (Numerals are usually short, and heavily parenthesized expressions rare.) Finally, the transformed representation produced significant gains in processing speed only with heavily parenthesized expressions. The representation chosen leaves JOSS open for the possible inclusion of commands for text construction and editing.

Before storage of strings for steps, forms, and formulas, trailing blanks are stripped off, and recognizable words and consecutive strings of blanks are replaced by single-byte encodings. Strings for steps undergo further processing. First, a trailing point is replaced by a unique byte denoting a period. Second, the last occurrence of

the word "if" not enclosed by quotation marks is replaced by a unique encoding to distinguish it as the beginning of a conditional clause. Finally, the step number itself is stripped off and replaced by three leading bytes. The first byte contains a count of the number of leading blanks in the original string plus the number of leading zeroes in the step number. (These are replaced by blanks whenever JOSS types the step.) The second byte contains a count that enables JOSS to type accurately the fractional part of the step number when typing the step. The third byte contains the index of the byte preceding a conditional clause if one seems to exist; otherwise, the index is zero. A specific code in the third byte, indicating that the step was not ended by a period, permits the interpretation of such steps to be aborted.

The third byte is an artifact for speeding up interpretation of commands by bringing conditional clauses into focus quickly and easily. Other devices could serve the same purpose; for example, conditional and imperative clauses might be interchanged before storage. The sole virtue of the device used is its simplicity and ease of implementation.

Stripping off step numbers from steps before storing generally saves a little space for users; its genesis, however, lies in a personal feeling that it should be possible to divorce the text of a step from the step's number with ease, an action that would be required for subprogram construction and text manipulation were such facilities to be included in the system's repertoire. This is achieved at a small cost to users of the system. (Tabs preceding a step number are not allowed.)

ACCOUNTING AND COMMUNICATION (Table 1)

When the user turns on his console, he is assigned a block of 1024 consecutive words of storage by the SU; during subsequent processing, additional, consecutive units of 1024 words are added as required by the CPU. The CPU routine X43 sets information in the user's block to initial conditions after completion of the log-on ritual, which is led by the SU. Initially, the user's block contains

1. SU-maintained identification and accounting information.
2. CPU-maintained console information: type of console, page number, and line counter.
3. CPU working storage.
4. Heads of empty lists for parts, steps, and forms.
5. Heads of empty push-down lists for keeping track of hierarchical tasks and for keeping track of operands, operators, and formulas during interpretation of commands.
6. A table of 52 entries for keeping track of assignments of values and formulas to identifiers.
7. A list of available cells to be used for storing steps, forms, and values and for bookkeeping during processing.

The CPU routine, whose address resides in the first word of the user's block, is used for all subsequent resumptions of processing. Currently, this is always the CPU routine X44; it is used to determine user/CPU state and to trigger or continue the appropriate routines. The determination is made on the basis of a reentry code and a CPU address number that are stored in the user's block. If

Table 1a

ACCOUNTING INFORMATION AND COMMUNICATION CELLS

LABEL	DESCRIPTION
INTENT	Address of routine, X44, used for all resumptions of processing
SEQ	Used by SU
INITIALS	The user's initials
JOBNO	The user's project number
PAGNO	The user's current page number
ONTIME	The time when the user logged on
COMTIME	Cumulative processing time since log-on
SPARE1-SPARE5	Used by SU
RISIG	Made nonzero by SU to notify CPU that the user has transmitted an *in* signal
UBUF	Temporary cache for address of last console buffer attached to the user by SU
ME	(1, 0-17) = a reentry code[a] used by CPU to resume processing of the user's block (1, 18-35) = the CPU address, X44
RETURN	(1, 18-35) = zero if the reentry code is to be used = CPU address where processing is to be picked up if reentry code is not to be used
WIDTH	Maximum allowable number of characters for lines to and from the user's console; 79 for JOSS consoles, 72 for TTY consoles

[a] 0: User has just logged on; CPU must preset the user's block.
1: CPU is expecting a direct command of the user.
2: CPU is expecting a response to a demand for input.
3: CPU is expecting a form.

the address number is zero, action is taken on the basis of the reentry code; otherwise, processing continues at the addressed instruction. Processing for the user is aborted if either the reentry code or the address is invalid.

ME and RETURN could become the first on a push-down list of cells for saving the states of interrupted central processors if another type of processing facility were to be added to JOSS (e.g., text editing). Note that the first twenty-odd items of information in the user's block are either maintained by the SU or would have to be maintained uniformly by any central processor.

WIDTH, which is updated by the CPU whenever a line is typed by the user, is used to control the width of output lines. Information about the type of console is recorded with the line's image by the IOU, as are notes about overlong lines. These are the only instances of direct communication between the IOU and the CPU, except for status information during disc operation.

The number of available cells is updated each time a cell is removed from or restored to the available space list. Originally, JOSS displayed this value whenever the user requested that *size* be typed; in the current system, the value displayed is the number of cells currently used for storage of steps, forms, assignments, and information about suspended and ongoing tasks.

Responsibility for keeping track of the user's line count rests with the CPU. The maximum number of lines per page (exclusive of the page heading and margin) is 54, except when an extra line is required to avoid splitting double lines.

Table 1b

SIZE, TIME, USERS, TIMER, AND LINE COUNTER

LABEL	DESCRIPTION
SIZE	Current number of available cells
SPACE	The number of 1024-word core blocks currently assigned to the user
LINE	Current value of the line counter, *$*
USIZE	Number of cells currently used for storage of information in the user's block *(size)*
UTIME	Current 4-digit time, as an integer *(time)*
UUSERS	Current number of users being serviced *(users)*
UMIN	Number of minutes that have elapsed since user logged on or last reset the value of *timer* to zero, rounded to the 1/100th minute *(timer)*
USEC	Value of the SU's seconds counter when user logged on or when he last reset the value of *timer*

The value displayed for *size*, *time*, *users*, and *timer* are set prior to the interpretation of every command, and only then. Thus, the values associated with these items and with *$* remain fixed during the interpretation of a command.

The high-speed registers of the PDP-6 are cached in the user's block whenever the CPU suspends processing for any reason. Register CR is stored only to facilitate debugging.

The *A* and *B* sets of registers are used variously. In particular, they contain the first and second arguments on entrance to the arithmetic and function subroutines. Results of these subroutines are always found in the *A* bank. The first of each set contains the sign and

Table 1c

HIGH-SPEED REGISTER CACHES

LABEL	DESCRIPTION
UCR	Cache for register CR, PDP-6 stack pointer
UA1	Cache for register A1
UA	Cache for register A
UA2	Cache for register A2
UB1	Cache for register B1
UB	Cache for register B
UB2	Cache for register B2
UACL	Cache for register ACL; address of first cell on available space list
UDS	Cache for register DS; address of first cell on operand-descriptor push-down list
UPS	Cache for register PS; address of first cell (second entry) on operator-descriptor push-down list
UCP	Cache for register CP; first entry on operator-descriptor push-down list
UCC	Cache for register CC

digit part of the argument and the digit part of the result; the second contains the sign of the result; and the third contains the exponent part of the argument and the result in two's complement representation.

Register CC usually contains either the current 8-bit (or 7-bit) character code being examined or a descriptor for the current, primitive syntactic element being examined.

Except for register ACL, all the registers are used loosely during preprocessing of incoming lines. After that, their use becomes

fixed (with the exception of the above-mentioned *A* and *B* banks and register CC, which is frequently free).

PLACE MARKERS AND OTHER INTRASTATEMENT CONTEXT (Table 2)

The PDP-6 byte-handling instructions all use reference words that describe the specific byte to be manipulated. These are called "pointers," "byte pointers," or "string pointers." A pointer "points at" the byte described and "points to" the byte following the one described.

Except for a few instances during the preprocessing of incoming lines and the composition of lines for output (when pointers may reside in high-speed registers), item U1 always points to the next byte to be examined.

Item U4 is used solely during expression evaluation to keep track of the relative precedence of operators.†

The CPU types only the chosen subexpressions of conditional expressions when identifying values for the user. This is accomplished by marking the beginning and end of each chosen subexpression encountered during the interpretation of *Type* commands,†† which are copied into working storage in the user's block before being interpreted. Item U6 is used to control the marking.

To allow users to type, file, and delete simply described objects when no space is available, three cells of available space are held back for use when typing, filing, and deleting. Item U7 is used to note that the three cells are being used.

---

†See p. 107.

††This seemed simpler than saving pointers or copies of substrings (see p. 134).

Table 2a

PLACE MARKERS AND OTHER INTRASTATEMENT CONTEXT

LABEL	DESCRIPTION
U0	Pointer to beginning of current command, form, or demand response being examined
U1	Pointer to next 8-bit byte (of current string) to be examined
U2	Numeric code for the imperative or declarative verb associated with the current command
U3	Descriptor for the expected delimiter of the current imperative clause being interpreted; either a period, a special token for *if*, or an end-of-string token
U4	Hierarchical weight associated with the last operator or punctuation mark encountered
U5	Used variously for temporarily relevant string pointers
U6	-1 if CPU is interpreting the main string of a *Type* command; otherwise, nonnegative
U7	-1 whenever the three-cell ace in the hole is being used for typing, filing, or deleting
U8	Used as a cache during typing, filing, and deleting
FPDL	Formula push-down-list header
LEVEL	(1, 18-26) = current level of subprogram nesting
	(1, 27-35) = current level of formula nesting
BASE	Current level of subprogram nesting

Item LEVEL is used to exercise control over the scope of formal parameters of formulas and, in the event that subprogram structures are added to JOSS, the scope of all identifiers.

The Formula Push-Down List (FPDL) is a simple, right-linked cell

list for keeping track of context when interpreting formulas and iterative functions. (See Table 2b.) In both cases, a place marker in the string of characters being examined must be saved so that scanning can be resumed correctly. Each entry for a formula carries the 8-bit code for the identifying letter and the number of parameters associated with the formula. Cleaning up and backing off require that FPDL's cells be released.

The list header resides permanently in the user's block, but is brought into a high-speed register as required.

Table 2b

FORMULA PUSH-DOWN LIST (FPDL)

(1, 0-35) = U1 when entry was made; where to resume scanning after formula is evaluated, or beginning of the expression to be evaluated iteratively if entry is for a function

(2, 0-7) = 8-bit code for letter if a formula; zero if a function

(2, 8-17) = number of parameters if a formula; identifying code if a function

0 indicates *sum*
1 indicates *prod*
2 indicates *max*
3 indicates *min*
4 indicates *disj*
5 indicates *conj*
6 indicates *first*

WORKING STORAGE (Table 3)

An exact image of each line received from the disc or from a console is copied into item US0. This enables the CPU to restart

Table 3

WORKING STORAGE

LABEL	DESCRIPTION
US0	Pointer for linear working string 0 followed by storage for 85 7-bit bytes
US1	Pointer for linear working string 1 followed by storage for 88 8-bit bytes
US2	Pointer for linear working string 2 followed by storage for 88 8-bit bytes
US3	Pointer for linear working string 3 followed by storage for 20 8-bit bytes
US4	Pointer for linear working string 4 followed by storage for 32 8-bit bytes
US5-US7	Caches for pertinent pointers
UP0-UP12	Scratch storage for typing arrays
UX1-UX4	Context (calling addresses) for various routines used when typing and deleting

processing of direct lines without requesting the user to repeat. In particular, if the processing of a direct line causes the CPU to request more core storage of the SU, the CPU cleans up and backs off to the beginning for a retry after space has been obtained.

The preprocessed images of input lines are found in US1, as are the copies of *Type* commands. Item US2 is used variously for composing output lines, as are US3 and US4. Item US3 is used for small fragments of output lines; US4 is generally used for numeric strings.

The pointer caches, US5 to US7, generally contain pointers to strings fixed in the CPU storage--error messages, strings of spaces for indentation of output lines, etc.

The execution of *Type* commands involves requests for assignment of line buffers and for transmission of buffers to consoles, so that the CPU frequently suspends processing of the user's block while typing. Items UP0 through UP12 are used variously as caches during such suspensions, as well as for other purposes. In particular, item UP12 is used as a push-down stack for typing arrays, the ten words of the stack mirroring the restriction to ten index values for indexed quantities.

KEEPING TRACK OF HIERARCHICAL TASKS (Table 4)

Each *Do* command causes the CPU to preserve information about the current state of processing, to drop a level to perform the indicated step or part as a subroutine, and then to return to pick up the interrupted thread of processing. If the command carries a modifying clause, the appropriate list structure is built up. The information that must be built up for the control of a task consists of

1. A record of what is being done--part or step;
2. The number of the part or step being done;
3. The list structure for the modifying clause, if necessary;
4. A record of how the command was given, either directly or indirectly.

For convenience in sequencing and control, part and step numbers of interest are fragmented into

1. A part index--the integral part of the number, represented as a binary integer;

Table 4a

TASK CONTROL INFORMATION

LABEL	DESCRIPTION
MODE	0 indicates CPU is processing a direct line ±1 indicates CPU is processing an indirect step 2 (3) indicates CPU is advancing the iteration variable associated with a direct (indirect) task
JPDL	Link to first cell on push-down list for hierarchical tasks
JD	Information about current task
JOB-CODE	(JD, 0-3) = 2 if doing step = 1 if doing part = 0 otherwise (JD, 4-5) = MODE when command was given
BREAK	(JD, 6-9) = 1 if iteration variable must be advanced before going on = 0 otherwise
JOB-MODE	(JD, 10-11) = MODE when task was interrupted
SKIP-CODE	(JD, 12-13) = 0 indicates next indirect command to be processed is the current one = 1 indicates next indirect command to be processed is the one after the current one (JD, 18-35) = link to modifying phrase list structure
U24-U25	The number of the part or step associated with the current task
CPI	Integral part of the current step number
CSI	10^8 times the fractional part of the current step number
CSA	Link to the current step header

2. A step index--10^8 times the fractional part of the number, represented as a binary integer.

The record of how the command was given is referred to as its *mode;* that is, in "the direct mode" or "the indirect mode."

Whenever the CPU drops a level to begin a new task, all of the above information must be saved. In addition, the CPU must save the information required for picking up the interrupted task on completion of the interrupting one. All necessary information is saved on JPDL (Table 4b), a simple, right-linked cell list with an implicit entry size of three cells. For convenience, the CPU recognizes the existence of the *null* task, when there is nothing to be done but await directions from the user.

Table 4b

PUSH-DOWN LIST FOR HIERARCHICAL TASKS (JPDL)

(1, 0-35)
(2, 0-17) = number, in JNF (see p. 50), of part or step to be done

(2, 18-35) = link to next cell

(3, 0-35) = CPI when task was deferred

(4, 0-17) = link to header for associated modifying phrase

(4, 18-35) = link to next cell

(5, 0-35) = CSI when job was deferred

(6, 0-17) = JD when task was deferred

(6, 18-35) = link to first cell of next triple of cells

CONTEXT FOR DISC OPERATIONS (Table 5)

Items are filed on magnetic-disc storage as collections of literal strings of characters for steps, forms, value assignment commands, and formula assignment commands in the standard 8-bit encoding. Thus, the routines for filing and recalling items reduce to routines for controlling extant routines for processing typewritten lines from the console and for composing and transmitting lines to the console. Items UDF1 and UDF2 are used for controlling these routines.

Items on the disc are ordered physically into 128-word records. A single 128-word buffer area is used to transmit records between disc and core storage, UBFR being used for sequencing through the strings in the buffer area.

Table 5

CONTEXT FOR DISC OPERATIONS

LABEL	DESCRIPTION
UDF1	-1 if CPU is using the disc; 0 otherwise
UDF2	-1 if CPU is expecting a form to be input from either a console or the disc; 0 otherwise
UBFR	Pointer to next byte in 128-word buffer used in disc transmissions
UFILE	The number of the disc file being used
UKEY	The key associated with the current file
UNAME	The key for the current file item being referenced
UITEM	The number of the current file item being referenced

KEEPING TRACK OF PARTS, STEPS, AND FORMS (Table 6)

The literal strings of characters for steps and forms are stored as elements of appropriately ordered list structures. Two-word headers for these are carried in fixed locations in the user's block labeled PARTS and FORMS, respectively.

Table 6

PART LIST AND FORM LIST

PARTS: (1, 0-35) = -1
(2, 0-17) = link to first part header (for lowest numbered part)
(2, 18-35) = reserved for potential use in sub-programs

Part headers:
(1, 0-35) = part number as a binary integer
(2, 0-17) = link to next part header (ordered by part number)
(2, 18-35) = link to first step header (lowest numbered step in part)

Step headers:
(1, 0-35) = 10^8 times the fractional part of the step number
(2, 0-17) = link to first cell of the string of characters for the step
(2, 18-35) = link to next step header (ordered by step number)

FORMS: (1, 0-35) = -1
(2, 0-17) = link to first form header (for lowest numbered form)
(2, 18-35) = reserved for potential use in sub-programs

Form headers:
(1, 0-35) = form number as a binary integer
(2, 0-17) = link to next form header (ordered by form number)
(2, 18-35) = link to first cell of the string of characters for the form

Strings of characters for steps, forms, and formulas are represented in the user's block as sequences of 8-bit bytes. Each byte represents either a single character, a word in JOSS's vocabulary, or a set of one to eight consecutive spaces. Strings are stored in lists of cells, seven bytes to the cell. Each string is terminated by a unique, 8-bit code (EOS); and the seventh byte in each cell is always one of sixteen 8-bit codes denoting end-of-cell. The overlapping of the seventh byte with the address of the next cell requires the use of a set of such indicators.

Recall that address numbers used to refer to the user's block are distinguished by being greater than $2^{15} - 1$ (but less than 2^{16}). Thus, the 8-bit octal codes *110* through *117* may serve as end-of-cell indicators; the codes *100* through *107* serve the same purpose. (This takes care of zero links and also allows the CPU to run as an independent unit, outside the JOSS system, on users' blocks that use absolute address numbers.)

As an example, the string fragment *if x = a+b.* requires two cells for storage and would appear as follows in, say, cells 235 and 313:

235	*if*	^	*x*	^	••••
236	=	^	01001000	00001100	1011

313	*a*	+	*b*	PERIOD	••••
314	EOS	EOS	01000000	00000000	0000

The four unused bits in the first word of each cell are ignored, and the choice of odd cell numbers does not indicate that cells are so constrained.

The use of end-of-cell indicators not only speeds up the scanning and general interpretation of strings (no counting or testing is required), but also makes string manipulations independent of string storage. This independence is used in manipulation on so-called linear strings that are stored in consecutive words at four bytes per word (e.g., strings of characters for error messages).

Step numbers are stripped off and replaced by three byte fields. The first holds the number of spaces preceding the step number, the second the number of trailing zeroes and/or decimal point. For example:

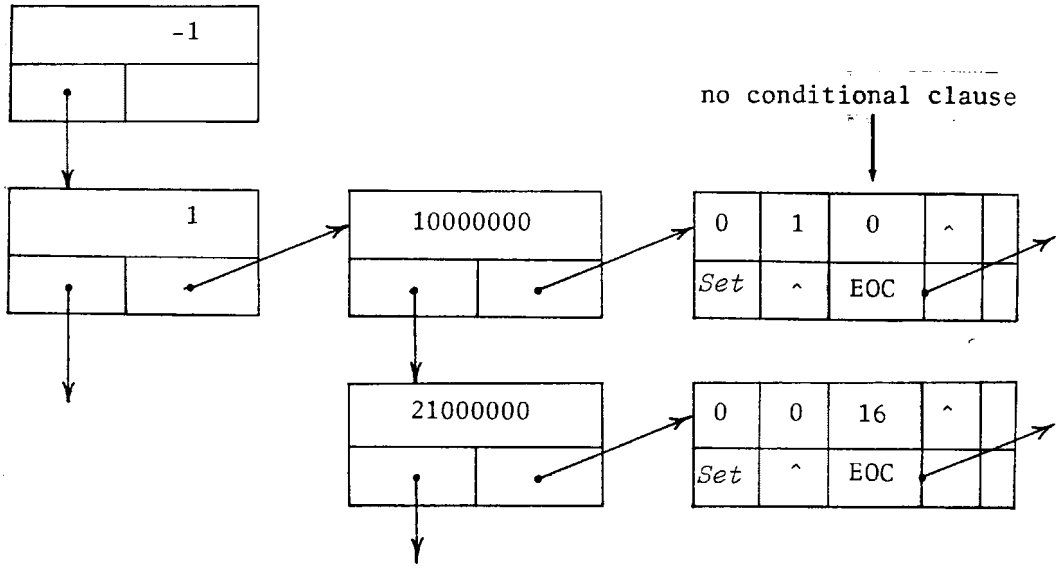
Line Image	First Byte	Second Byte
^^1.35 ...	2	0
^01.35 ...	2	0
1.35 ...	0	0
1 ...	0	0
1. ...	0	1
1.0 ...	0	2
1.10 ...	0	1

The third byte is used to keep track of the beginnings of conditional clauses. For example, the steps

1.10 Set x = 2.
1.21 Set z = x + y if y = a. ← 16th byte of stored string

would be represented in the parts list as follows:

PARTS:



KEEPING TRACK OF AVAILABLE SPACE

The Available Cell List (ACL) is a simple, right-linked cell list. The following processing notes are worthy of mention although they have no bearing whatsoever on the operation of the system:

1. Initially, ACL is ordered by increasing address numbers, the top cell (first on the list) having the lowest address number;
2. Cells are taken off the top as required and returned to the top when released;
3. Additional 512 cell units of space are ordered by increasing address number but are then slipped in atop the ACL.

The list header resides in a high-speed register during all processing, and is cached in the user's block on suspension of processing. Operations on ACL are independent of the contents of cells, other than

the necessary links; therefore, ACL does not need to be kept "clean."

Three cells of available space are held back from the user as an "ace in the hole" to be used only for typing, filing, and deleting objects (actions that must be possible for simply described objects at any time) and for all other operations on the disc files.

KEEPING TRACK OF ASSIGNMENTS (Table 7)

The objects that may be manipulated and assigned to letters are decimal values, logical values, arrays of values, and formulas. Each such assignment causes information to be posted in the assignment-table entry associated with the letter. The information consists of a descriptor of the object assigned and information concerning the "scope" of the assignment. Descriptors contain

1. The 8-bit code for the associated letter.
2. A numeric code identifying the kind of associated object.
3. A link, if necessary, to the first cell associated with the object's storage.

When an assignment is made to an iteration variable of a function, or to a formal parameter of a formula, the existing assignment must be saved. Thus, each assignment-table entry is treated as the top of a push-down list of entries, so that during the evaluation of iterative functions and formulas more than one entry in the assignment table may describe the same object in storage. Accordingly, each object has associated with it in storage a use count of the number of descriptors for that object that are extant in the assignment table.

In order to exercise some control over the scope of assignments to letters (in particular, to formal parameters), the level of formula nesting that attained at the moment of an assignment is also posted in the assignment table (i.e., the number of referrals to formulas within formulas that attained). In addition, space is reserved in each entry for the level of subprogram nesting associated with assignments, although such objects do not exist at present.

One hundred and four consecutive words of the user's block are reserved for the 52-cell assignment table. The first 26 cells are

Table 7

ASSIGNMENT TABLE ENTRIES

(1, 0-7) = 8-bit code for the associated letter

(1, 8-12) = not used

(1, 13) = 1 if a descriptor for a *sparse*[a] array
= 0 otherwise

(1, 14-17) = numeric identifying code for the object assigned

(1, 18-35) = link to first cell of object storage

(2, 0-17) = LEVEL when assignment was made

(2, 0-8) = reserved for level of subprogram nesting (BASE) that attained when assignment was made

(2, 9-17) = level of formula nesting that attained when assignment was made

(2, 18-35) = link to next descriptor on the entry's push-down list

---

[a]See p. 123.

associated with the upper-case letters, the rest with lower-case letters (each half is ordered lexicographically). Initially, all entries are posted as being unassigned.

KEEPING TRACK OF OPERANDS AND RESULTS (Table 8)

The Operand-Descriptor Push-Down List (DS) is a simple, right-linked cell list for keeping track of object descriptors used and generated by the CPU during processing. Some descriptors on this list are simply copies of assignment-table entries, and others describe objects (generated during processing) that may see only temporary service or may attain permanent status in the user's block. Generated objects can cause no irreversible changes in the user's block as long as their descriptors are recorded on the DS. By keeping descriptors for all generated objects on the DS, the business of cleaning up and backing off from errors does not involve much more than returning the cells of the DS and of any associated "scratch" objects to available space.

In addition to descriptors for assignable objects, the DS can hold descriptors for

1. The array identifier and associated index values for referring to components of arrays.
2. The list of values required for an iteration.
3. A combination of the above for describing *for* phrases.
4. The number of a required part, step, or form--or a description of a collection of objects (e.g., *all parts, all values,* and *all).*

Table 8

OPERAND-DESCRIPTOR PUSH-DOWN LIST (DS)

(1, 0-7) = 8-bit code for the associated letter if the descriptor was copied from the assignment table

= 0 otherwise

(1, 13) = 1 if a descriptor for a *sparse* array

= 0 otherwise

(1, 14-17) = numeric identifying code for object assigned

(1, 18-35) = link to first cell of object storage

(2, 0-17) = used at times to reorder a sequence of related descriptors on DS in a "first-in-first-out" manner; otherwise, not used

(2, 18-35) = link to next descriptor on DS

Some descriptors have no associated storage for objects; these include

1. Formal parameters of formulas to which assignments have been temporarily made during evaluation of the formula.
2. System-specific marks and words such as *size*, *time*, *users*, and underscores.

Descriptors for arrays, formulas, and functions are always identified; that is, they always carry the 8-bit code for the letter to which they are currently assigned. Descriptors for values resulting from arithmetic and logical calculations are always unidentified. Note that the length of the identification field can be increased from eight to twelve, accommodating expansions of the set of permissible identifiers.

Object Descriptors (Table 9)

The assignment of an actual parameter to a formal parameter prior to evaluation of a formula is kept track of by posting the appropriate (type 7) descriptor on the DS. Since the same information is effectively available in the assignment-table entry (via the assignment level recorded there), this is simply a device to save time when releasing such assignments by obviating searches through the assignment table.

"LHS" denotes the structures compiled from left-hand sides of assignment commands or *for* phrases: a letter and a set of index values.

"ROV" denotes the structures compiled from lists of values and ranges of values for iterations.

The structure for a *for* phrase is a list of two structures, an LHS and an ROV.

"OOD" denotes list structures compiled from such phrases as *all, all parts, all steps, all forms, all formulas, all values, part..., step..., form...,* and *formula....* The type of OOD is identified in (1, 10-13) by a numeric identifying code.

The choice of how much descriptive material to keep with the descriptor and how much to keep with the object itself is sometimes difficult, particularly when one moves away from general list structures. A simple scheme sufficed for JOSS: Scalar values are distinguished (as decimal or logical) by information carried in their descriptors; values identified by indexed letters are so distinguished by information carried with the values themselves (see Table 10). This scheme is readily extended to handle, say, complex values, or extended-range decimal values; beyond that, a fairly major reorganization would be required.

Table 9

OBJECT DESCRIPTORS

Object	(1, 14-17) Identifying Code	(1, 18-35) Link
Logical value	0	Link to item TRUE or item FALSE
Decimal value	1	Link to object cell
Array	2	Link to header cell for array
Formula	3	Link to header cell for formula
Function	4	Line number in system-function table
Iterative function	5	Line number in system-function table
Unassigned	6	Zero
Formal parameter	7	Link to cell in assignment table corresponding to the formal parameter
LHS	8	Link to header cell for LHS list
ROV	9	Link to header cell for ROV list
for phrase	10	Link to header cell for the list
OOD	11	Link to cell containing object's identification (see subtable below)
Special elements	12	0 indicates a string of underscores 1 indicates *size* 2 indicates *time* 3 indicates *users*

OOD	(1, 10-13)	(1, 18-35)
all	0	0
all parts	1	0
all steps	2	0
all forms	3	0
all formulas	4	0
all values	5	0
part	8	Link to cell for part number
step	9	Link to cell for step number
form	10	Link to cell for form number
formula	11	Link to cell containing assignment-table address

Storing Assignable Objects (Table 10)

The nine-digit, floating-decimal numbers of JOSS require one cell for their representation. The representation consists of a tripartite, binary encoding called the "JOSS Normalized Form" (JNF):

1. The sign of the number;
2. The digits of the number, sans exponent, normalized to nine digits with trailing zeroes;
3. The exponent part of the number.

The logical values, *true* and *false*, are represented by single copies of the JNF representations of 1 and 0, respectively, that are stored in the user's block and labeled TRUE and FALSE, respectively. All descriptors for logical values describe one of these copies.

Representations of the scalar elements of arrays are organized into list structures, ordered in an obvious manner. Vectors, for example, are represented by a right-linked list of elements, each labeled with its index; matrices are represented by a list of such lists, each list header carrying its row index. The restriction of indices to the range [-250, 250] is an implementation decision based mainly on the packing required to fit sign, digit part, exponent part, index, and link into a single cell. Components that have logical values are distinguished by a marked exponent of zero.

To speed up orderly searches through list structures for arrays, the list headers used in these structures contain information about the most recently referenced list elements. The routines for assigning and deleting array components are responsible for keeping this information up to date.

Table 10

REPRESENTATIONS OF ASSIGNABLE OBJECTS

Decimal values (JNF)

(1, 0-0) = binary representation of the sign
(1, 1-35) = binary representation of the nine-digit integer
(2, 0-17) = two's complement representation of the exponent part
(2, 18-35) = use count

Logical values

true = JNF 1
false = JNF 0
} one and only one copy of each

Arrays of dimension $n \leq 10$

(1, 0-17) = link to last referenced subarray of dimension $n - 1$
(1, 18-35) = link to first subarray of dimension $n - 1$
(2, 0-17) = n, as a binary integer
(2, 18-35) = use count

Subarrays of dimension $k > 0$

(1, 0-17) = link to last referenced subarray of dimension $k - 1$
(1, 18-35) = link to first subarray of dimension $k - 1$
(2, 0-8) = index of subarray, in two's complement representation
(2, 9-17) = not used
(2, 18-35) = link to next subarray of dimension k

Array components

(1, 0-35) = as for decimal numbers and logical values
(2, 0-8) = index of component, in two's complement representation
(2, 9-17) = two's complement representation of exponent part if decimal number; octal *400* if logical value
(2, 18-35) = link to next component

Formulas

(1, 0-17) = link to string of letters used as formal parameters
(1, 18-35) = link to literal string of characters of the formula
(2, 0-17) = number of formal parameters, as a binary integer
(2, 18-35) = use count

Functions (in CPU)

(1, 18-35) = address of evaluation routine
(2, 0-17) = number of arguments; -1 if an iterative function
(2, 18-35) = use count

Tables for all functions reside in the system's core store, rather than in the user's block. Assignments of such objects can be made only to formal parameters of formulas.

Storing Nonassignable Objects (Table 11)

Objects of types 7 through 12 are compiled or otherwise generated by the CPU during interpretation. They flare into existence, are used in further interpretation or execution, and are then released.

Each of these objects corresponds to a well-defined syntactic

Table 11

REPRESENTATIONS OF NONASSIGNABLE OBJECTS

Left-hand sides

(1, 18-35) = link to letter's assignment-table entry
(2, 0-17) = number of indices
(2, 18-35) = link to first index value

Index values in LHS structures

(1, 0-35) = index value, in two's complement representation
(2, 0-17) = not used
(2, 18-35) = link to next index value

Range-of-values list elements

(1, 0-0) = binary representation of the sign
(1, 1-35) = binary representation of the nine-digit "magnitude"
(2, 0-8) = octal *400* if number is a singleton or the right bound of a range of values; otherwise, zero
(2, 9-17) = two's complement representation of exponent part if a decimal number; octal *400* if a logical value
(2, 18-35) = link to next element

for and *number-of-times* phrases

(1, 0-17) = link to LHS list if *for*
= 0 if *number-of-times*
(1, 18-35) = link to ROV list if *for*
= link to cell for *number-of-times*

construct in JOSS's language, and each has an associated routine for interpreting such constructs and compiling the appropriate information.

KEEPING TRACK OF OPERATORS AND REENTRANT ROUTINES (Table 12)

The Operator-Descriptor Push-Down List (PS) is a simple, right-linked cell list for keeping track of operators, punctuation, and addresses of calling instructions for reentrant routines. In addition to descriptors for such elements, descriptors for arrays, formulas, and functions are occasionally cached on the PS during processing. The top descriptor on the list resides in the high-speed register CP; the link to the next descriptor's cell resides in the high-speed register PS. These are cached in the user's block on suspension of processing. Cleaning up and backing off require that the cells of the PS be released.

Most descriptors on the PS represent terminal characters of the language. These are organized into nine classes:

0. Letters.
1. Decimal numerals; *true; false;* function names; *$; timer.*
2. Left groupers (parentheses and brackets).
3. The absolute value bar.
4. Plus, minus, multiplication, division, exponentiation signs.
5. *not.*
6. *and, or.*
7. The six numerical relation signs.
8. Right groupers, punctuation marks, and special symbols.
9. Words in JOSS's vocabulary (except those used as logical operators) and strings of underscores.

Classes are divided into subclasses and, when a finer differentiation is appropriate, subclasses are divided into "types."

Descriptors for terminal characters are generated during interpretation (the routine P51 is used for recognizing the next terminal character in the string of characters under examination), and are (with the exception of descriptors for decimal numbers and strings of underscores) derived directly via table look-up, with an 8-bit character code as argument.

The descriptors (see Table 12) occupy a single word with format:

(1, 0-8) = subclass identification

(1, 9-17) = class identification

(1, 18-35) = type identification or other required information

Terminal characters of class 2 through 7 are treated as operators during interpretation. Associated with each such operator is a pair of numeric values that are used to govern the order in which adjoining operations are carried out when evaluating expressions. Differentiation among operators is provided by the type identification, which is used as a line number in various CPU tables--including the table of numeric-value pairs mentioned above.

Three internally generated operator descriptors may also be found on the PS:

unary plus: class = 4; subclass = 0; type = 17

unary minus: class = 4; subclass = 0; type = 18

"backstop": class = return address for reentrant routines; subclass = 0; type = 19

Table 12

TERMINAL CHARACTER DESCRIPTORS

Terminal Character	Class	Subclass	Type
Letter	0	0	Link to assignment-table entry
Decimal number	1	1	Link to object cell, CPU storage
Logical number	1	0	Link to item TRUE or item FALSE
Function	1	4	Line number, CPU function table
Iterative function	1	5	Line number, CPU function table
$ (dollar sign)	1	6	Link to item LINE
timer	1	7	Link to item UMIN
((open parenthesis)	2	0	0
[(open bracket)	2	0	1
| (absolute value bar)	3	0	2
+ (plus sign)	4	0	3
- (minus sign)	4	0	4
• (multiplication dot)	4	0	5
/ (division slash)	4	0	6
* (asterisk)	4	0	7
not	5	0	8
and	6	0	9
or	6	0	10
= (equals sign)	7	0	11
≠ (not equal to)	7	0	12
< (less than)	7	0	13

Table 12--continued

Terminal Character	Class	Subclass	Type
> (greater than)	7	0	14
≤ (less than/equal to)	7	0	15
≥ (greater than/equal to)	7	0	16
) (close parenthesis)	8	0	0
] (close bracket)	8	0	1
, (comma)	8	0	2
; (semicolon)	8	0	3
: (colon)	8	0	4
. (period)	8	0	5
? (question mark)	8	0	6
Carrier return	8	0	7
Page	8	0	7
End-of-string	8	0	8
' (single quote)	8	0	9
" (double quote)	8	0	10
Illegal code	8	0	12
Set	9	0	0
Let	9	0	1
Do	9	0	2
Type	9	0	3
Delete	9	0	4
Line	9	0	5
Page	9	0	6
Cancel	9	0	7

Table 12--continued

Terminal Character	Class	Subclass	Type
Go	9	0	8
To	9	0	9
Done	9	0	10
Stop	9	0	11
Demand	9	0	12
Form	9	0	13
Parenthetic Do	9	0	14
Parenthetic Cancel	9	0	15
Discard	9	0	16
File	9	0	17
Recall	9	0	18
Use	9	0	19
Quit	9	0	20
Reset	9	0	21
part	9	1	1
step	9	1	2
form	9	1	3
formula	9	1	4
file	9	1	6
item	9	1	7
parts	9	2	1
steps	9	2	2
forms	9	2	3
formulas	9	2	4
values	9	2	5

Table 12--continued

Terminal Character	Class	Subclass	Type
all	9	3	0
if	9	4	0
Special if	9	4	1
in	9	4	2
for	9	4	3
times	9	4	4
as	9	4	5
list	9	4	6
be	9	4	7
sparse	9	4	8
String of underscores	9	5	0
size	9	5	1
time	9	5	2
users	9	5	3

III. THE CENTRAL-PROCESSING ROUTINES

The general organization of the central-processing routines mirrors the principal responsibilities of the CPU: (1) communication and synchronization with the supervisory unit; (2) intercommand sequencing and control; and (3) command interpretation and execution. The central-processing routines fragment into several collections of routines and information that reflect this division of effort and responsibility:

1. The coroutines and major subroutines, X43 through X57, for
   a. Interfacing with the supervisory routines.
   b. Processing lines typed by the user.
   c. Composing and controlling the typing of lines.
   d. Interstep sequencing and control.
2. The major routines, V0 through V16 and D56 through D62, for controlling the interpretation of commands once the type of command has been determined.
3. The reentrant subroutines, P35 through P49, for controlling the interpretation of major syntactic constructs such as expressions for values.
4. The routines, E0 through E60, for error diagnosis, control, and commentary.
5. Three collections of ancillary subroutines:
   a. P51 through P74 for general processing.
   b. S50 through S74 for operations on literal strings of characters and for typing and deleting.

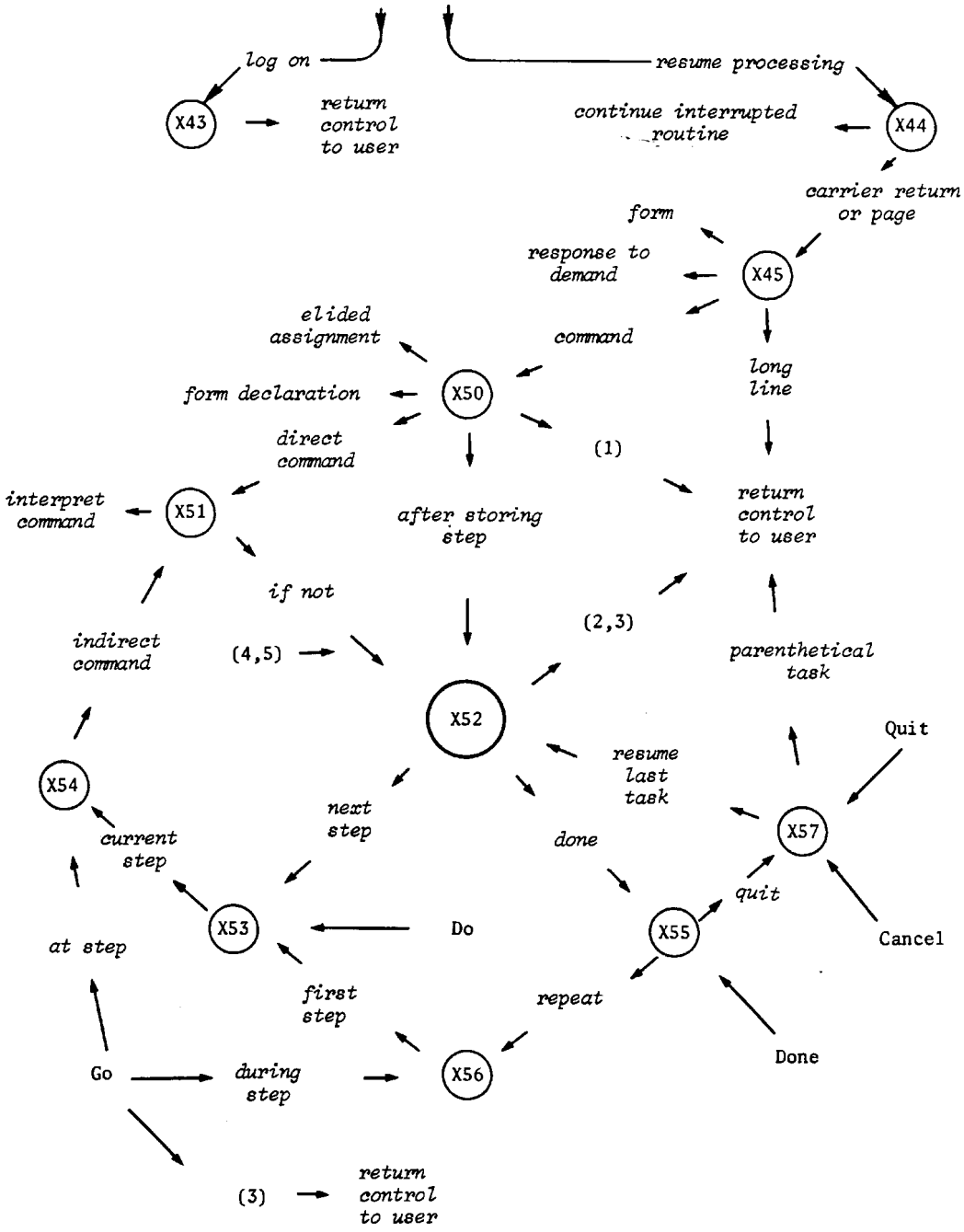
c. D50 through D54 for controlling the transmission of information to and from the magnetic-disc files.

6. A pool of fixed information, including controlling parameters, tables, strings of characters for messages, and machine-dependent extractors and codes.
7. A separate cache for volatile information that need not be saved in the user's block.

Routines will usually be described as active agents rather than as passive algorithms to be interpreted by the machine, and will always be referred to by their machine-language labels. Flowchart 1 sketches the organization and general flow of control, each labeled circle representing a routine or collection of routines.

Routine X43 sets information in the user's block to initial conditions after the log-on ritual, and then returns control to the user. Routine X44 controls all subsequent resumptions of processing. Cases in which processing had been suspended to await the typing of a line by the user are factored out for individual attention by routine X45; all other suspensions are resumed at the point of interruption. Routine X45 shunts lines that represent forms and responses to demands for input to the routines for interpreting the associated commands. Routine X50 processes all other lines typed by the user and all lines input from the magnetic-disc files. Form declarations and elided assignment commands are factored out for individual attention, and indirect steps are stored away in the user's block.

Routine X51 controls the interpretation of all steps other than form declarations and elided assignment commands. Conditional clauses



(1) Ignored line, transmission error.
(2) After direct commands and errors.
(3) Nothing to do.
(4) After error diagnosis and commentary.
(5) After interpretation of all commands except those indicated above.

Flowchart 1--Gross Flow

are interpreted before the command is ever examined. The type of command uniquely determines which of the *V* or *D* (disc commands) routines is to be used for interpreting the imperative clause. Routine X52, which is used for interstep sequencing and control (except after *Do*, *Go*, *Done*, *Quit*, and *Cancel* commands) is the central traffic controller, and is invoked whenever a step has been completed and whenever control is to be returned to the user: It cleans up the user's block and then turns control over to the user or to the routines for continuing the interpretation of a stored program.

Routines X53 and X54 pick out the next step of a stored program and initiate interpretation of the step after pausing to honor interruptions from the user and recalls from the SU.

Routine X55 controls repetitions of steps and parts that the user had told JOSS to do; routine X56 restarts the interpretation of steps and parts that JOSS had been told to do repeatedly; routine X57 cleans up after the completion of such tasks and controls the continuation of any suspended tasks.

Not shown in Flowchart 1 are

1. A manifold of short routines for interfacing with the supervisory routines on all suspensions of processing.
2. Most of the *V* and *D* routines for controlling the interpretation of individual types of commands.
3. Subroutine X48 for controlling the collection and typing of all output lines.
4. Subroutine X47 for detecting and honoring interruptions from the user and recalls from the SU.

5. Subroutine X46 for selecting the appropriate interface routine when processing the disc files.

PROCESSING LINES TYPED BY THE USER

Whenever the user has control of the typewriter, JOSS stands ready to accept and interpret the next line typed by him. Interpretation of the line will depend on what JOSS expects the user to type. Once past the log-on ritual, only three possibilities exist:

1. The user is going to type a full-line form to be stored away and made available for specifying formal output. In this case, the user should be able to observe or recall that the last instruction he gave was that JOSS stand ready to accept a form.
2. The user is going to type a response to a demand from JOSS that he input a value to be identified and stored away. This situation is triggered by a step of a stored program that caused JOSS to type the user-specified identification, followed by an equals sign, before returning control to the user.
3. The user is going to type one of the following:
   a. A command to be carried out directly.
   b. A labeled step to be saved as a step of a stored program.
   c. A blank line or a line with an asterisk (*) at either end; these constitute implied commands to ignore the line and return control to the user without comment.

In the last situation, JOSS has either typed a status, stopping, point-of-interruption, or error message before returning control to

the user, or has returned control without comment, for reasons JOSS assumes are apparent.

As long as the user has control of the typewriter, he can edit the line freely: backspacing, overstriking characters, and erasing characters by overstriking them with the space sign (#). Not until the user has hit the carrier-return or pagination key does JOSS take control of the typewriter and begin processing the line. JOSS responds to many input lines with dispatch. For example, if the line represents a form or an indirect step to be stored away, or represents a request for the assignment of a formula or a quickly computed value, JOSS carries out the action well before the typewriter carrier has returned to its rest point. JOSS responds just as promptly to requests to type out quickly computed values, or elements of the user's stored program and data.

While the user is typing the line, JOSS replaces all instances of the strike-out sign (#) by spaces. When the user releases control, JOSS examines the line for invalid characters, which may indicate errors in transmission or double-strikes. Not all errors can be detected, since errors may cancel and double-striking (or single-striking) may result in the transmission of a valid character different from the one typed on the page. JOSS responds to detected errors by retyping the line, with all invalid characters replaced by the strike-out sign (#).[†] JOSS then requests the user to retype the line. If expecting a response to a demand for input of a value, JOSS retypes the identification and equals sign before returning control.

[†]Primarily to expedite console repairs.

If the line is "error free," JOSS examines the line's length. If there are more than 78 characters in the line, including spaces, JOSS notifies the user that he has typed a line too long to be handled (excess characters having been lost in transmission) and requests him to retype the line.† If the line is acceptable, JOSS takes differential action depending on what is expected. The interpretation of forms and responses to demands for input will be described later. In all other cases, JOSS first determines whether the line is to be ignored. If so, JOSS returns control to the user without comment; otherwise, the line must represent either a direct command or a labeled step to be stored away. Step labels must be properly typed and positioned and must represent valid step numbers; that is,

1. Be preceded by spaces, if at all;
2. Be set off from the remainder of the line;
3. Be explicit decimal numerals containing no more than nine significant digits, exclusive of leading and trailing zeroes; and
4. Represent step numbers lying in the range $1 \leq$ *step number* $< 10^{10}$.

If the step label is improper, or if it is preceded by tabs, JOSS notifies the user and returns control to him. If valid, JOSS carries out the implied addition, insertion, or replacement and then returns control to the user. If there is not enough storage space available to make the change, JOSS notifies the user and then returns control, leaving the existing program intact. *JOSS makes no changes*

†In the case of commands, this should properly be done after it has been determined whether the line is to be ignored, rather than before.

in the user's program or data until certain that enough space is available. If space is available, JOSS makes the change without examining the remainder of the line. Thus, an indirect step may be complete gibberish as a command. JOSS does not know until the step is encountered while carrying out a stored program for the user.

If the line is not prefixed by a step label, it must represent a direct command, and JOSS takes differential action depending on whether the command begins with

1. The word *Form*, which indicates that the user is identifying a form for formal output to be typed on the next line.
2. A single letter, which may indicate that the user has typed an elided assignment command.

The interpretation of these two cases is direct and will be described later. In all other cases, the line must represent a direct step to be interpreted in exactly the same manner as an indirect one.

The Input/Output Unit (IOU) collects the 7-bit representation of the typewritten line in one of the core-buffer areas reserved for this purpose, together with information indicating the source of the line (Teletype or JOSS console) and whether or not the line was too long to fit in a fixed-length buffer area. Because Teletype consoles have narrower pages than JOSS consoles and type only upper-case letters,† the CPU records the type of console in the user's block for subsequent

†A slight change in both the IOU and the CPU would allow Model 37 TTY's, which type in upper- and lower-case, to be used. However, the IOU would have to be able to differentiate among the different types of consoles.

use in processing the current line and in regulating the width of lines that it may output.

The CPU notes parenthetical forms of the *Do* and *Cancel* commands by stripping off the parentheses and replacing the 8-bit token for the verb with a unique 8-bit surrogate denoting the parenthetical form.

INTERPRETING COMMANDS

Except for elided assignment commands and form declarations, all commands take the form of conventional English sentences (and may, in fact, be read aloud): a capitalized imperative clause followed optionally by a conditional clause and terminated by a period.

JOSS first examines conditional clauses, which are formed by appending a valid expression for a logical value to the word *if*. *JOSS examines the imperative clause only if the conditional clause is valid and the condition holds*. If no conditional clause exists or if the condition holds, JOSS looks at the beginning of the command for an imperative clause, which must begin with a capitalized verb. Further interpretation is based uniquely on the action specified by the verb.

Reporting Errors

Once committed to the interpretation of any part of a command, JOSS insists on the use of conventional rules for capitalization, spacing, punctuation, and spelling. In particular, commands must begin with a capitalized verb and end with a period; words and numerals may neither be broken by spaces nor run together. With few exceptions, spaces can be used freely or, where unnecessary, not at all.

Any error causes JOSS to abort interpretation immediately and return control to the user after commenting on the error. Whenever the interpretation of a command is aborted for any reason (including interruptions by the user), JOSS makes sure that all information stored away for the user is precisely as it was when JOSS began interpreting the command (including all steps, forms, assignments, and information about ongoing tasks being done for the user). This allows the user to take corrective action, and then have JOSS continue from the point of error.† An exception to this rule occurs whenever JOSS is preparing to repeat a part or step that the user had asked JOSS to carry out repeatedly. At such times, JOSS may run out of storage space or may be unable to find the required step or part. Whenever this occurs, JOSS makes sure that all information is as it was when JOSS finished carrying out the most recent action, before returning control to the user. JOSS tries to identify both the error and the point of error for the user. The point of error is usually made clear as follows:

1. If the error occurred while interpreting the last line typed by the user, JOSS simply comments on the error.
2. If the error occurred while interpreting a step of a stored program, JOSS describes the error as having occurred *at step....*
3. If the error occurred while trying to repeat a step or part, JOSS describes the error as having occurred

†See p. 143.

a. *during above*, if the task was initiated by a direct command;

b. *during step...*, if the task was initiated by an indirect step.

4. If the error occurred while JOSS had dropped one or more levels to evaluate a formula, JOSS further identifies the error as having occurred *in formula....*

JOSS's behavior is modified whenever a transmission between magnetic-disc storage and core storage† is aborted because of detected hardware malfunctions or errors that could only be caused by hardware malfunctions. In such cases, it is not feasible (and usually impossible) to prevent irreversible changes in either the user's files or in his block of information. (This is particularly true during recalls of items from the files into the user's block.) Despite snapping back to the beginning of such commands, JOSS reports the errors as having occurred *during* the command, as a warning to the user.

Error messages†† themselves are of two types: (1) those that report linguistic violations, references to undefined objects, and other explicit violations of form; and (2) those that report situations for which a precise and unambiguous error message cannot be composed. The first type is infrequent, and the error messages are explicit. Errors of the second type are reported by the brief message *Eh?*, because they are often easier for the human eye to detect than for JOSS to unravel.

---

†See p. 153.

††See p. 159.

Implementation Notes

The formal rules for composing commands factor into

1. Rules for forming terminal characters: words, numerals, and other permissible strings of atomic characters.
2. Rules for forming, from the above, expressions for individual objects, collections of objects, ranges of values for iterations, etc.
3. Rules for forming commands from the above.

Deriving naturally from this classification are the routines for interpreting commands:

1. The subroutine P51 for advancing to and "recognizing" the next atomic or terminal character in the command being interpreted.
2. The reentrant routines and major subroutines, P35 through P49, for handling complex expressions.
3. The *V* and *D* coroutines for controlling the interpretation of commands once the type of command has been determined.

An extra level of interpretation controlled by a tabular representation of the rules is neither required nor appropriate. Instead, the routine associated with a type of command "reads" like the rules for forming instances of the command, and is composed (in the main) of direct examinations of primitive entities (via P51) mixed with uses of the routines for interpreting more complex expressions. For example, V1, the routine for interpreting assignment statements, can be described metaphorically:

1. Use routine P40 to determine whether the command begins with an acceptable left-hand side of an assignment statement; if so, compile the appropriate information.
2. Using routine P51, determine whether an equals sign comes next.
3. Use routine P49 to determine whether the right-hand side is an acceptable expression for a value; if so, compile the appropriate information.
4. Using routine P51, determine whether the right-hand side is properly terminated.
5. Carry out the assignment.

The "looser" or more "general" the grammar, the simpler the parsing routines; thus, a relaxation of JOSS's rules would result in some simplification. For example, it would be a straightforward task to erase the boundary conditions on many of the routines and simplify them so that a single routine could be used for interpreting expressions for a general object or list of objects in any context. Such modifications would be justified were it felt that the resulting generality would be more important than clarity of exposition and consistency of behavior. Modification might even be necessary if JOSS (or a JOSS-like system) were to be produced on a machine with austere operational capabilities or with little core storage. On the other hand, simplification itself does not seem to be a good reason for relaxing all rules of grammar and behavior. Many of JOSS's rules are dictated by convention, others by personal taste; they will be described later with the appropriate routines.

To such rules is added a modus operandi for dealing with error commentary. It was felt that the commentary should not be misleading--that error messages should be appropriate and clear. This meant giving attention to the order in which errors were detected and commented on. For example, the command *Do form 3.* is improper because only steps and parts can be done by JOSS. If *form 3* had not been defined, it would be inappropriate to notify the user of that fact before mentioning the impropriety of the statement. In JOSS's language, as in others, a pecking order of error detection and commentary must be established--in part natural, in part arbitrary. Such pecking orders can be maintained in two ways:

1. By ordering searches for alternative instances of a syntactic construct (and this order will vary with context).
2. By performing a complete analysis of a command each time it is interpreted and recording all errors.

The latter tack reduces processing speeds and is not feasible for interpretive systems such as JOSS. The former, which is the one used in the central-processing routines, is implemented by factoring the general notion of "operand" into classes of operands appropriate to different contexts and using a different routine for each class; for example, operands suitable for computation, for execution, for deletion, for display, for assignment to formal parameters of formulas, and for other distinct purposes.

Instances of expressions for such operands may be considered as prescriptions for computing values and compiling information, while the associated routines may be considered as active agents for carrying

out the prescriptions. From another point of view, these routines may be looked at as mechanisms for answering the question, "Is the upcoming string of characters an instance of the associated syntactic construct?"--evaluation and compilation being carried out in passing. Few of these routines are so used; that is, failures are rarely "reported." Instead, one of the *E* routines is used for independent fielding of errors, and for cleaning up and backing off if required. This strategy is appropriate, because the rules for forming commands provide the user with few real alternatives except at the most elementary level involving letters, numerals, operators, and groupers. In almost all cases, the upcoming syntactic construct is uniquely determined by what has preceded.† Metaphorically speaking, once a commitment to a specific syntactic construct has been made, there exist no alternative avenues of escape in the event of errors.

EXPRESSING VALUES

From the primary nature of JOSS as a computational assistant, it follows that expressions for values predominate in most programs. Expressions for decimal values are used for

1. Identifying steps, parts, forms, files, and items of files.
2. Assigning values to letters and indexed letters for identification purposes.
3. Assigning values to formal parameters of formulas for evaluation purposes.

†The sole exception arises because of the two alternative forms associated with such functions as *sum* and *prod*; this case is handled simply by a few machine-language instructions in the appropriate routine (P36).

4. Expressing index values.
5. Expressing initial and final values and increments for iteration purposes.
6. Expressing values to be typed.

Expressions for logical values are used for

1. Assigning values to letters, indexed letters, and formal parameters of formulas.
2. Expressing values to be typed.
3. Expressing values for conditionally controlling the interpretation of imperative clauses (in conditional clauses).
4. Expressing values for controlling the differential choice of expressions for values (in conditional expressions for values).
5. Choosing the first value in a set or range of values for which a condition is satisfied (with the function *first*).

The simplest expressions for values are

1. Decimal numerals, which may contain up to nine significant digits, exclusive of leading and trailing zeroes, and which must represent numbers less than 10^{100} in magnitude.
2. The logical "numerals" *true* and *false*, which carry the corresponding logical values.
3. The dollar sign ($), which carries a decimal value (from 1 through 54) equal to the line number at which the user's typewriter carrier is positioned.
4. The word *timer*, which carries a decimal value equal to the

number of minutes† that have elapsed since the user logged on or last reset the value to zero.

A single, general rule governs the formation of expressions for values in JOSS's language: *Wherever a decimal (logical) numeral is allowed in a command, an arbitrarily complex expression for a decimal (logical) value may be used; there are no exceptions to this rule.*

Direct Expressions

Expressions for values are formed from numerals, $, and *timer* by repeated application, in any order, of a set of conventional rules for forming linearized expressions.

1. Values can be assigned to letters and indexed letters, and arbitrarily complex expressions for the letters and indexed letters can then be used to refer to these values for purposes of computation.
2. Decimal values can be
   a. Added (plus sign),
   b. Subtracted (minus sign),
   c. Multiplied (centered dot),
   d. Divided (slash),
   e. Raised to a power (asterisk), and
   f. Compared under any of the six numerical relations

      $=, \neq, <, \leq, >, \geq$
3. Expressions for extended numerical relations (e.g., $a < b \leq c$)

†Rounded to the nearest 100th minute.

are permitted and are interpreted in the conventional manner.

4. The logical operations of negation (*not*), conjunction (*and*), and disjunction (*or*) can be applied to logical values.
5. A set of elementary decimal-valued and logical-valued functions can be applied to one or more expressions for values as arguments, and a subset of these can be applied to a range of values defined iteratively.†
6. Expressions for values can be grouped by using parentheses, brackets, or absolute-value bars in matching pairs.
7. Traditionally numerical operations can be applied to decimal values only, and traditionally logical operations to logical values only. This rule is relaxed to allow logical values and "mixed" values to be compared for equality or inequality, mixed values always being unequal.

JOSS interprets expressions from left to right, honoring groupings as they occur. That is:

1. Whenever encountering a properly used left grouper, JOSS evaluates the upcoming grouped expression before doing anything else.
2. Whenever encountering a properly used identifier for a function, formula, or array, JOSS evaluates any upcoming list of arguments, parameters, or index values and then evaluates the function or formula or searches for the array component before doing anything else.

---

†See pp. 84-85.

In the absence of groupings:

1. Addition and subtraction associate from the left:

$$2 + 2 - 3 + 2 = ((2 + 2) - 3) + 2 = 3$$

2. Multiplication and division associate from the left:

$$2/2 \cdot 3/2 = ((2/2) \cdot 3)/2 = 3/2$$

3. Exponentiation associates from the left:[†]

$$2*2*3 = (2*2)*3 = 64 ;$$

4. Addition and subtraction are deferred in favor of multiplication and division, and all operations are deferred in favor of exponentiation:

$$2 + 2 \cdot 3*2 - 6 = 2 + (2 \cdot (3*2)) - 6 = 14 ,$$

5. Plus and minus signs used as unary operators are treated as though they had been prefixed by a zero and used as binary operators (but $2/-3$, for example, is illegal):

$$-2*2*3 = 0 - 2*2*3 = -64$$

6. Logical operations are deferred pending the evaluation of the associated logical operands, and relations are deferred pending the evaluation of the associated decimal operands:

$$x + 1 < y/z \quad \text{or} \quad x + 1 = w$$

is interpreted as

$$((x + 1) < (y/z)) \quad \text{or} \quad ((x + 1) = w)$$

7. Disjunction is deferred in favor of conjunction, and both are deferred in favor of negation:

[†]It is more customary to associate exponentiation from the right, probably because $(2*2)*3$, for example, is more efficiently carried out when written as $2*(2 \cdot 3)$.

not true or *false and false*

is interpreted as

(not true) or *(false and false)*

Extended relations are treated in the conventional manner; for example,

$$a + b < c \leq d$$

is interpreted as

$$((a + b) < c) \quad \text{and} \quad (c \leq d)$$

JOSS carries decimal numbers in scientific notation: nine significant digits and a base-ten scale factor with exponent in the range -99 through +99. Addition, subtraction, multiplication, division, and square root yield true results rounded to nine digits. Zero is substituted on underflows, while overflows cause an error message.

Conditional Expressions

Conditional expressions are used to choose one of a set of expressions for values differentially on the basis of a set of associated conditions, and may be used wherever an expression for a value is permitted. Expressions for logical values are used as conditions, which precede and are set off from their associated expressions by colons. Pairs of associates are separated by semicolons, and the entire set of pairs must be enclosed in parentheses or brackets.

JOSS evaluates conditional expressions from left to right, as they occur. If the first condition holds, JOSS evaluates the associated

expression and then skips over the remainder of the conditional expression. If the first condition does not hold, JOSS skips over the associated expression to examine the next condition, and thus continues until a condition holds or until assured that none of the conditions hold. In the latter case, JOSS looks for a final expression (not associated with any condition) to be chosen in default. If none exists, the conditional expression is in error. For example, if $x = 1$:

$$(x < 1: 5;\ x > 1: 10;\ x = 1: 20) = 20$$

$$(x < 1: 5;\ x > 1: 10;\ 20) = 20$$

$$(x < 1: 5;\ x > 1: 10) = ???$$

If a conditional expression is naturally grouped by virtue of being used to express the argument of a function or a formula, or the index value of a vector, an extra set of groupers is not required.

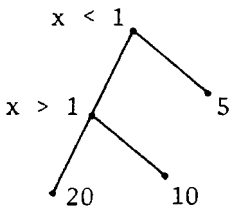
Both the facility and the LISP-like[†] format for conditional expressions were dictated by a desire to get as much mileage as possible out of JOSS's line-at-a-time style. A more general facility is provided by many programming languages through constructions exemplified by

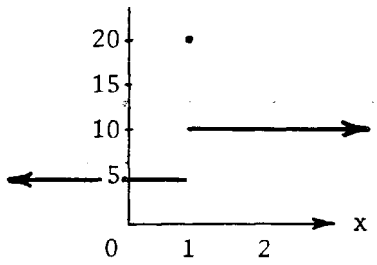
$$\text{if } x < 1 \text{ then } 5 \text{ else if } x > 1 \text{ then } 10 \text{ else } 20$$

This mode of expression seems appropriate if the conditions can be used to control the execution of statements or groups of statements. However, the style seems unnecessarily wordy, and even confusing, when used solely for choosing expressions--a comment that is completely independent of the fact that little could be said in a single JOSS

[†]John McCarthy et al., *LISP 1.5 Programmer's Manual*, MIT Computation Center and Research Laboratory of Electronics, 1962.

line. It is this author's opinion that such conditional choices for values are viewed, in the mind's eye, as spatial entities that can be grasped in toto, rather than as overly qualified, strung-out sentences in which the parts obscure the whole. For example:





In any event, the facility is useful enough to warrant incorporation. The colon separating the condition from its associated expression seemed appropriate, connoting (as it usually does) a strong connection or association between what precedes and what follows. The semicolon was adopted in favor of the more immediately appealing comma because it seemed to stand out better, particularly when comma-separated lists of arguments and index values were used.

Another compact expression for differential choices can be obtained by permitting traditionally numerical operations to be applied to logical values and conversely. For numeric computations, logical values are treated as a one if true and as a zero if false; for logical operations, numeric values are treated as false if zero and as true if not zero. The above example can then be written

$$(x < 1)\cdot 5 + (x > 1)\cdot 10 + (x = 1)\cdot 20$$

The JOSS style is less compact for some simple, binary choices, but more compact when the final condition is omitted.

Despite the scheme's first-blush appeal to "completeness" and its persistent vogue (since, at least, 1959), it has little merit and (in JOSS) less utility. Several objections can be raised, some rather obvious and others less so.

1. The meaning of an expression (the intent of the programmer to make a differential choice) can easily be obscured in a welter of arithmetic operations. This fact may be of little concern if the program is designed to be read only by a machine. However, humans do read programs, particularly their own.

2. Even when a particular expression is clear, the fact that it expresses a differential choice may not spring easily into focus under a quick scan of the program.

3. Expressing differential choices always involves a rerepresentation of a single entity (a graph, a small decision tree, a table) as a sequence of primitive arithmetic operations; that is, the user must always make a translation.

4. Even when subjected to the most exacting scrutiny by a carefully designed interpreting or compiling mechanism, the eventual interpretation of such expressions will be wasteful of processing time. In brief, the scheme replaces punctuation by operations, and implementing operations is always more costly than determining punctuation, even when the operations are never carried out or are replaced by equivalent, but faster, operations.

5. Errors in preparing and transmitting a program to the computer can result in valid expressions that differ markedly from the desired ones. Such inadvertencies can be guarded against by careful design of

both the keyboard and the encoding of information. For example, four of the six numerical relation signs on the JOSS keyboard are in upper-case above the numerals, and all are well separated from the symbols for the arithmetic operations; on other more widely used keyboards, the symbol for multiplication is immediately adjacent to one of the numerical relation signs, and in the same case.

Expressions that involve mixed arithmetic can be spruced up to a certain extent by demanding that the user be more explicit. For example, JOSS and some other current systems do not allow mixed arithmetic, but allow the user to signal the conversion of a logical value to its decimal equivalent by enclosing the expression for the logical value within absolute value bars. The expression in the example therefore becomes

$$|x < 1| \cdot 5 + |x > 1| \cdot 10 + |x = 1| \cdot 20$$

This mode of expression is a bit less error-prone and somewhat more readable than the usual one, but it still results in inefficient processing.

Elementary Functions

Sixteen functions are available to the user. Arbitrarily complex expressions can be used for the argument(s). Unless stated otherwise, all arguments and functions are decimal-valued. The arguments must be enclosed in parentheses or brackets, and the left parenthesis or bracket must follow the function identifier immediately, without intervening spaces (a compromise in favor of readability and standard practice).

Square root: *sqrt( )*

The argument must have a nonnegative value.

Natural logarithm: *log( )*

The argument must be positive.

Exponential: *exp( )*

Underflows ($e^x \leq 10^{-99}$) result in zero; overflows ($e^x \geq 10^{100}$) cause an error condition.

Sine and cosine: *sin( )* *cos( )*

The argument is assumed to be in radian measure and must have magnitude < 100.

Argument: *arg(,)*

The function requires two arguments separated by a comma and as a result gives the angle between the positive x-axis of the (x, y) plane and the line joining the point $(0, 0)$ and a point (x, y) defined by the two arguments. The result is in radian measure in the range $-\pi < arg(,) \leq \pi$. By definition, $arg(0, 0)$ yields zero.

Translating decimal values into logical equivalents: *tv( )*

If the value of the argument is zero, the result is *false*; if not zero, the result is *true*.

Translating logical values into decimal equivalents: *tv( )*

If the argument is *true*, the result is +1; if *false*, 0.

Signum: *sgn( )*

The value of signum is +1 for an argument greater than zero, 0 for an argument equal to zero, and -1 for an argument less than zero.

Integer part and fraction part: *ip( )* *fp( )*

A nonzero integer or fraction part of a number carries the sign of the number.

Digit part: *dp( )*

The digit part of a nonzero number is the number normalized to lie in the interval $1 \leq$ *magnitude of the digit part* < 10, and carries the sign of the number.

Exponent part: *xp( )*

The exponent part of a number is the power of ten which, when multiplied by the digit part, yields the number itself:

ip(-12345.678) = -12345

fp(-12345.678) = -.678

dp(-12345.678) = -1.2345678

xp(-12345.678) = 4

Absolute value: | |

If the argument is decimal-valued, the result is the magnitude of the argument. If the argument is logical-valued, the result is the decimal equivalent of the argument (+1 if *true*, 0 if *false*).

Sum, product, maximum, and minimum of a set of decimal values: *sum* *prod* *max* *min*

The set of values may be represented by a list of expressions separated by commas, or by an expression to be evaluated iteratively over a range of values of a variable. In the latter case, the variable of iteration and the range of values are separated from the expression by a colon:

```
Type sum[i = 1(1)6: i].
sum[i = 1(1)6: i] =  21

Type sum(1, 2, 3, 4, 5, 6).
sum(1, 2, 3, 4, 5, 6) = 21
```

Both letters and indexed letters can be used as variables of iteration. Existing assignments to such letters are saved over the course of the evaluation and restored when the function has been calculated. Thus, variables of iteration for iterative functions are treated as dummy variables. Arbitrarily complex expressions can be used for the expression that is to be evaluated and for the decimal values and ranges of values of the iteration variable. For example:

```
Type sum[i = 1(1)3: sum(i = 1(1)6: i)].
sum[i = 1(1)3: sum(i = 1(1)6: i)] = 63
```

First value of a variable of iteration for which a condition holds:

```
Type first[i = 1(1)6: i/3 = 1].
first[i = 1(1)6: i/3 = 1] = 3
```

If the condition is not satisfied for any value of the iteration variable, an error results:

```
Type first[i = 1(1)6: i/7 = 1].
first[i = 1(1)6: i/7 = 1] = ???
```

Conjunction, disjunction of a set of logical values:

conj disj

Conjunction and disjunction admit the dual forms described under *sum*, *prod*, etc. For example:

```
i = 2
k = 4

Type conj[j = 1(1)6: i ≤ j ≤ k].
conj[j = 1(1)6: i ≤ j ≤ k] =     false

Type disj[j = 1(1)6: i ≤ j ≤ k].
disj[j = 1(1)6: i ≤ j ≤ k] =      true

Type conj[i ≤ k, i = 2].
conj[i ≤ k, i = 2] =      true
```

ASSIGNING VALUES (Set)

Decimal values and the logical values *true* and *false* may be assigned to any of the 52 upper- and lower-case letters admitted as identifiers. Values may be organized into arrays by using indexed letters. Letters and indexed letters may then be used to refer to their assigned values for purposes of computation, typing, deleting, and filing in long-term storage. In addition, the letters themselves may be used for purposes of typing, deleting, and filing entire arrays.

JOSS interprets the entire imperative clause, from left to right, before considering the assignment itself. Any error encountered during this interpretation will cause JOSS to abort interpretation of the command after commenting on the error.

The left-hand side must be either a single letter or a valid expression for an indexed letter: a single letter followed immediately by a properly grouped and comma-separated list of valid expressions for index values. No more than 10 indices are allowed, and the index values must be integers not exceeding 250 in magnitude.

The right-hand side can be any expression for a decimal or logical value, and must be separated from the left-hand side by an equals sign.

Initially, after the log-on ritual, all letters are unassigned. Once made, an assignment stands until rescinded by another assignment or deleted by the user. JOSS always deletes the current assignment before assigning a scalar value to a letter. Assignments to indexed letters are treated as additions, insertions, or replacements to an existing array if the letter used already identifies an array whose dimensions match that of the assignee. Otherwise, the existing assignment is deleted before the new one is made. JOSS makes no assignment until certain that there is enough storage space. If not, JOSS notifies the user and returns control to him, leaving all existing assignments intact.

```
Type x.
x = ???
Set x = 10.
Type x.
          x =        10
Set x = x + 1.
Type x.
          x =        11
Set a(1) = 1.
Set a(2) = a(1) + 1.
Set a(a(1) + 3) = 3.
Type a.
       a(1) =         1
       a(2) =         2
       a(4) =         3
Set a(1, 1) = 1.
Type a.
    a(1, 1) =         1
```

An elided form of the command sans the word *Set* can be used, but only as a direct command. Conditional clauses may not be appended to the elided form, while the trailing period is optional (a concession to some users who became compulsive about periods). For example:

```
a = 1
Type a.
                a =         1
a = 2.
Type a.
                a =         2
a = 1 if a ≥ 0
Eh?
```

JOSS rejects assignment commands whose right-hand sides contain equals signs used in relations whenever the resulting command might be confused with so-called extended assignment statements; for example:

```
Set x = a = 1.
Please use parens or brackets to set-off ambiguous equals signs.
Set x = (a = 1).
Type x.
            x =      true
```

Extended assignment statements are usually interpreted as requests for the simultaneous assignment of a value to a set of letters or indexed letters. Because no explicit symbol for assignment is available, the usual notation (e.g., *Set* $x = y = 0$.) would have to be replaced by a less ambiguous one (e.g., *Set* $x, y = 0$.) to protect users from inadvertently causing JOSS to interpret an extended assignment as a single assignment of a logical value and conversely. In either case, the implementation of "simultaneous" assignments is tedious, because assignments cannot be made simultaneously. For example, because no irreversible changes are made in the user's block until there is enough space to make the change, JOSS would have to make sure that enough space was available for all the assignments before making any. This is a simple process only if done sloppily--without, in effect, simulating the entire set of assignments to determine the exact amount of

space available at each successive step of the process. (For example, the assignment of a scalar value to an array identifier could cause available space to expand.) More important, some assignments could be so blatantly contradictory as to warrant an error message (e.g., *Set A, A(2) = ...*). It was felt that simultaneous assignment commands were not worth the effort required to implement them properly, and they were discarded.

ASSIGNING FORMULAS (Let)

A value assignment command causes JOSS to evaluate the right-hand side of the expression and assign the resulting value to the specified letter or indexed letter. A formula assignment command causes JOSS to assign the literal string of characters constituting the right-hand side to the specified letter. Arbitrarily complex expressions for values may be assigned to letters, with or without formal parameters. The letters and expressions for actual parameters, in conventional functional notation, may then be used wherever an expression is valid.

```
Let f(x) = sqrt(x) + x.
Type f(2 + 2).
    f(2 + 2) =          6

Type f[f(4)].
     f[f(4)] =          8.44948974

Let h = f(4) + f(9).
Type h.
           h =         18
```

No more than ten formal parameters are allowed, and these must be identified by distinct letters. The grouped list of formal parameters must follow the letter of assignment with no intervening spaces.

JOSS strips leading spaces off expressions before storing them away and will not accept void expressions. If not enough storage space is available, JOSS notifies the user and then returns control to him, leaving all existing assignments intact. Otherwise, JOSS deletes the current assignment to the formula's identifier before making the new assignment.

Before computing the value of a formula, JOSS first evaluates the values of the actual parameters from left to right. If no errors are encountered, *JOSS, in effect, replaces all occurrences of formal parameters in a copy of the formula by the simplest expressions for the actual parameter values (e.g., single letters, decimal or logical numerals), and then interprets the resulting copy.* Thus, existing assignments to letters used as formal parameters remain intact during the interpretation of the formula; that is, formal parameters are "dummy" variables.

```
Let f(x) = sqrt(x) + x.
Set x = 10.
Type f(4).
      f(4) =        6
Type x.
       x =       10
```

Single letters may be used to refer to formulas for purposes of typing, deleting, and filing, except for typing the definition of a formula with no formal parameters. JOSS interprets such references as references to the values of the formula. In order to type such abbreviations, the user must refer to them explicitly as a formula; for example:

```
Let h = (b - a)/n.
Let f(x, y) = 3•x + y/x.
b = 1
a = 0
n = 100
Type h, formula h.
          h =          .01
          h:  (b - a)/n
Type f(3, 1), f.
      f(3, 1) =        9.33333333
      f(x, y):  3•x + y/x
```

Just as conditional expressions and iterative functions allow the user to write concise expressions for complex procedures, so formulas allow him to parameterize such expressions. The user can often use a few short formulas in place of a stored program, which, since JOSS allows recursive formula definitions,† can be a branching program. This ability can be a real boon to the user, particularly when he is interested in quick, unformatted output and knows his data well enough to forgo the extensive validations and checks that might go into a general, stored program.

For example, the amount, A_n, of a loan left outstanding after n periodic payments is

$$A_0 = \text{the original principal, } P,$$

$$A_n = A_{n-1} + IA_{n-1} - p,$$

where I denotes the periodic interest rate and p denotes the periodic payments. The corresponding formula, $A(n)$, can be defined as a conditional expression:

---

†Formulas that use themselves.

```
Let A(n) = [n = 0: P; A(n - 1)•(1 + I) - p].
P = 10000
I = .065
p = 1000
Type A(10).
      A(10) =      5276.95212
```

The last value typed by JOSS gives the approximate amount left after ten yearly payments of $1000 on a loan of $10,000 at an annual interest rate of 6.5 percent. The rounding required to do "penny" arithmetic could be incorporated into the formula, if desired, by using expressions involving the integral and fractional parts of *A(n)*.

The loan is paid off when the amount drops to zero (or less, because of the lack of precision). The user can, therefore, request JOSS to calculate the amount left at the end of each year and type the number of payments required to amortize the loan:

```
Type first[i = 1(1)25: A(i) ≤ 0].
first[i = 1(1)25: A(i) ≤ 0] = 17
Type A(16), A(17).
      A(16) =      636.09641
      A(17) =     -322.557323
```

The loan will be amortized sometime during the seventeenth year. In evaluating the *first* function, JOSS calculates *A(1)*, *A(2)*, and so on until the first value that drops to zero or less is found. The upper limit of 25 is simply tossed in as a reasonable guess and could be made larger or smaller (within reason). To see how long it would take to amortize the loan with payments of $1500, change *p*:

```
p = 1500
Type first[i = 1(1)25: A(i) ≤ 0].
first[i = 1(1)25: A(i) ≤ 0] = 10
```

A general formula to compute *N(p)*, the number of payments of p dollars required to amortize the loan, can be written

```
Let N(p) = first[i = 1(1)25: A(i) ≤ 0].
Type N(1000), N(1500).
     N(1000) =       10
     N(1500) =       10
```

Although the formula is formally correct, it gives the same answers for all values assigned to p. The reason for the error lies in JOSS's treatment of actual parameters. JOSS replaces occurrences of formal parameters by actual parameters in the formula itself, but not in formulas that are used by the initial formula.† To achieve the desired result, p must be used as a formal parameter of both formulas:

```
Let N(p) = first[i = 1(1)25: A(i, p) ≤ 0].
Let A(n, p) = [n = 0: P; A(n - 1, p)•(1 + I) - p].
Type N(1000), N(1500).
     N(1000) =       17
     N(1500) =       10
```

To get values correct to the penny, amounts must be multiplied by 100, augmented by ½, chopped off, and then divided by 100:

```
Let a(n, p) = ip[100•[A(n - 1, p)•(1 + I)] + .5]/100.
Let A(n, p) = [n = 0: P; a(n, p) - p].
Type N(1000).
     N(1000) =       17
Type A(16, 1000).
 A(16, 1000) =      636.11
```

As expressed above, *A(n)* is an example of a function of a single index of recursion, n; its value for a specific index is expressed solely in terms of its values for preceding indices and of fixed parameters and

†See p. 109 for a more detailed treatment.

initial conditions. Such simple recursive functions can often be written in closed form as functions of n alone, and can always be evaluated by a programmed computation carried out repeatedly for successive values of an index of iteration. For example,

```
Let A(n) = [(1 + I)*n]•P - p•[(1 + I)*n - 1]/I.
```

defines the function in closed form, while

```
1.1 Set A = P.
1.2 Set i = 1.
1.3 Set A = (1 + I)•A - p.
1.4 Set i = i + 1.
1.5 To step 1.3 if i ≤ n.
```

is a program that computes the value A for a given n by iteration. Similarly, the factorial function can be defined recursively:

```
Let f(n) = [n = 0: 1; n•f(n - 1)].
```

or iteratively:

```
Let f(n) = [n = 0: 1; prod[i = 1(1)n: i]].
```

Although recursive formulations are shorter, neater, and frequently (as in the example) closer at hand, they engender greater demands on storage space (during evaluation) and lengthier computations than do iterative formulations. On the other hand, some computations that are palpably iterative in nature can often be expressed and evaluated more efficiently as recursive functions. For example, given an approximation r to a solution of an equation $f(x) = 0$, Newton's method for refining the approximation is described by

```
Set r = r - f(r)/g(r).
```

where g represents the derivative of the function f. Thus, having assigned some initial guess to r,

```
1.1 Set r = r - f(r)/g(r).
1.2 To step 1.1 if |f(r)| > 10*(-p).
1.3  ...
```

would repeatedly refine the value assigned to r until $f(r)$ was close enough (to p decimal places) to zero.

A simpler formulation that uses less storage space and less time is

```
Let R(x) = [|f(x)| < 10*(-p): x; R[x - f(x)/g(x)]].
```

As an example, to find an approximate root of

$$x^3 - 10x^2 - 6x + 10$$

that gives a value within, say, six decimal points of zero, define

```
Let f(x) = x*3 - 10•x*2 - 6•x + 10.
Let g(x) = [f(x + d) - f(x)]/d.
d = .000001
Let R(x) = [|f(x)| < 10*(-p): x; R[x - f(x)/g(x)]].
p = 6
```

where a first-order difference approximation over what seems to be a suitable mesh size, d, has been used. A root is found by using zero as an initial approximation:

```
Type R(0), f[R(0)].
      R(0) =          .765278678
 f[R(0)] =       -1•10*(-7)
```

EXPRESSING MORE GENERAL OBJECTS

Expressions for objects other than decimal and logical values are admitted by JOSS as follows:

1. The user can refer to individual steps, parts, forms, files, and items of files for purposes of typing, deleting, filing, and (when appropriate) execution. The corresponding expressions consist of the associated noun (*step, part, form, file, item*) followed by an expression for a decimal value.
2. The user can refer to collections of objects for purposes of typing, deleting, and filing: *all steps, all parts, all forms, all values, all formulas,* and *all.*
3. For purposes of typing only, the user can request
   a. Blank lines or blank fields in a form, indicated by an underscore or a string of consecutive underscores.
   b. The number of storage units that are being used by JOSS for storing the user's program, data, and other information (*size*).†
   c. The time of day at RAND (*time*).†
   d. The number of users being serviced by JOSS (*users*).†
   e. A list of the items in any of the user's files (by item number, item identification, date of filing, and project number (*item-list*)).
4. The user can use single letters to refer to formulas and arrays

†JOSS updates the values associated with *size, time, users* (and *timer*) each time the interpretation of a step (direct or indirect) begins, and only then. Thus, these values remain fixed during the interpretation of a single step.

for purposes of typing, deleting, and filing, and for assignment as actual parameters of formulas.

It turns out that the most general expressions for such objects, as well as for values, can best be described in terms of expressions for so-called elementary operands:

1. Decimal and logical values not assigned to any letter.
2. Single letters to which scalar values, arrays of values, or formulas have been assigned.
3. *$, timer, size, time, users,* and strings of consecutive underscores.
4. Identifiers for functions (e.g., *sin, cos, sum, max*).

The simplest examples of expressions for values not assigned to a single letter are decimal and logical numerals, *$*, and *timer*. More complex expressions for elementary operands are formed recursively from these and the other elementary operands in conjunction with groupers, punctuation, and operator signs. It is clear that many expressions that are ostensibly well formed have little utility. For example, one would be hard put to ascribe meaning to the expression *(sin/sum) + 3•max*. On the other hand, if a decimal value had been assigned to the letter *x*, one could read the expression *[x = 0: sin; cos](y)* as "If the value assigned to *x* is zero, evaluate *sin(y)*; otherwise, evaluate *cos(y)*." Here, there seems to be a choice. Should such conditional expressions for function identifiers be allowed or should they be legislated against either on stylistic grounds (they may be difficult to read) or on the grounds that such expositional power might lead to cases in which errors could

produce unintentionally acceptable expressions? Similar questions must be raised about expressions where symbols for traditionally numerical operations are used with logical operands, and conversely, and about expressions where operations are applied to operands that identify complete arrays or formulas. The easiest tack would have been to allow arbitrarily complex expressions for objects wherever the simplest expression for such objects had "meaning." JOSS's actual behavior is less permissive and, therefore, harder to implement and to explain.

1. Mixed arithmetic is not allowed: Expressions for decimal values must be used with traditionally numerical operations, and expressions for logical values with traditionally logical ones.† The user is provided the notation and terminology for specifying the explicit replacement of decimal values by "equivalent" logical values, and conversely.††
2. Conditional expressions for function identifiers can be used only for expressing actual parameters of formulas.
3. Conditional expressions for array and formula identifiers can be used only for expressing actual parameters of formulas and for purposes of typing, filing, and deleting.
4. Expressions for *size*, *time*, *users*, and strings of underscores may be used for purposes of typing only.
5. Within the above restrictions, arbitrarily complex expressions

---

†However, mixed operands and logical operands may be tested for equality and inequality; mixed operands are always unequal.

††See p. 83 (*tv*) and p. 84 (absolute value).

for elementary operands may be used freely for purposes of computation, typing, expressing actual parameters of formulas, and for assignment as formulas.

The principal effect of this extended treatment of operands on the user (and its justification) is that it allows him to use identifiers for arrays, formulas, and functions (as well as expressions for decimal and logical values) as actual parameters of formulas. For example, to define a formula for computing the average of a list of *n* values:

```
Let m(L, n) = sum[i = 1(1)n: L(i)]/n.
a(1) = 1
a(2) = 2
a(3) = 3
a(4) = 4
a(5) = 5
Type m(a, 5).
     m(a, 5) =        3
```

INTERPRETING EXPRESSIONS FOR OBJECTS

The bulk of JOSS's interpretation is handled by a group of re-entrant subroutines:

1. P49 controls the interpretation of expressions for elementary operands.
2. P42 controls the interpretation of properly grouped lists of expressions for elementary operands, separated by commas.
3. P40 controls the interpretation of left-hand-side expressions: a single letter or a single letter followed by a properly grouped list of expressions for index values, separated by commas.

4. P39 controls the interpretation of *for* phrases: a left-hand-side expression followed by an equals sign followed by a comma-separated list of expressions for values and ranges of values for iteration.
5. P35 controls the interpretation of suspected conditional expressions for elementary operands.

As a convenience in error control, other syntactic entities are handled by distinct subroutines:

1. P38 controls the interpretation of expressions for objects of discourse: individual steps, parts, forms and formulas, and collections of objects.
2. P38E controls the interpretation of expressions for individual steps and parts for purposes of execution.
3. P37 controls the interpretation of expressions for elementary operands to be deleted or filed.

Routines P38, P38E, and P40 are also used for determining and reporting whether an instance of the associated construct might be coming up. The choice of which of the routines to use is always made on the basis of what type of construct is expected. If the interpretation is successful, the appropriate information is compiled and stored in the user's block, and a descriptor representative of the compiled information is stored atop a common operand stack in the user's block. This stack (DS)† is the key to error recovery and snap back, since it

†See p. 46.

constitutes a record of all temporary uses of the user's available space. The major part of cleaning up and backing off after errors is the releasing of all space used for these operand descriptors and for the compiled information associated with each descriptor. In general, the direct representation of objects, once evaluated and compiled, is rarely used; all bookkeeping is done in terms of descriptors for the objects. In programming parlance, all bookkeeping is done "by name" rather than "by value."

The routines associated with the major syntactic constructs are similar to those for controlling the interpretation of commands and "read" like a description of the construct. Routine P49 is the major exception to this rule, reading like a simplified syntactic description of operand, wherein the conventional rules for association and precedence of operators have been stripped from the syntax and arithmetized. Arithmetization consists of associating a pair of numerical values with each operator and then replacing the original syntax by a collapsed one, augmented by a rule for comparing the values associated with operators to determine the order of operations. Pairs of values rather than single values are required, since conventions concerning left and right associativities of operations usually result in a partial ordering of the operations.

Routine P35 is an appendage of P49, and its principal function is the marking of chosen subexpressions of conditional expressions to be typed.[†]

[†] See p. 134.

Elementary Operands

The reentrant routine P49 controls the interpretation of expressions for elementary operands. The operand's descriptor is stored atop the operand stack (DS), and the descriptor for the terminal character to the right of the expression is stored in a fixed location, labeled CC.

Expressions for elementary operands are formed recursively from

1. Terminal operands: decimal numerals, *true*, *false*, letters, $, strings of underscores, *size*, *time*, *users*, *timer*, and function identifiers.
2. A unary operator followed by an elementary operand expression.
3. Two elementary operand expressions connected by a binary operator.
4. An elementary operand expression bracketed by matched groupers or absolute value bars.
5. A conditional expression for an elementary operand.
6. A function, array, or formula identifier followed immediately by a properly grouped list of comma-separated elementary operand expressions.
7. A function identifier followed by a properly grouped expression consisting of an iteration specification separated from an elementary operand expression by a colon.

Minor variations on this theme are caused by

1. The use of conventional rules for spacing.

2. Prohibitions against the application of numerical operations and relations to logical operands and vice versa.
3. Restrictions on the use of underscores, *size*, *time*, and *users* (which may be typed only).
4. Prohibitions against the use of consecutive operators (e.g., *a**-*b*) not set off by groupers (for readability).
5. The fielding of unassigned letters as they are met (as a convenience in error control).

Central to all interpretation is Routine P51, which is used to advance to the next terminal character of the string of characters being interpreted. A descriptor for the terminal character is stored in CC, and a count of the number of leading blanks is stored in a fixed location. Descriptors for operators and left groupers (treated as operators to speed up processing) consist of two numeric codes. One is used to differentiate among operators, the other to differentiate operators from other classes of terminal characters. Once the latter differentiation has been made, its code is no longer used. Two uses are made of this free slot:

1. As a cache for counting the number of operands in a grouped list--associated with the code for the leading left grouper.
2. As a cache for the address number at which processing will be picked up after use of a reentrant routine--associated with a code for the "back stop," an operatorlike character used to indicate "beginning of syntactic construct."

Routine P49 may be pictured as a two-state machine. In one state, an elementary operand is expected; in the second, an operator or a delimiting terminal character. Flowchart 2 depicts the overall flow of control from P49.1 (expecting an elementary operand) to P48 (expecting an operator or delimiter for the expression) and back.

The flowcharts are presented as lists of actions and are meant to be read from top to bottom. Alterations in the sequencing (caused by differential choices) are indicated by labels in the two rightmost columns. The absence of a label indicates the continuation of the top-to-bottom scan.

Note that P49.1 is used again after the occurrence of a left grouper, rather than P42 (the routine for evaluating grouped lists of elementary operands). That routine need not be used because JOSS does not permit direct arithmetic or logical operations on lists of operands. For example,

$$(a, b, c) + (x, y, z)$$

does not express a legitimate JOSS operation. A slight gain in speed results, because some context must be saved each time P42 is used. Function, formula, and array identifiers require excursions to carry out the necessary interpretation. These are depicted in Flowcharts 3 (page 108) and 4 (page 115) and will be described in detail later. These excursions usually result in reentrant uses of P49; for example, when evaluating lists of elementary operands. Once an operand has been determined, its descriptor is stacked atop DS, and control flips to the second state (P48).

Flowchart 2--ELEMENTARY OPERANDS

LABEL	ACTION	YES	NO
P49	Store a back-stop descriptor, with the calling instruction's address number, in CC.		
P49.1	P52: Save the current operator CP on the operator stack PS.		
	Set CP = CC.		
	P51: Fetch the next terminal character's descriptor to CC.		
P49.2	What is the next terminal character?		
	Left grouper, absolute value bar, unary plus, unary minus, *not*.	P49.1	
	Letter. Fetch assigned operand's descriptor from assignment table.		
P49.7	Not assigned	E6	
	Scalar value	P49.9	
	Array of values	P44	
	Function identifier	P44F	
	Formula	P41	
	String of underscores, *size*, *time*, *users*	→	
	None of the above	E5	
	Function identifier.	P44F	
	Numeral, $, string of underscores, *size*, *time*, *users*, *timer*.	→	
	None of the above.	E5	
→	Copy in the user's block if a decimal value. Compose the operand's descriptor.		
P49.9	Copy the operand's descriptor on the operand stack DS.	P48	

Flowchart 2 (continued)

LABEL	ACTION	YES	NO
P48	P51: Fetch the next terminal character's descriptor to CC. If the next terminal character is a binary operator: Set U4 = right weight for the operator. Otherwise: Set U4 = 0.		
P48.6	Is the left weight for the current operator CP less than U4? That is, must things be deferred until the next operand is obtained?	P49.1	
	Use the evaluation-control routine (MP1 through MP8) associated with the current operator CP.		
	Left grouper	MP1	
	Absolute value	MP2	
	Binary arithmetic	MP3	
	Unary plus and minus	MP4	
	Binary logic (*and*, *or*)	MP5	
	Unary logic (*not*)	MP6	
	Numerical relations	MP7	
	Back stop	MP8	
	NOTE: The descriptor for the operand resulting from the operation, if any, is always copied or left on the operand stack DS. The descriptor for the previous operator is popped off the operator stack PS and copied in CP. Control is passed to P48 after groupers and absolute values, to the calling instruction after back stops, and to P48.6 after all other operations.		

In the second state, a binary operator or a delimiting character is expected. Any terminal character other than a binary operator is an acceptable delimiter for expressions for elementary operands. The right weight associated with the new operator (zero for delimiting characters) and the left weight associated with the last operator are compared to determine whether the last operation will be carried out or will be deferred in favor of the new one. The left and right weights associated with all operators are listed in Table 13. Note that the operation of raising to a power associates from the left and takes precedence over unary plus and minus. In effect, unary uses of plus and minus are treated as though a zero operand had been inserted before the plus or minus sign.

Table 13

OPERATOR WEIGHTS

Operator	Left Weight	Right Weight
*	90	90
+ - (unary)	80	80
• /	70	70
+ -	60	60
relations	50	50
not	40	0
and	30	30
or	20	20
(	0	0
back stop	0	--
all others	--	0

Flowchart 3a--ARRAYS

LABEL	ACTION	YES	NO
P44	P55: Followed immediately by a left grouper?		P49.9
	P42: Compile the upcoming grouped list of elementary operands, stacking their descriptors on the operand stack DS.		
	P61: Peel descriptors off the operand stack DS, examining the elementary operands. Are they acceptable index values?		E9
	Examine the assignment-table entry for the letter being indexed.		
	Does it identify an array?		E10
	Does its dimensionality match the number of index values just compiled?		E10
	P57: Has this particular component of the array been assigned?	→	
	Is the array sparse?		E10
	Use a zero value.		
→	Copy the component in the user's block if a decimal value, and compose the operand's descriptor.	P49.9	

Flowchart 3b--FUNCTIONS

LABEL	ACTION	YES	NO
P44F	P55: Followed immediately by a left grouper?		P49.9
	P42: Compile the upcoming grouped list of elementary operands, stacking their descriptors on the operand stack DS.		
	Is the number of operands appropriate?		E11
	Use the appropriate routine (SP6 through SP23) to control the evaluation.		
	Copy the result in the user's block if a decimal value, and compose the operand's descriptor.	P49.9	

Formulas and Iterative Functions

Several distinct mechanisms exist for interpreting instances of formulas. Although the details may differ, the effects may be characterized as follows. Suppose an instance of a formula occurs in an expression being interpreted:

1. The expressions for the actual parameters of the formula may be interpreted in two ways:
   a. Replace the formal parameters, in a copy of the formula, by the expressions for the actual parameters.
   b. Evaluate the expressions, obtaining elementary operands, and replace the formal parameters, in a copy of the formula, by the simplest expressions for the corresponding elementary operands.
2. The copy of the formula may then be interpreted in two ways:
   a. Replace the instance of the formula, in a copy of the expression being interpreted, by the copy of the formula obtained from item 1, and continue interpretation of the expression at the point of insertion.
   b. Evaluate the copy of the formula, obtaining an elementary operand; replace the instance of the formula, in a copy of the expression being interpreted, by the simplest expression for the elementary operand and continue interpretation at the point of insertion.

The formal substitution of items 1a and 2a is quite general in that ill-formed expressions may be used. For example,

Let f(x, y) = xy).

Type f(3/, (6).

would result in

Type 3/(6).

The mechanism is most useful in applications where the construction and manipulation of character strings is more frequent or more important than their eventual interpretation. However, in such languages as JOSS, the mechanism must be used with care to avoid violations of conventional practices. For example,

Let f(x, y) = 3·x + y.

Type f(10 + 5, 100)/100.

allows the following four interpretations:

1a, 2a	*3·10 + 5 + 100/100 = 36,*
1a, 2b	*(3·10 + 5 + 100)/100 = 1.35,*
1b, 2a	*3·15 + 100/100 = 46,*
1b, 2b	*(3·15 + 100)/100 = 1.45.*

In JOSS, the last result is probably the expected one. In general, rules 1b and 2b produce more conventional results than the other combinations when used with "algebraic" programming languages. A more versatile mechanism to achieve the same effect could be attained by using rules 1a and 2a, and placing parentheses about the inserted strings when "necessary." Some such middle ground is required, because the actual parameters may be single letters to which arrays or formulas have

been assigned; or they may be conditional expressions for such letters. In JOSS the effect is achieved by using P49 to evaluate expressions for the actual parameters, producing elementary operands as results.

None of the mechanisms allow the use of abbreviations in a natural manner. For example, define a function $P(p, a, b)$, where p denotes the probability of an event, and a and b are parameters, by

$$P(p, a, b) = ap + bq \quad \text{where} \quad q = 1 - p. \tag{1}$$

In JOSS, the corresponding definition would be written

$$\begin{aligned} &\textit{Let } P(p, a, b) = a \cdot p + b \cdot q. \\ &\textit{Let } q = 1 - p. \end{aligned} \tag{2}$$

A request to evaluate, for example,

$$P(.5, .8, .2)$$

would result in an evaluation of

$$(.8)(.5) + (.2)(1 - p).$$

That is, the replacement of the formal parameter p by the actual parameter has occurred only in the definition of P, despite the fact that the intent was that the substitution be carried out in formula q as well. To achieve the desired interpretation, q would have to be defined as $q(p)$. The problem is one of determining the intent of the user. In Eq. (1), the intent is made clear by the use of the word "where," which effectively indicates how q is used. No such intention can be inferred from Eq. (2) without reading the user's mind, although

some guesses could be made on the basis of the continuity of the two statements and of the "usual" relationship between p and q.

The mechanization of such inferences would undoubtedly be costly in terms of processing speeds. More important, there seems to be no mechanization that would give simple and consistent interpretations in the absence of specific declarations of intent from the user. For in the absence of declarations of intent, only two strategies can be employed:

1. Make substitutions in the formula only.
2. Make substitutions in the formula and in all subformulas to all levels.

The first, as has been shown, does not permit the use of abbreviations without explicit use of formal parameters in the definitions of the abbreviations. The second has several serious drawbacks:

1. The user would have to know the precise order in which substitutions are made before he could determine the interpretation of an expression.
2. Slight modifications in formulas and subformulas could lead to different interpretations; the user would have to be continually on his guard.
3. Most important, the user would have no control over the process of substitution other than by the careful use of identifiers.

On the other hand, the method's generality could make it appropriate

to systems designed for users who require the generality and are willing to exercise the necessary care. It seems less appropriate for systems such as JOSS that are tuned to the casual user. JOSS's interpretation can best be described in terms of assignments of elementary operands to formal parameters and to letters used as variables of iteration for iterative functions.

Operands can be assigned to letters in four ways:

1. By an assignment statement.
2. By using the letter as a variable of iteration to control the execution of a stored program repeatedly.
3. By using the letter as a variable of iteration to control the evaluation of an iterative function.
4. By using the letter as a formal parameter of a formula and then using the formula.

In the first two cases, the existing assignment is deleted before the new one is made; such assignments are said to be global. In the last two cases, the existing assignment is saved before the new one is made; such assignments are said to be local (to the formula or to the function). Each entry in the assignment table is the most recent of a stacked sequence of assignments made to the associated letter. The first assignment, the global one, may indicate that the letter is unassigned. Associated with each assignment is a record of the level of formula nesting that attained at the moment of the assignment; global assignments are at level zero.

In the absence of formulas, the choice of which assignment to

associate with an occurrence of a letter is immediate: Always choose the most recent one. The mechanism is fine for iterative functions but fails for formulas. For example,

y = 2

Let f(x) = x + y.

Let g(y) = f(y).

Type g(1).

would result in

g(1) = 2

which is incorrect. In brief, the user would not be able to use letters freely as formal parameters--an intolerable situation. A direct mechanism for treating formal parameters correctly is to

1. Replace all instances of formal parameters in a formula by unique 8-bit codes denoting "first formal parameter," "second formal parameter," etc.
2. Associate with each of these objects a unique entry in an augmented assignment table--ten more entries suffice for JOSS.
3. Use the most recent assignment associated with letters or the ten parameter codes.

The technique actually used (see Flowchart 4), which permits a little more control over the situation, is to

1. Keep track of the level of formula nesting by incrementing a level counter prior to the evaluation of a formula and

decrementing the counter on completion of the evaluation.

2. Post with each assignment a record of the level of formula nesting that attained when the assignment was made.
3. Choose the most recent assignment associated with a letter if the assignment has been made at the current level of formula nesting; otherwise, choose the first assignment.

Flowchart 4a--FORMULAS

LABEL	ACTION	YES	NO
P41	Formal parameters associated with the formula?	P41.1	
	Increment level of formula nesting (LEVEL).		P41.6
P41.1	P55: Followed immediately by a left grouper?		P49.9
	P42: Compile the upcoming grouped list of elementary operands, stacking their descriptors on the operand stack DS.		
	Is the number of operands correct?		E22
	Are all the operands assignable objects?		E5
	P58: Save (push) the assignment-table entries associated with the formal parameters.		
	Assign the actual parameters to the formal ones, incrementing the use count when assigning decimal values, arrays, and formulas.		
P41.6	Save formula identification and place marker in the current string being interpreted on the formula push-down list FPDL.		
	P49: Evaluate the formula.		
	Decrement the level of formula nesting (LEVEL).		
	Formal parameters associated with the formula?		P49.7
	Delete the actual parameter assignments to the formal parameters, and restore the original assignments.	P49.7	

Flowchart 4b--ITERATIVE FUNCTIONS

LABEL	ACTION	YES	NO
P36	P55: Followed immediately by a left grouper?		P49.9
	Which notational styles does the function admit?		
	Grouped list of operands.	P36.9	
	Expression defined over a range of values.	P36.0	
	Either style; does upcoming string resemble a *for* phrase?	P36.0	
P36.9	Assume grouped list of operands. Does function admit this style?		E5
	P42: Compile the upcoming grouped list of elementary operands, stacking their descriptors on the operand stack DS.		
	Use the appropriate routine (SP18 through SP23) to control the evaluation.		
	Copy result in the user's block if a decimal value, and compose the operand's descriptor.	P49.9	
P36.0	P39: Compile the upcoming *for* phrase, leaving descriptor for the list structure on the operand stack DS.		
	Followed by a colon?	→	
	Followed by the appropriate right grouper?		E5
	Erase *for* phrase list structure.	P36.9	
→	Save function identification and place holder in current string on the formula push-down list FPDL.		
	Save (push) assignment-table entry for the variable of iteration and generate the appropriate initial value as the first partial result.		
P36.1	Copy descriptor for partial result on DS.		
	Is next value for iteration a decimal value?		E5
	P67: Assign it to the variable of iteration.		
	P49: Evaluate the next expression.		
	Followed by the appropriate right grouper?		E5
	Calculate next partial result.		
	P71: Any more values for iteration?	P36.1	
	Tidy up and restore context.	P49.9	

The interpretation is described simply: Letters used as formal parameters in a formula are replaced by the simplest expressions for the actual parameters (parenthesized if necessary) prior to evaluation of the formula.

Since this scheme does not allow "local" abbreviations to be defined in a natural manner, it became necessary to consider methods for allowing the user to exercise some control over the scope of assignments to formal parameters. Several methods were examined and discarded for a variety of reasons; for example:

1. Allowing the user to associate with a formula, subformulas as local abbreviations: This foundered because of the difficulty in easily expressing, modifying, and overlapping such abbreviations.
2. Allowing the user to define some formulas as global abbreviations: This was discarded as being incomplete without the ability to define local abbreviations.
3. Treating formulas with no formal parameters as global abbreviations: This was felt to be dangerous, as well as incomplete.
4. Allowing the user to mark those formal parameters of a formula (in the definition of the formula) whose assignments were to be substituted in all subformulas of the formula: This was discarded because of incompleteness, and vague feelings of dissatisfaction about what appeared to be a "programming" artifice.

Were subprogram structures to be added, the entire question would be reopened. In fact, the principal reason for discarding the methods described above was a belief that they should be considered in the light of the general problems of program structure and declarations, rather than in a restricted context.

Grouped Elementary Operands and Conditional Expressions (Flowchart 5)

Routine P42 compiles lists of elementary operands, each evaluated by using P49. Operand descriptors are stored on the operand stack, DS, in order of appearance, and a count of the number of operands is stored. Routine P42 is also used to sense upcoming conditional expressions, so that they may be used for function and formula parameters or as indices of vectors without the required extra set of groupers. For example, *sin(a = b: x; y)* may be used instead of the technically required *sin((a = b: x; y))*. The evaluation subroutine for left groupers, MP1, also determines upcoming conditional expressions by noting the current terminal character, CC, whenever MP1 is "fired." If CC is not a matching right grouper, P35 is used.

The main duty of P35 (not indicated on Flowchart 5) is to mark conditional expressions to be typed, so that the subroutines for typing can type only the chosen subexpression.† Routine P35 replaces the left and right delimiters (groupers, colons, semicolons) of the chosen subexpressions by unique surrogate characters.

†See pp. 128, 134.

Flowchart 5--GROUPED LISTS AND CONDITIONAL EXPRESSIONS

LABEL	ACTION	YES	NO
P42	Save context and initial count (zero).		
P42.1	P49: Evaluate the next elementary operand.		
	Increment the count.		
	Followed by a comma?	P42.1	
	Followed by the appropriate right grouper?	P42.3	
P42.2	Only one elementary operand collected?		E5
	P35: Evaluate upcoming conditional expression.		
P42.3	Restore context.	EXIT	
P35	Save context.		
P35.1	Followed by a colon?		E5
	Pop last operand off DS.		
	Logical value?		E5
	Is it true?	P35.5	
P35.2	Skip over the next expression.		
	Followed by appropriate right grouper?	E61	
	Followed by a semicolon?		E5
	P49: Evaluate the next elementary operand.		
	Followed by appropriate right grouper?	P35.9	P35.1
P35.5	P49: Evaluate the next elementary operand.		
	Followed by a semicolon?		→
	Skip over the remainder of the conditional.		
→	Followed by appropriate right grouper?		E5
P35.9	Restore context.	EXIT	

Objects of Discourse (Flowchart 6)

Routines P38 and P38X evaluate expressions for individual steps, parts, forms, and formulas (e.g., *formula f*), and for collections of objects. Routine P38 simply notes the type of object and the object's identification, if applicable, in scratch storage; P38X copies the

Flowchart 6--OBJECTS OF DISCOURSE

LABEL	ACTION	YES	NO
P38	Save context; fetch the next terminal character.		
	The word *formula*?	P38F	
	The word *all*?	P38.0	
	A singular noun?	P38.1	
	Restore context and report failure.	EXIT	
P38.0	If next terminal character is a plural noun, compose the appropriate OOD descriptor; otherwise, compose the descriptor for *all*.	P38.9	
P38.1	Is this a simply described object (a singular noun followed by a decimal numeral that does not seem to be embedded in a complex expression for an elementary operand)?	P38.3	
P38.2	P49: Evaluate the upcoming elementary operand.		
	P53: Pop result off DS. Is it a decimal value?		E5
P38.3	P70: Is the object's number appropriate?		E32
	Has the object been defined?		E33
	Compose the appropriate OOD descriptor.	P38.9	
P38.9	Restore context and report success.	EXIT	
P38F	Is the word *formula* followed by a single letter to which a formula has been assigned?		E5
	Compose the appropriate OOD descriptor.	P38.9	

object's identification in the user's block and stores the proper descriptor on DS.

The procedure is straightforward except for the extra detail necessary to detect expressions for simply described objects without using any available space.

If the first terminal character encountered indicates that no expression for an object of discourse is coming up, context is restored and failure is "reported."

Iteration Specifications and Left-hand Sides (Flowchart 7)

Routine P39 compiles list structures associated with *for* phrases: a left-hand side followed by an equals sign, followed in turn by a list of values and ranges of values for an iteration. The structure consists of a left-hand-side list and a list of scalar values. Single values and right limits for ranges of values are marked to permit control of the iteration, and logical values are marked to distinguish them from decimal ones. Ranges of values whose left and right limits are equal collapse into a single value. Otherwise, zero step values and unreachable right limits (e.g., *1(-1)10*) are treated as errors.

Routine P40 compiles list structures associated with acceptable left-hand sides for assignment: a single letter or a letter followed immediately by a properly grouped list of expressions for index values. If the first terminal character is not a letter, context is restored and failure is "reported." Otherwise, barring errors, a list consisting of the letter's assignment-table address and the index values (if any) is compiled, and a descriptor for the list is stored on DS.

Flowchart 7a--RANGES OF VALUES AND *for* PHRASES

LABEL	ACTION	YES	NO
P39	Save context.		
	P40: Compile list structure for left-hand side, if one is coming up.		E5
	Followed by an equals sign?		E5
	Compose descriptor for null range-of-values list.		
P39.1	P49: Evaluate upcoming elementary operand.		
	P63: Store value on tail of ROV list.		
P39.11	Followed by a left grouper?		P39.2
	P42: Compile upcoming grouped list of elementary operands.		
	A single item?		E5
	P63: Store value on tail of ROV list.		
	P49: Evaluate upcoming elementary operand.		
	P63: Store value on tail of ROV list.		
	Examine the last three values:		
	Do they define an acceptable range?		E23
	Collapse into a single value if the right and left limits are equal.	P39.11	
P39.2	Mark the last value as being the last in a range (or a single value).		
	Followed by a comma?	P39.1	
	Compose *for* phrase list structure from left-hand side and range-of-values structures.		
——	Compose descriptor; leave it atop DS; restore context.	EXIT	

Flowchart 7b--LEFT-HAND SIDES

LABEL	ACTION	YES	NO
P40	Save context. Fetch the next terminal character.		
	Is it a letter?	→	
	Restore context and report failure.	EXIT	
→	Followed immediately by a left grouper?		P40.2
	P42: Compile upcoming list of elementary operands, stacking descriptors on DS.		
	Are there too many?	E8	
	P61: Peel the descriptors off DS and examine the objects they define.		
	Values?		E5
	Acceptable as index values?		E9
P40.2	Compose the appropriate list structure.		
	Compose the list's descriptor and store it atop DS.		
	Restore context and report success.	EXIT	

SPARSE ARRAYS

JOSS permits the user to conserve storage and increase processing speeds for arrays of predominantly zero values by allowing him to assign only pertinent elements of such arrays and then identify the array as being "sparse." For purposes of computing and typing, JOSS assumes that all unassigned components of sparse arrays are zeroes.

The user identifies the array by its letter and tells JOSS to treat it as sparse; for example, *Let A be sparse.* If the letter identifies an array, JOSS notes that it is to be so treated, *but does not clean out existing zero elements.* If the letter does not identify an

array, JOSS deletes the current assignment before making the new one, but rejects all subsequent references to the array until its dimension has been established by the assignment of a value to an element of the array.

When the user requests that a sparse array be typed or filed, JOSS identifies the array as being sparse after typing or filing its components:

```
a(1) = 10
a(3) = 7
a(10) = 12
Type a.
        a(1) =       10
        a(3) =        7
       a(10) =       12
Type sum[i = 1(1)10: a(i)].
a(2) = ???
Let a be sparse.
Type sum[i = 1(1)10: a(i)].
sum[i = 1(1)10: a(i)] = 29
Type a.
        a(1) =       10
        a(3) =        7
       a(10) =       12
a is sparse
Type a(4).
        a(4) =        0


Let B be sparse.
Type B.
B = ???
Type B(4).
B(4) = ???
B(1) = 1
Type B(4).
        B(4) =        0
Type B.
        B(1) =        1
B is sparse
```

DELETING OBJECTS (Delete)

Users can delete

1. Individual steps, parts, forms, values, arrays of values, and formulas.
2. Collections of these objects via

all	*all parts*	*all formulas*
all steps	*all forms*	*all values*

For example:

```
Delete all.
Delete part 1, all forms, A, B.
```

JOSS interprets the entire clause from left to right before deleting anything. If an object has not been defined, or if any other error is detected, JOSS aborts interpretation and notifies the user.

Whenever the user deletes all, JOSS cancels all outstanding tasks, resets the user's block to its log-on state (except for accounting information), and returns control to the user without comment.

Whenever all components of an array have been deleted, JOSS effectively deletes the array:

```
Delete all.
x = 1
Delete x, y.
y = ???
Type x.
          x =        1
a(1) = 1
Delete a(1).
Type a.
a = ???
```

TYPING OBJECTS (Type)

The user may request JOSS to type any combination of

1. Decimal values.
2. Logical values.
3. Arrays of values.
4. Formulas.
5. Individual steps, parts, and forms.
6. Collections of the above via

all	*all forms*
all steps	*all formulas*
all parts	*all values*

7. Blank lines, indicated by an underscore or a string of consecutive underscores.
8. The number of units of storage currently being used by JOSS for storage of the user's program, data, and other information (*size*).
9. The time of day at RAND (*time*).
10. The current number of users being serviced by JOSS (*users*).

For example:

```
Type sqrt(x), x, y.
Type part 1, all forms, A, B, x + sqrt(y).
```

JOSS interprets the entire imperative clause from left to right before typing any of the required output. An error encountered during

interpretation will cause JOSS to abort interpretation of the command after commenting on the error:

```
Type sqrt(x), x < y.
x = ???
x = 12
Type sqrt(x), x < y.
y = ???
y = log(x)
Type sqrt(x), x < y.
     sqrt(x) =        3.46410162
       x < y =      false
```

JOSS will type values one-per-line, using a standard indentation whenever possible, and will try to line up decimal points and equals signs, compromising first on the equals signs, then on the decimal points, and finally on the single-line style:

```
Type x, sqrt(x), sqrt(sqrt(x)), sqrt(sqrt(sqrt(x))), sqrt(sqrt(sqrt(sqrt(x)))).
           x =       12
     sqrt(x) =        3.46410162
sqrt(sqrt(x)) =       1.86120972
sqrt(sqrt(sqrt(x))) = 1.3642616
sqrt(sqrt(sqrt(sqrt(x)))) = 1.1680161
```

```
Type sqrt(1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1).
sqrt(1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1)
 = 5.56776436
```

JOSS uses fixed-point notation for decimal values except where the magnitude of the number makes this unreasonable:

```
x = sqrt(x)
Type x, x•10*5, x•10*6, -x•10*6.
           x =        3.46410162
      x•10*5 =   346410.162
      x•10*6 =        3.46410162•10*6
     -x•10*6 =       -3.46410162•10*6
```

JOSS will usually identify values by the exact expressions used

in the command calling for the output. If a conditional expression is used, JOSS will use the chosen subexpression, enclosed in parentheses or brackets, to identify the output. Values, arrays, and formulas identified by single letters will always be identified by the letter, no matter how expressed. For example:

```
a(1) = 1
a(2) = 2
i = 2
Type a(2), a(i), [i = 0: x; sqrt(x)].
        a(2) =          2
        a(i) =          2
   [sqrt(x)] =          1.86120972

Type (i = 0: x; a).
        a(1) =          1
        a(2) =          2

Let f(x) = x.
Type f(f), f(a), f(_), f(time), f(i), f(i = 2).
        f(x):  x
        a(1) =          1
        a(2) =          2

        time:  0939
        f(i) =          2
    f(i = 2) =       true
```

JOSS always interprets references to formulas having no formal parameters as implied requests for the actual evaluation of the formula. To see the actual definition of such a formula, the user must request JOSS to type *formula*.... For example:

```
Let h = (b - a)/n.
b = 1
a = 0
n = 100
Type h, formula h.
          h =          .01
          h:  (b - a)/n
```

```
Let h = f.
Type f, formula f, h, formula h.
        f(x):  x
        f(x):  x
        f(x):  x
           h:  f
```

JOSS uses a fixed order of output when requested to type all:

1. All steps, ordered by step number and separated into parts.
2. All forms, ordered by form number.
3. All formulas.
4. All scalars.
5. All arrays of values, separated into arrays of common dimension, in order of increasing dimension.

Collections of formulas and values are typed in the lexicographic order of the identifying letters, first the upper-case letters, then lower-case.

Blank lines are used to set off subcollections and to indicate the absence of members of subcollections of parts, forms, and values. For example:

```
Type all.
```

```
f(x):  x
   h:  f

   a =         0
   b =         1
   i =         2
   n =       100
   x =         3.46410162
   y =         2.48490665
```

Skipping Lines and Pages (Line, Page)

Line causes JOSS to effectively depress the carrier-return key, and *Page* causes JOSS to depress the pagination key. Whenever JOSS ejects the paper to the top of the next page, a standard page heading is printed, followed by a one-inch margin. The heading contains date, time, user identification, console identification, and page number.

When either command is given, if the typewriter carrier is positioned at the end of a page (line 54), JOSS ignores the command and simply ejects the paper and types the usual heading and one-inch margin.

Defining Forms for Formal Output (Form)

Users can direct JOSS to stand ready to accept a full-line form to be stored away and made available for typing formal output. The command may only be given directly and consists of the word *Form*, followed by a valid expression for a decimal value to identify the form, and a colon. Form numbers must be positive integers.

The user then has the entire width of the next line for specifying literal information and fields to be filled in with answers. Two types of fields are provided for displaying numeric answers: A string of underscores with an optional decimal point specifies fixed-point notation, and a string of four or more periods specifies a tabular form of scientific notation in which the digit part and exponent part of the answer are typed. JOSS rounds answers to fit the field and notifies the user whenever unable to express a value in a field. Only strings of underscores can be used for displaying logical values and the current time. For example:

```
Form 1:
Fixed point field: _____    scientific notation: .........

Type form 1.
Fixed point field: _____    scientific notation: .........
Type all forms.

Form 1:
Fixed point field: _____    scientific notation: .........

Type n, x in form 1.
Fixed point field:   100    scientific notation:  3.464 00
```

Typing in User-defined Forms

The user may request JOSS to type any combination of the following in a form he has specified:

1. Decimal values.
2. Logical values.
3. Blank fields indicated by an underscore or string of underscores.
4. *size*, *time*, and *users*.

JOSS interprets the entire list of expressions and the expression for the specified form before typing. An error encountered during the interpretation causes JOSS to abort interpretation after commenting on the error.

Once the expressions have been evaluated and the form found, JOSS begins to replace the fields in a copy of the form by the literal strings of characters representing the values. JOSS does not type the line until assured that

1. No used fields abut.
2. Each field is long enough to hold the associated literal string.

3. There are at least as many fields as there are items. For example:

```
Type n, x in form 1.
Fixed point field:   100     scientific notation:  3.464 00

Form 2:
___.........

Type n, x in form 2.
I can't make out your fields in the form.

Form 2:
__ .........

Type n, x in form 2.
I can't express value in your form.

Form 2:

___

Type n, x in form 2.
I have too many values for the form.

Form 2:
___  .........

Type n, x in form 2.
100  3.4641 00
```

If there are more fields in the form than items to be typed, JOSS will chop off the form after filling in the last required field, and will try to type pertinent, literal information following the last filled field:

```
Form 1:
current = ___.__ amps.   voltage = ____.__ volts

Type x in form 1.
current =   3.46 amps.
```

JOSS can make mistakes in chopping off forms, but will include conventionally spaced, literal information when chopping off forms.

Implementation Notes

Users must be able to type, file, and delete objects without concerning themselves with the number of available cells. Even when available space has been "exhausted," JOSS must be able to perform the requested actions when the objects are described by simple expressions (e.g., *all parts*, *form 2*, *step 1.5*, *formula f*, *x*, *A*). This can be done by holding back a part of available space to be used only when typing, filing, or deleting; or by using detailed machine-language segments for sensing simple cases. A compromise is struck as follows:

1. P38 is used for evaluating expressions for simply described steps, parts, forms, and collections of objects without using any available space.
2. Three cells of available space are held back from the user for general use by the routines for typing, filing, and deleting.

The technique for reserving the cells is mundane:

1. The recorded number of available cells is always three less than the actual number.
2. Before the coroutines for typing, filing, and deleting are used, the recorded count is increased by three, and a flag is set in the user's block to record the fact.
3. Whenever the user's block is cleaned up, the flag is reset and the count decreased by three if necessary.

Since the typing and filing of all but the simplest objects will

assuredly be suspended at least once, some fixed space in the user's block must be reserved to cache information during such suspensions. This fixed overhead is small except when typing arrays, which must be done in an orderly manner. Restricting the number of indices to ten strikes a balance between denotative ability and fixed drains on the user's block, sets well with the line-at-a-time style of JOSS, and is easily remembered.

The CPU selects chosen subexpressions of conditional expressions to be typed by marking the bytes that delimit each chosen subexpression, unless embedded in an iterative function or a formula. To avoid having to undo or otherwise deal with such markings, only strings included in *Type* commands are so treated, and these are copied into working storage before being interpreted. A flag that controls the marking is set to -1 before the main string of a *Type* command is to be interpreted and to 0 prior to command interpretation. The flag is then increased by one each time the CPU drops a level to evaluate a formula or an iterative function, and is decreased when the evaluation is done. Marking occurs only when the flag is negative.

Quotations and phrases such as *item-list* are factored out first. The remainder of the command is copied, additional space is obtained, and the appropriate flags are set. The balance of the command is interpreted, and descriptors for the objects to be typed are stacked on the operand stack in order of occurrence. During this process, information for typing is collected in the user's block: UP3 is used for counting the number of expressions in the list, UP1 for noting the first byte of the first expression, and UP2 for noting the last byte

of the last expression. Commas separating the expressions are replaced by a unique 8-bit code to facilitate the typing of the actual expressions as identification for scalar values.

Subroutine X48 collects message fragments and types the collected message. It is the only such subroutine and is an integral part of the routines for error and status commentary, typing, and filing.

Except for S55 (the routine for collecting and transforming messages into the 7-bit representation for typing), the process is simple. Requests for pagination are noted, and a buffer area is requested from the supervisory routines. A 7-bit representation of the line to be typed is made in the assigned buffer area, and the message is typed; if another line is required for the message, the process repeats.

Routine S55 controls the collection and transformation of message fragments, which may be represented by

1. The address number of a core location containing a pointer to the first 8-bit token of the 8-bit representation of the string of characters.
2. The pointer itself.
3. The 8-bit representation of the string itself, with a distinctive leading token to differentiate it from 1 and 2.

The transformation to 7-bit encoding is a recursive process, since a single 8-bit token may represent a word or a larger message fragment, and fragments may themselves contain tokens for fragments.

There is no necessity for automatic end-of-line justification. Instead, permissible line breaks are indicated with the message fragments.

EXECUTING STORED PROGRAMS (Do)

To start the task of interpreting steps of a stored program, the user tells JOSS to do a step or part a specified number of times, or repeatedly, for a set of values to be assigned to a variable of iteration.

Expressions for decimal values are used to

1. Identify steps and parts to be done.
2. Specify the number of times a step or part is to be done.
3. Specify initial values, final values, and increments for ranges of values to be assigned a variable of iteration.

Expressions for decimal or logical values can be used to specify individual values to be assigned to a variable of iteration. Ranges of values having identical initial and final values collapse into single values, and ranges of values whose initial and final values are incompatible with the increment are rejected. For example:

```
1.1 Type i.
Do part 1.
Error at step 1.1:  i = ???
Do part 1 for i = 1(-1)2.
Illegal set of values for iteration.
Do part 1 for i = 1(1)2(2)6, 8.
          i =       1
          i =       2
          i =       4
          i =       6
          i =       8
Do part 1, 2 times.
          i =       8
          i =       8
```

The comma preceding the number of times is mainly required for readability, although a glottal-stop convention is sometimes necessary (e.g., *Do part n-x+1 times.* is ambiguous).

Any letter or indexed letter can be used to identify the variable of iteration; JOSS deletes an existing assignment to the letter before assigning a new value for iteration. Since JOSS keeps track of the last value assigned for iteration, the required step or part will always be done for the specified values and ranges of values. For example:

```
1.1 Type i.
1.2 Set i = 2•i + 1.
1.3 Type i, _.
Do part 1 for i = 1(1)2.
          i =        1
          i =        3

          i =        2
          i =        5
```

JOSS calculates the successive values of the iteration variable by adding the increment to the last value rather than by adding the product of the increment and an iteration counter to the initial value. Either method can produce iteration values that are slightly "off" because of round-off error. The first method exhibits a cumulative loss of precision; the second method is sporadically imprecise. In either case, a rule for termination and for assignment of the final value must be selected. JOSS always hits initial and final values on the nose in order to honor all "fixed points" of extended ranges.

For example, the range of values *1(i)10(j)20* will be interpreted so that the values *1*, *10*, and *20* will be hit exactly (and only once) even though the values assigned to i and j do not divide ten evenly. JOSS does this by snapping back to the final value when overshoots occur, despite the fact that this strategy sometimes causes JOSS to carry out an iteration for an extra value:

```
Delete all.
1.1 Type i.
Do step 1.1 for i = 0(1/3)1.
          i =        0
          i =         .333333333
          i =         .666666666
          i =         .999999999
          i =        1
```

It is clear that two rules are required: one for snapping back on overshoots, the other for moving up whenever the current value is close enough to the final one. "Closeness" is usually defined in terms of the increment I; I/n, $I/2^n$, $I/10^n$ are frequently mentioned candidates. However, because no value for n could be found that exhibited uniform behavior, the method was abandoned (a mistake: $I/10^8$ works acceptably).

JOSS never begins execution of a task until certain that

1. The required step or part can be found.
2. There is enough storage space to save the status of the task being suspended.
3. There is enough storage space to carry out the assignment of the first value to the iteration variable.

JOSS begins the interpretation of a part at the first step of the part, and then interprets each step in sequence. Whenever encountering a *Do* command, JOSS pauses to carry out the indicated task as a subroutine before continuing to the next step. When the original step or part has been done the required number of times, JOSS returns control to the user without comment and awaits the next direct command from him. Errors, branching commands, stop commands, commands for terminating a task, and interrupt signals from the user modify this general behavior.

Branching Commands (To)

To break the step-by-step sequence, branching commands direct JOSS to take a specified step or the first step of a specified part as the next step in sequence. JOSS continues the step-by-step sequencing at the new step. The command consists of the verb *To*, followed by a valid expression for a step or a part:

To step 1.1.

To part n + 1.

The branching command may not be given directly, although the obvious interpretation could have been made if given directly during a task interrupted at a step. However, no reasonable interpretation could be found for direct *To*'s given during intratask iteration (*I have nothing to do.* would be misleading in such cases).

Indirect Stopping Commands (Stop)

The *Stop* command causes JOSS to suspend step-by-step sequencing and return control to the user after typing a status message (e.g., *Stopped by step 1.2.*). If the user subsequently tells JOSS to continue, JOSS does so at the step following the stopping command.

Indirect Terminations of Tasks and Portions of Tasks (Done, Quit)

The *Done* command causes JOSS to omit the remaining steps of a part.

The *Quit* command causes JOSS to terminate the current task, as if the required step or part had been done as often as requested.

After ending the current task, JOSS looks around for something to do. If there is nothing to do, JOSS returns control to the user without comment. If there is a suspended task, JOSS continues with the suspended task without comment.

Direct Terminations of Tasks (Cancel, Quit)

The *Cancel* command causes JOSS to terminate all suspended and ongoing tasks and then return control to the user without comment. The user can ask JOSS to *Quit* directly. JOSS notifies the user if there is nothing to do; otherwise, JOSS terminates the current task and looks for a suspended task. If none exists, JOSS returns control to the user without comment. Otherwise, JOSS determines whether the suspended task had been initiated directly or indirectly. If indirectly, JOSS continues with the suspended task; if directly, JOSS returns control to the user after typing a brief status message.[†]

Interruptions

Once the user has directed JOSS to start or to continue a lengthy computation or filing action, he can only regain control of the typewriter before JOSS has finished (or encountered an error or stopping command) by depressing the *interrupt* button. JOSS's response to interruptions is dictated by the "inertia" of the action being performed for the user at the moment. If typing output for the user, JOSS always finishes typing the current line (and often types a few more lines) before honoring the interruption. If a filing action for the user is in

[†]See p. 144.

progress, the interruption is ignored until JOSS has finished or has aborted the action for any reason.

If the interruption occurs after the beginning of a stored program, JOSS responds by typing its position in the stored program before returning control to the user (e.g., *I'm at step 3.2.*). Otherwise, JOSS simply types *Revoked by interrupt.* In either case, JOSS makes sure that the status of the user's block is precisely as it would have been had the command never been started.

The SU sets flags to signal the CPU that the current user has transmitted an *in* signal or that the CPU is to be reassigned to a different user. The overall timing of activities in the JOSS system does not require that the CPU monitor these flags continuously. Instead, they need be examined only at points of "minimum context": between steps, prior to descending a level to evaluate a formula; between iterations for iterative functions; and at other appropriate instances during typing and filing. The maximum delay in honoring these signals is measured by the longest computation that does not involve formulas or iterative functions. In JOSS, this occurs for a string of 38 successive raisings-to-a-power, resulting in a computational delay in the order of 50 ms, or about three-quarters of the least time in which a single character can be typed. The probability of a processing jam occurring during such an interval (by almost simultaneous *carrier-return, off, on,* and *in* signals) is vanishingly small. By the same token, *in* signals are fielded smartly enough to appear instantaneous to the user, except during most disc actions, which are allowed to run to completion before the interruption from the user is honored.

Parenthetical Do's

Each direct *Do* command causes JOSS to cancel all suspended tasks before beginning a new one, unless the command is given in parentheses. JOSS interprets such parenthetical *Do*'s as implied commands to save all suspended tasks and pause to carry out a new task as a subroutine; that is, JOSS treats direct, parenthetical *Do*'s as indirect *Do*'s (but see p. 145).

Originally, JOSS did not admit parenthetical tasks. The treatment of direct tasks was appropriate, because users generally desired a cancellation before introducing a new direct task. Requiring users to cancel would have become onerous; treating direct tasks like indirect ones could have caused the user to use up, inadvertently, the small amount of storage space available in the original system. Since the original system did not tell the user the point in his program where storage space had been exhausted, the user could have been led into indeterminate situations. JOSS now gives the user more status information in such situations. However, since tasks cannot be selectively canceled, users could still be led into situations where the space used to store information about inadvertently uncanceled tasks would be unrecoverable without undoing the current tasks. Accordingly, direct, nonparenthetical tasks are handled as in the original system, and parenthetical *Do*'s are introduced to allow the user to hold a direct task before starting a new one. JOSS always notifies the user of its status on completion or when told to terminate a parenthetical task, and then returns control to the user instead of continuing any suspended tasks.

Parenthetical Cancel's

The *Cancel* command in parenthetical form causes JOSS to cancel only the last directly initiated task and all indirectly initiated tasks stemming from it. JOSS then notifies the user of status before returning control to him.

Continuing Tasks (Go)

Whenever JOSS is awaiting a direct command, the user can instruct JOSS to continue a suspended task by typing the command *Go*. If no suspended task exists, JOSS responds by typing *I have nothing to do*. Otherwise, JOSS's response depends on the manner in which the task was suspended; three possibilities exist:

1. The task was suspended because of an interruption or an error encountered during the interpretation of an indirect step. Such suspensions are characterized as having occurred *at a step*. JOSS continues the task as though that step were the next one to be interpreted.
2. The task was suspended by a stopping command. JOSS's state is characterized as being ready to go *from a step*. JOSS continues the task as though that step had been the last one to be carried out.
3. The task was suspended while JOSS was preparing to repeat a step or part being done repeatedly or over a range of values of an iteration variable. If the *Do* command that initiated the task had been given indirectly, the suspension would have been characterized as occurring *during a step;* otherwise,

during above. JOSS continues at the point of suspension by searching for the required step and carrying out any reassignment of the iteration variable before beginning the step or part anew.

Reporting Status

JOSS reports status and returns control to the user whenever the CPU

1. Honors an interruption from the user.
2. Encounters a stopping command.
3. Completes a direct command that refers to the magnetic-disc files.
4. Completes or is directed to terminate a parenthetical task.
5. Encounters an error.
6. Is unable to obtain enough storage space for the user's block of information.†

If interrupted during the interpretation of a direct command, JOSS types

Revoked by interrupt.

If interrupted during the interpretation of an indirect step, JOSS types

I'm at step....

†See pp. 155-158.

If interrupted by a stopping command, JOSS types

Stopped by step....

JOSS always notifies the user on successfully completing a directly given command that refers to the magnetic-disc files: The typed response *Roger.* indicates that JOSS has found the file requested by the user. The message *Done.* is typed on completion of any action on a file.

Because parenthetical commands can only be given directly, JOSS always returns control to the user on completing or when directed to terminate such a command. At such times it seems reasonable to remind the user that JOSS's status has reverted to what it was when the command was given. If no suspended task exists, JOSS returns control to the user without comment. Otherwise, JOSS types one of the following messages:

1. *Done. I'm ready to go at step ...* if the task was suspended by an interruption or error during the interpretation of an indirect step.
2. *Done. I'm ready to go from step ...* if the task was suspended by a stopping command.
3. *Done. I'm ready to go in step ...* if the task was suspended during an indirectly initiated *Do*.
4. *Done. I'm ready to go.* if the task was suspended during a directly initiated *Do*.

In the first three cases, JOSS also checks to make sure that the step

still exists (the user may have deleted it during a suspension of the parenthetical task), and notifies the user if the step cannot be found (e.g., *Done. I'm ready to go at step 3.2, altho I can't find it.*).

The following set of examples is designed to illustrate JOSS's responses to a number of situations:

```
Delete all.
Form 1:
        Typed by step _._        i = __
1.1 Stop if i = 2.
1.2 Type 1.2, i in form 1.

Do part 1 for i = 1(1)3.
        Typed by step 1.20       i =  1
Stopped by step 1.1.
Go.
        Typed by step 1.20       i =  2
        Typed by step 1.20       i =  3

Do part 1 for i = 1(1)3.
        Typed by step 1.20       i =  1
Stopped by step 1.1.
(Do part 1 for i = 11, 17.)
        Typed by step 1.20       i = 11
        Typed by step 1.20       i = 17
Done. I'm ready to go from step 1.1.
Go.
        Typed by step 1.20       i = 17
        Typed by step 1.20       i =  3

1.1 Done if i = 2.

Do part 1 for i = 1(1)3.
        Typed by step 1.20       i =  1
        Typed by step 1.20       i =  3

1.1 Quit if i = 2.

Do part 1 for i = 1(1)3.
        Typed by step 1.20       i =  1
Go.
I have nothing to do.

1.1 Stop if i = 2.
2 Do part 1 for i = 1(1)3.
2.1 Type 2.1, j in form 1.
Type all parts.
```

```
1.1 Stop if i = 2.
1.2 Type 1.2, i in form 1.

2  Do part 1 for i = 1(1)3.
2.1 Type 2.1, j in form 1.

Do part 2 for j = 11, 17.
       Typed by step 1.20        i =  1
Stopped by step 1.1.
(Do part 1 for i = 1(1)3.)
       Typed by step 1.20        i =  1
Stopped by step 1.1.
Quit.
Done. I'm ready to go from step 1.1.
Go.
       Typed by step 1.20        i =  2
       Typed by step 1.20        i =  3
       Typed by step 2.10        j = 11
       Typed by step 1.20        i =  1
Stopped by step 1.1.
(Do part 1 for i = 1(1)3.)
       Typed by step 1.20        i =  1
Stopped by step 1.1.
Delete step 1.1.
Quit.
Done. I'm ready to go from step 1.1, altho I can't find it.
Go.
       Typed by step 1.20        i =  2
       Typed by step 1.20        i =  3
       Typed by step 2.10        j = 17
```

Implementation Notes

Information defining the current task consists of[†]

1. What is being done: step, part, or nothing at all.
2. How the task was initiated: directly or indirectly.
3. A break-code to indicate whether or not the iteration variable must be advanced before going on.
4. A skip-code to indicate whether the next step to be interpreted is the current one or its successor.

[†]See p. 36.

5. The number of the step or part associated with the current task.

Status and sequencing information residing at fixed locations in the user's block consists of

1. CPI: the integer part of the current step's number.
2. CSI: 10^8 times the fractional part of the current step's number.
3. CSA: the address of the current step's header.
4. U24, U25: the JNF representation of the number of the part or step to be done by the current task.
5. MODE: 0 if in direct mode, 1 if in indirect mode, and 2(3) if advancing to the next iteration of a direct (indirect) task.

In addition, the task status word, JD, contains

1. Job-code: 0 if doing nothing, 1 if doing a part, and 2 if doing a step.
2. Job-mode: 0 if a direct task and 1 if indirect.
3. Break-code: 1 if advancing to the next iteration; otherwise, 0.
4. Skip-code: 1 if advancing to the step after the current one; otherwise, 0.
5. The address of the *for* phrase list header, if any.

The CPU saves this information whenever a task is suspended in favor of a new one, using three cells on the JPDL, the job push-down

list. The mode that attained (direct or indirect) at suspension time is also saved as part of the task status.

Routine X52 controls interstep sequencing after indirect commands and maintains synchronization with the user after direct commands and error commentary. Subroutine S60 is used for cleaning up debris in the user's block and storing all necessary context in the user's block before examining MODE. If MODE equals zero, the CPU returns control to the user without further comment. If the CPU has been told to do a step, routine X55 is used for determining whether it has to be done again. If the CPU has been doing a part, it searches for the next step in the part. If none exists, X55 is used. Otherwise, the CPU honors interruptions from the user and recalls from the SU before transferring control to X54, which serves as a secondary, traffic-control point for restarts when JOSS is told to continue after having returned control to the user for any reason. The user can take any action he wishes when in control, including changing the current step. Accordingly, CSA is set to zero whenever the user deletes or changes the current step, as a signal that a new search must be carried out before the step can be interpreted. A skip-code is also used to control step searches: If zero, the current step is required; otherwise, the step following the current one. A stopping command causes the CPU to make the skip-code nonzero.

Routines X55 and X56 are used for intratask sequencing after a step or part has been done; routine X57 is for mopping up after completion of a task and picking up the interrupted task if one exists.

Routine X55 makes the break-code nonzero to indicate that intratask

Flowchart 8a--ADVANCE TO NEXT COMMAND

LABEL	ACTION	YES	NO
X52	S60: Tidy up the user's block.		
	Is control to be returned to user? (MODE = 0?)		X52.3
X52.2	Set reentry code and return address to indicate awaiting command, and return control to the user.		
X52.3	Anything to do?		X52.2
	Doing a step?	X55	
	Carrying out a *To* command?	X53	
X52.4	Set skip-code = 1 to force P74 to search for the next step in the part.		
	P74: Search for next step.		
	Done?	X55	
X53	Reset skip-code and break-code to zero to indicate *at step*.		
	Make MODE nonzero to indicate indirect mode.		
	X47: Honor interruptions from the user and the SU.		
X54	Make MODE nonzero.		
	P74: Search for current (skip-code = 0) or next (skip-code = 1) step.		
	Done?	X54.1	
	Compose pointer to beginning of step.	X51	
X54.1	Were we doing a part?	X52.4	
	Is step being done repeatedly?	X54.3	
	P72A: Cancel current task; examine suspended task.		
	Type error message (*I can't find step ...*).	X52	
X54.3	Type error message (*I can't find step ... for iteration.*).	X52	

Flowchart 8b--CONTROL REPETITIONS OF STEPS AND PARTS

LABEL	ACTION	YES	NO
X55	Is step or part being done repeatedly?		X57
	Set break-code = 0 to indicate *during*.		
	Set skip-code to zero.		
	Set MODE = mode when task was initiated.		
	P71: Set up for next iteration.		
	Done?	X57	
X56	Set MODE = MODE + 2 to indicate *during*.		
	P73: Search for step or part, taking independent error actions.		
	Set CSA, CSI, and CPI.		
X56.2	Is task governed by a *for* phrase?		X53
	S63: Prepare to make assignment to the iteration variable.		
	P67: Make the assignment if there is enough space.	X53	E3
X57	P72A: Cancel current task; examine status of suspended task.		
X57.1	Anything to do?		X52
	Was this task suspended directly?		X52
	Was it suspended *during* the task?		X57.2
	Was it initiated directly?		X57.2
X57.6	Type *Done. I'm ready to go.*		
	Return control to the user.		
X57.2	Get correct step number for status message.		
	If step has been deleted, force (*altho I can't find it*) to be appended to status message.		
	Use the break-code and the skip-code to compose the appropriate status message (*at step*, *from step*, or *in step*). Type the status report and return control to the user.		

sequencing is going on and resets MODE to its value at the time the task was initiated.

Routine X56 reassigns the variable of iteration, if required, and restarts the interpretation of the required step or part. MODE is incremented by two to indicate that this is going on (as a convenience for the error routines in the event that the required step or part can no longer be found or there is not enough space to carry out the reassignment).

DEMANDING VALUES TO BE INPUT (Demand)

Demand causes JOSS to return control to the user so that he can input a value to be assigned to a letter or indexed letter. The command consists of the word *Demand* followed by a letter or a valid expression for an indexed letter to which the input value is to be assigned. JOSS types the identifying letter or indexed letter followed by an equals sign and then turns control over to the user. When the user releases control, JOSS evaluates the expression typed by the user, makes the assignment, and continues to the next step of the stored program in progress. If the user types nothing at all, JOSS takes this as an implied request that the system suspend whatever it is doing and return control to the user (i.e., as an interruption).

JOSS responds to errors in the expression by commenting on the error and then repeating the command. The user can backspace, overstrike, and strike out characters, and can cancel the line by typing an asterisk as the last character. (JOSS responds to canceled lines by repeating the command.)

No more than one value can be demanded in a single step. Statements such as

Demand a(i), i, b(i).

admit two interpretations: The assignments can be made "simultaneously," or sequentially. Although the statement itself carries a connotation of simultaneity, its execution in terms of JOSS demands and user responses is palpably sequential. In either case, the implementation is not straightforward, since multiple assignments are involved. As in the case of simultaneous-assignment commands,† it was felt that multiple-demand commands were not worth the effort required to implement them properly.

The user can specify an "alias" for the identifying letter or indexed letter. For example,

Demand t as "temperature".

would cause JOSS to use the identification, *temperature*, when requesting the value to be assigned to *t*.

USING THE LONG-TERM FILES

Files are identified by a unique number, which is assigned to the user on application to the proper authorities at RAND, and by a nickname (up to five letters and/or digits) chosen optionally by the user. Any collection of objects that can be deleted constitutes an acceptable item for filing. Items of a file are identified by a

†See p. 88.

number and an optional nickname, both supplied by the user when he requests JOSS to file the item. Items can be recalled into the user's current collection of program and data and can be discarded from the files.

JOSS files away the constituent elements of an item as an exact image of a line or set of lines the user could type to define the elements directly: Values are filed as elided assignment statements and formulas as formula-assignment statements; steps and forms are filed as they would appear if the user had requested JOSS to type all steps and forms. JOSS recalls items from the file a line at a time and treats each line as though it had been typed directly by the user.

Expressions for files and items consist of the word *file* or *item*, followed by the object's identification: an expression for the object's number followed by its nickname, which must be parenthesized or bracketed and set off from the number by spaces.

The user must tell JOSS what file to use for filing operations. JOSS will continue to use the specified file until told to use another one. When requested to type a file's item-list, JOSS will type, in a tabular format, each item's identification, filing date, and other information.

Filing actions can be aborted because of hardware malfunctions and because of lack of storage space on either the disc (during filing) or in the user's core storage (during recalls). JOSS makes no effort to protect the user from prematurely terminated recalls. This is the sole exception to the general rule prohibiting irreversible changes in the user's block until JOSS is certain that the entire action can be carried out without error.

The standard 8-bit representation is used for filing line images, and collections of 8-bit line images are recorded and read as 128-word groups (line images do not extend over group boundaries).

The routines and subroutines for typing objects are used for filing. A switch (UDF1) is used to shunt the line's image to the disc buffer area rather than to the console after the line has been composed.

The routines and subroutines for processing lines are used for recalling an item. For recalling items, the same switch shunts control back to the routines rather than to the user. Lines received from the disc may be direct commands, form declarations, or indirect steps to be stored.

One of the input/output subroutines is used for typing item-lists, an item at a time, from a representation of the list in a 128-word buffer. Because only one user can access the files at a time, the original representation of the list in the disc buffer is copied directly into the user's block, and the disc is released before typing begins. This requires that the user's block be temporarily expanded while the list is being typed. The necessary space is always furnished by the supervisory routines unless no more space is available.

STORAGE MANAGEMENT

In multiuser systems it is prudent to attempt to keep as many users "in core" as possible to reduce the overhead associated with core reorganization and with the swapping of users' blocks between core storage and intermediate drum storage. In JOSS this is achieved

by limiting the size of individual user's blocks. The 2^{12} word limit that was tentatively chosen seemed to strike a balance among (1) the expected number of concurrent users, (2) their expected requirements, and (3) the 2^{14} words of core storage available for users' blocks. This limit still stands. Most users are comparatively "small," while "large" users segment their programs and data and use the disc files to effect program and data overlays (all file actions can be carried out indirectly as well as directly). The distinction between these two broad classes of users is actually a sharp one. Users in the former class rarely butt up against the 2^{12} word limit (except briefly, in the case of errors caused by inadvertent infinite recursions); "large" users, on the other hand, are always wringing out the last drop of space. This fact was overlooked in the design of the CPU. A cavalier attitude toward the use of the last few words available, coupled with a profligate use of general routines for interpreting classes of complex expressions, made life awkward for the space-limited users. Two errors in design were particularly contributory:

1. A general, reentrant routine (P40) for interpreting expressions for left-hand sides was used for interpreting left-hand sides of assignment statements. The routine requires a minimum of three cells for the simplest expressions (e.g., x) and more for lengthier, but equally simple, expressions such as *A(1)*, *M(2,3)*.

2. The assignment of new objects (steps, forms, values, and formulas) to previously used identifiers was carried out without taking account of the space that would be made available by the deletion of the old objects assigned to the corresponding identifiers.

This treatment was justified by the rationalization that the user could always delete an object before redefining it, although even this was sometimes impossible for elements of arrays. The real problem popped to the surface when disc files were added to the system, catalyzed by yet another design decision: to represent elements of a user's program and data on the disc files as exact representations of steps, forms, and assignment statements for values and formulas, and to use the general routines for *Set* and *Let* statements to carry out the assignments when recalling items from the file. It turned out that it was sometimes impossible to recall items from the files, because the CPU was using too much space for bookkeeping by wheeling in ponderous, space-using routines to interpret simple expressions for left-hand sides. The situation was rectified, and the general strategy for the use of space during interpretation is now as follows:

1. Three cells of available space are held back from the user and used only for interpreting *Type*, *Delete*, *Use*, *File*, *Discard*, and *Recall* commands. This is the smallest number of cells required by the routines for interpreting expressions for steps, forms, and elementary operands.
2. Left-hand sides of assignment statements are examined for simplicity before using the general routine for interpreting left-hand sides, and simple left-hand sides are compiled without recourse to available space.
3. Whenever new objects are assigned to previously used identifiers, the space used for storing old objects is taken account of in providing space for new objects.

Whenever available space is exhausted, JOSS tidies up and snaps back to the beginning of the command being interpreted before requesting an additional core block of the SU. If the request is granted, the additional block of 1024 words is incorporated into the available space list, and JOSS begins the interpretation of the interrupted command anew. (This requires that copies of direct commands and commands coming from the disc files be saved for such contingencies.) If no space is available, JOSS notifies the user and then returns control to him. If the incident occurred during an operation involving the disc files, JOSS types

I've run out of space during above.

or

I've run out of space during step....

as a warning to the user that the aborted action might have caused irreversible changes in the user's program, data, or files. The same messages are typed when JOSS runs out of space while preparing to repeat a step or part. Otherwise, JOSS types

Revoked. I ran out of space.

or

I ran out of space at step....

ERROR MESSAGES

Whenever an error (or a system malfunction) is detected, JOSS tries to identify both the error and the point of error† for the user. Errors that cannot be described precisely and unambiguously, and errors that are easily detected by the user, are reported by the brief message *Eh?*. System malfunctions, references to undefined objects, errors in arithmetic evaluations, and many explicit violations of form occur infrequently, and the error messages are explicit. These are best described by the following record of a session with JOSS.

```
Demand x.
Don't give this command directly.

Done.
Don't give this command directly.

Stop.
Don't give this command directly.

To step 1.1.
Don't give this command directly.

1.1 Go.
Do step 1.1.
Error at step 1.1:  Don't give this command indirectly.

1.1 Cancel.
Do step 1.1.
Error at step 1.1:  Don't give this command indirectly.

Delete all.

Type x.
x = ???

i = 1
j = 2

Type A(i, j).
A = ???
```

†See p. 67.

```
A(1, 1) = 11

Type A(i, j).
A(1, 2) = ???

A(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11) = 1
Please limit number of indices to 10.

A(300) = 1
Index value must be integer and |index| ≤ 250.

Type 10*100.
I have an overflow.

Type i/0.
I have a zero divisor.

Type (-i)*(1/2).
I have a negative base to a fractional power.

Type 0*(-2).
I have zero to a negative power.

Type sqrt(-i).
I have a negative argument for sqrt.

Type log(-i).
I have an argument ≤ 0 for log.

Type sin(100).
Please keep |x| < 100 for sin(x) or cos(x).

Type sum[i = 1(-1)10: i].
Illegal set of values for iteration.

Type 123456789.9.
Please limit numbers to 9 significant digits.

Type step -1.
Step number must satisfy 1 ≤ step < 10*9.

Type part 1.1.
Part number must be integer and 1 ≤ part < 10*9.

Type form 1.2.
Form number must be integer and 1 ≤ form < 10*9.

123456789.9 Type i.
Please limit step labels to 9 significant digits.

Type step 1.1.
I can't find step 1.1.
```

```
Type part i.
I can't find part 1.

Type form j.
I can't find form 2.

1.1 Stop.
Do part 1, 2 times.
Stopped by step 1.1.
Delete part 1.
Go.
Error during above:  I can't find part 1 for iteration.

1.1 Stop.
Do step 1.1, 2 times.
Stopped by step 1.1.
Delete part 1.
Go.
Error during above:  I can't find step 1.1 for iteration.

Cancel.
Go.
I have nothing to do.

Let f(a, b, c, d, e, f, g, h, i, j, k) = a formula with too many parameters.
Please limit number of parameters to 10.

Form 1:
___

Type i, j in form 1.
I have too many values for the form.

Type 123456 in form 1.
I can't express value in your form.

A(i) = 1
Type A in form 1.
I need individual values for a form.

Form 1:
___.........

Type i, j in form 1.
I can't make out your fields in the form.

A transmission error was forced while typing this line.
A transmission##error was forced while typing this line.
Sorry.  Say again:

x = 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1.
Please limit lines to 78 units (check margin stops). Say again:
```

```
1.1 Type i.
Do part 1, 1.2 times.
Number-of-times must be integer and ≥ 0.

x = i = 1.
Please use parens or brackets to set-off ambiguous equals signs.

Type (i = 2: i; i = 3: i + 1).
(i = 2: i; i = 3: i + 1) = ???

Type first(i = 1(1)10: i < 0).
first(i = 1(1)10: i < 0) = ???

Use file 4 (S618).
I can't find the required file.

Use file 4000 (S6180).
File number must be positive integer ≤ 2750.

Use file 4 (S.6180).
Please limit ID's to 5 letters and/or digits.

Use file 4 (S6180).
Roger.

Recall item 120.
Item number must be positive integer ≤ 25.

Recall item 2 (A.m1).
Please limit ID's to 5 letters and/or digits.

Recall item 2 (Am1).
I can't find the required item.

File all as item 2 (Amort).
Please discard the item or use a new item number.
```

Other messages that may be typed by JOSS are

Something's wrong. I can't access the files.

I've run out of disc space.

Something's wrong. Try again.

The last message is typed by the CPU on finding something awry in its records or receiving contradictory status reports from the IOU during actions on the file.

IV. REPRISE

The PDP-6 version of JOSS was initially conceived as a recasting of the original JOHNNIAC version into faster and more reliable hardware. Core storage for individual users' blocks was to be increased at least fourfold; facilities for filing programs and data were to be added; the consoles were to be redesigned using more modern typewriters and electronics; and the new system was to be agile enough to service at least sixty consoles simultaneously without noticeable interactive degradation. These modifications and extensions were sufficient in themselves to cause major changes in the profile of the users' community. In brief, users' requirements expanded to take advantage of the increased resources of JOSS. The fourfold increase in the size of users' blocks attracted users who wrote larger programs than could have been accommodated by the original system. The availability of the disc files for long-term storage generated a class of users who wrote very large, segmented programs, and used the files to carry out program and data overlays. To cater to the latter class, JOSS eventually had to allow filing actions to be carried out indirectly as well as directly. The behavior of the system was affected in two, fairly predictable ways. First, the average computing speed of JOSS turned out to be less than originally estimated: Because large programs[†] generally run longer than small ones, any increase in the percentage of large programs causes a shift in the computation/interaction balance--the more programs that are "compute bound," the slower the processing of individual computational

[†]More precisely, programs that use a large amount of data.

requests. Second, the indirect use of the files for program and data overlays often becomes so heavy as to cause irritating delays to the casual users who are accessing the files infrequently.

Further, in the face of larger programs using more data, the set of 52 single-letter identifiers becomes restrictive--not in size, but in mnemonic richness. The necessity for extending the identifier set in some regularized manner was recognized fairly early. Postscripted letters (e.g., *A2*, *m5*) and primed letters (e.g., a', a'') were considered, but no clear-cut decision could be reached, and the 52 letter set remains. The current version of the CPU could be modified, with some care, to accommodate postscripted or primed letters as identifiers. (A richer designatory capability is probably neither required by nor suitable for most JOSS users.) However, there is a growing class of users who are writing general programs designed to be used by people having no familiarity with JOSS, from whom--in the estimation of the program designers--JOSS should be sealed off. The capability for demanding input values under an alias was added to JOSS to partially satisfy such requirements. This, together with the capability for typing in forms, allows users to seal off their programs to some extent. However, the error messages remain: Users cannot be completely isolated from JOSS until provisions for sensing and fielding error conditions are provided in the language.

Neither block structures nor subprogram structures are offered by JOSS; that is, users have no way of isolating collections of program and data so that step numbers and other identifiers used within a collection can be treated by JOSS as being local to the collection and, therefore, reusable in coexisting collections without conflict. It is difficult to assess the effects of this deliberate omission, since

formulas, conditional expressions, and iterative functions seem to go a long way toward alleviating the problem (even making it nonexistent) for all but the "larger" users. The proper handling of block structures in an interactive environment is considerably more difficult than the relatively direct task of recognizing and handling hierarchies of identifiers (in fact, the latter mechanism is already incorporated in the current CPU). The difficulty lies in the fact that JOSS would have to be prepared to keep track of two distinct hierarchies: a dynamic execution-hierarchy for carrying out stored programs and subprograms, and a dynamic editing-hierarchy for modifying programs, subprograms, and data. The questions that must be answered are

1. How freely may users move within and between blocks and subprograms for purposes of editing and computation?
2. How freely may they modify such structures during execution?
3. How may they express their requests?
4. What effect would such facilities have on the general behavior of the system? How would the apparent simplicity of JOSS be affected?

There is always the temptation to simplify the problem by (1) allowing only a single-level block structure (i.e., a "main" block and a set of nonoverlapping subblocks); (2) having JOSS cancel all suspended and ongoing tasks whenever the user "opened" a subblock for editing tasks; and (3) not allowing direct, parenthetical *Do*'s (at least of subblocks). Even this simplifying tack does not legislate the problem out of existence. One must still furnish JOSS (and the user) with a

simple notation for distinguishing between a letter used as a local identifier within a block and the same letter used globally (either as an alias for a global identifier or as a parameter "passed down" to a subprogram).† Having JOSS snap back to the "main" program on errors inside subblocks and type, for example, *I have an error in sub-block...*, would leave the user stranded and informationless in such situations, and is completely unreasonable for handling interruptions and *Stop* commands. The general question is too subtle to be considered in this study and will not be mentioned further; it is, however, far too important a question to be swept under the rug for long.

Tripling the number of in-house consoles and speeding up the system made JOSS more available. The increased availability, coupled with file storage, resulted in the development and extensive use of a rash of multiconsole games whose participants communicate via common files. Such exercises make heavy use of JOSS's scarcest resource, consoles, and tend to decrease the system's availability. Further, the designers of such games now find that they require the capability for interconsole signaling and communication. The ability to access and reset the value associated with *timer* was added to the language as an ad hoc (and poorly thought out) solution to the signaling problem. Most users of *timer*, however, now do so to generate pseudo-random, decimal digits (the least significant digit of *timer*).

In summary, the current version of JOSS differs markedly from the original version, despite the fact that it appears to be a direct,

---

†Current notations for so-called symbol qualification are crippled by the limited character sets available, and most are incomplete.

innocent extension of the original. Increasing the resources and facilities provided by JOSS changed the behavior of users (and of JOSS itself), sometimes in predictable ways, often in completely unanticipated ways. Many of the changes can be attributed mainly to the increased storage and file capability of the current version rather than to extensions of the language--JOSS, like most systems, cannot be measured by its language alone. On the other hand, although most of the extensions were chosen so as not to intrude on "basic" JOSS, casual users can now get into trouble, often in subtle ways. For example, a user can inadvertently wipe out an array of values by assigning a scalar value to the letter identifying the array. In the original system, such blunders were probably infrequent, because programs were small and the user could keep track of his identifiers. Moreover, the effects were not drastic, because of the limited storage provided by the original system. Currently, as users write larger programs and strain the identifier set, the probability of such blunders increases, and--with increased storage available--the effects may be catastrophic. JOSS could easily be made to demand that the user explicitly delete arrays (and probably formulas) before reassigning the letter of identification to another type of object by treating such reassignments as errors. Unfortunately, this conflicts with the mechanism used to recall items from the disc files: Formulas and values are recorded on the files as *Let* statements and elided assignment statements, so that the user would have to take extreme care that recalls of items from his file not be aborted by reassignments of letters to different objects.

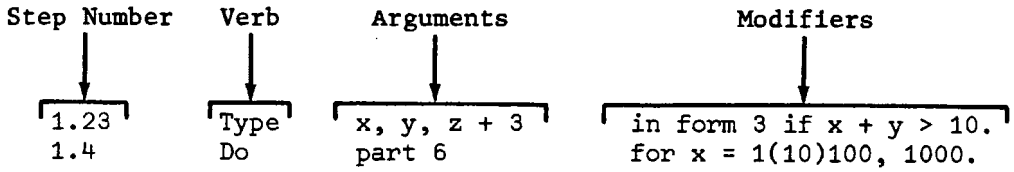
Examples like the above are legion in JOSS and, undoubtedly, in

other changing systems. Many carefully considered design points turn out badly, while some hasty, off-the-cuff decisions turn out to be the right ones; errors of commission are matched by errors of omission, and little turns out to be just right. The situation is best summed up in the words of Congreve:

> *Thus grief still treads upon the heels of pleasure;*
> *Married in haste, we may repent at leisure.*
> *Some by experience find those words misplaced*
> *At leisure married, they repent in haste.*

Appendix

SUMMARY OF JOSS LANGUAGE



DIRECT COMMAND: Step number not present; command is executed immediately.

STORED COMMAND: Step number present; command is stored in order of step number.

STEP: A stored command; step number is limited to 9-digit number ≥ 1.

PART: A group of steps whose step numbers have the same integral part.

FORM: A pictorial specification of literal information and fields to be filled with values, for formal output. Fields are indicated by strings of underscores (with optional decimal point) or strings of periods (for a tabular form of scientific notation):

Form 7:

I = __.___ amps. V = ... volts

NUMBERS: Nine significant digits; $10^{-99} \leq |value| \leq 9.99999999 \cdot 10^{99}$ or *value* = 0.

SYMBOLS: Single-letter identifiers, upper- or lower-case. May identify decimal values, logical values (true, false), formulas, and arrays of values.

FORMULAS: May have up to ten formal parameters (distinct letters) or none (see *Let*).

ARRAYS: May have up to ten, integer-valued indices in the range [-250, 250].

ARITHMETIC: Addition (+), subtraction (-), multiplication (•), division (/), exponentiation (*), and square root (*sqrt*) give true results rounded to nine significant digits. Zero is substituted on underflow.

RELATIONS: < > ≤ ≥ = ≠ (extended relations permitted; e.g., a < b ≤ c).

LOGIC: and or not

GROUPERS: ( ) [ ] used interchangeably, in pairs.

IMPLIED GROUPINGS:

3 + 1/2 + 1/4•5 → (3 + 1/2) + (1/4)•5

-2*3•4 - 5 → [-(2^3)•4] - 5

2*3*4 → (2^3)^4

a or b and not c or d → a or [b and (not c)] or d

Set — Assigns a value. *Set* and final period may be omitted on direct commands.

```
Set x = 3.
Set a(5, x) = y + 3•z - x*2.
```

Let — Defines a *formula* of up to ten parameters.

```
Let f(x, y) = x*2 + 10•x - 6•y.
Let h = (b - a)/2.
Let D(f, x) = [f(x + d) - f(x)]/d.
```

Delete — Erases values, parts, steps, forms, formulas.

```
Delete x, part 3, all forms.
Delete all values, all formulas.
Delete all.
```

Type — Types quoted text, or types blank lines (_), values, parts, steps, etc.

```
Type "The quick brown fox".
Type x + 3, D(sin, 0), _, all steps.
Type all.
```

Demand — Types identification and equals sign, then waits for user to input value. Treats blank input lines as interruptions.

```
Demand a(3, i).
Demand t as "temperature".
```

Do — Initiates execution of step or part (step by step beginning at first step of part), repeatedly if modified by a *for* or a *times* phrase.

```
Do part 6 for x = .1, 2(2)10, 100•a + 2•b.
```

(Do ...) — Interprets direct *Do* as a stored *Do* (i.e., does not cancel before execution), but returns to user when done.

```
(Do part 3.)
```

Done — Terminates execution of current *Do* for current repetition.

Quit — Terminates execution of current *Do* for all repetitions.

Cancel — Terminates execution of all *Do*'s.

(Cancel.) — Direct only! Terminates execution of last *(Do ...)*.

To — Alters step-by-step sequencing. Continues at indicated part or step.

```
To step 3.5.
```

Stop — Suspends step-by-step execution to await instructions from user.

Go — Continues execution after *interrupt*,† error message, or *Stop* command.

---

†Pressing the INTERRUPT button on the console.

Page — Advances paper to next page.

Line — Types a blank line.

Form — Identifies (by an integer) and stores the form typed on the next input line.
Form 3:
x = ____.___ y = ____.___ z = ...

Use — Prepares to use indicated file for all subsequent file actions.
Use file 107 (*code*).†

File — Stores an item in the file.
File part 3, x, z as item 7 (*code*).

Recall — Retrieves an item from the file.
Recall item 3 (*code*).

Discard — Erases an item from the file.
Discard item 3 (*code*).

if — Modifies any command. JOSS carries out command if condition holds.
Type x if $0 \leq x < 5$.
Set y = 3 if $x \leq 10$ and $x \cdot y = 10$.

for — Modifies *Do* only. JOSS executes part or step repeatedly for specified set of values.
Do part 3 for x = 1(1)10(10)100, 1000.
Do step 1.2 for x = .01, .03, .1(a)b.

times — Modifies *Do* only. JOSS executes part or step specified number of times.
Do part 4, 43 times.
Do step 7.3, n + 1 times.

in form — Modifies *Type* only. JOSS types values in fields of specified form.
Type x, y, z*2 in form 3.

sparse — Modifies JOSS's treatment of missing array elements. JOSS treats them as zeros and they require no storage.
Let A be sparse.

item-list — A summary of items in the file being used.
Type item-list.

time — Time of day at RAND.
Type time.

users — Number of consoles being serviced by JOSS at the moment.
Type users.

size — Number of storage units currently occupied by user's program and data; about 1900 are available.

timer — Time in minutes and hundredths since log-on or last *Reset timer*.

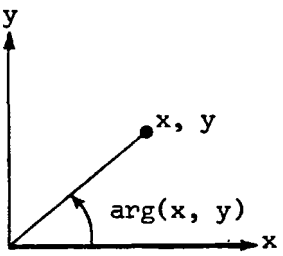
$ — Current line number (1-54) on typed page.

---

†Codes, if used, are composed of no more than 5 letters and/or digits; no distinction between upper- and lower-case.

sqrt(x)	*square root, x ≥ 0*
sin(x), cos(x)	*|x in radians| < 100*
log(x)	*natural log, x > 0*
exp(x)	e^x
arg(x, y)	*angle (see figure) in radians,* arg(0, 0) = 0.

sgn(x)	-1, 0, +1 for x < 0, x = 0, x > 0	
ip(x)	*integer part*	ip(100.5) = 100
fp(x)	*fraction part*	fp(100.5) = .5
dp(x)	*digit part*	dp(100.5) = 1.005
xp(x)	*exponent part*	xp(100.5) = 2
|x|	*absolute value for decimal values*	
	|true| = 1	
	|false| = 0	



sum[i = a(b)c: f(i)]	sum(x, y, z + 10)
prod[i = a(b)c: f(i)]	prod(x, y, z + 10)
min[i = a(b)c: f(i)]	min(x, y, z + 10)
max[i = a(b)c: f(i)]	max(x, y, z + 10)
conj[i = a(b)c: P(i)]	conj(x < y ≤ z, y > 3, P)
disj[i = a(b)c: P(i)]	disj(x < y ≤ z, y > 3, P)

first[i = a(b)c: P(i)] *gives first value of* i *for which* P(i) *is true*

tv(P) *= 0 if* P *is false, = 1 if* P *is true,*
= false if P *is zero, = true if* P *is nonzero*

$$(P_1: E_1; P_2: E_2; \ldots; E_n)$$

where: P_i **are expressions for logical values,**

means: **If** P_1 **is true use** E_1**, otherwise if** P_2 **is true use** $E_2, \ldots$**, otherwise use** E_n**.**

Set x = (0 < y ≤ 5: 0; y < 10: y*2; 5).
Let P(x) = [x = 0: 1; prod(i = 1(1)x: i)].

JOSS BIBLIOGRAPHY

PUBLICATIONS OF CURRENT INTEREST

Baker, C. L., *JOSS: Console Design*, The RAND Corporation, RM-5218-PR, February 1967.

-----, *JOSS: Introduction to a Helpful Assistant*, The RAND Corporation, RM-5058-PR, July 1966.

-----, *JOSS: Rubrics*, The RAND Corporation, P-3560, March 1967.

Bryan, G. E., *JOSS: Accounting and Performance Measurement*, The RAND Corporation, RM-5217-PR, June 1967.

-----, *JOSS: Assembly Listing of the Supervisor*, The RAND Corporation, RM-5437-PR, August 1967.

-----, *JOSS: Introduction to the System Implementation*, The RAND Corporation, P-3486, November 1966; also published by the Digital Equipment Computer Users Society, *DECUS Proceedings*, Fall 1966.

-----, *JOSS: 20,000 Hours at the Console--A Statistical Summary*, The RAND Corporation, RM-5359-PR, August 1967.

-----, *JOSS: User Scheduling and Resource Allocation*, The RAND Corporation, RM-5216-PR, January 1967.

Bryan, G. E., and E. W. Paxson, *The JOSS Notebook*, The RAND Corporation, RM-5367-PR, August 1967.

Bryan, G. E., and J. W. Smith, *JOSS Language (Aperçu and Précis, Pocket Précis, Poster Précis)*, The RAND Corporation, RM-5377-PR, August 1967.

Gimble, E. P., *JOSS: Problem Solving for Engineers*, The RAND Corporation, RM-5322-PR, May 1967.

Greenwald, I. D., *JOSS: Arithmetic and Function Evaluation Routines*, The RAND Corporation, RM-5028-PR, September 1966.

-----, *JOSS: Console Service Routines (The Distributor)*, The RAND Corporation, RM-5044-PR, September 1966.

-----, *JOSS: Disc File System*, The RAND Corporation, RM-5257-PR, February 1967.

Marks, S. L., and G. W. Armerding, *The JOSS Primer*, The RAND Corporation, RM-5220-PR, August 1967.

PUBLICATIONS OF HISTORICAL INTEREST

Baker, C. L., *JOSS: Scenario of a Filmed Report*, The RAND Corporation, RM-4162-PR, June 1964.

"The JOSS System: Time-Sharing at RAND," *Datamation*, Vol. 10, No. 11, November 1964, pp. 32-36. (This article is based on RM-4162-PR above.)

Shaw, J. C., *JOSS: Conversations with the Johnniac Open-Shop System*, The RAND Corporation, P-3146, May 1965.

-----, *JOSS: A Designer's View of an Experimental On-Line Computing System*, The RAND Corporation, P-2922, August 1964; also published in *AFIPS Conference Proceedings* (1964 FJCC), Vol. 26, Spartan Books, Baltimore, Maryland, 1964, pp. 455-464.

-----, *JOSS: Examples of the Use of an Experimental On-Line Computing Service*, The RAND Corporation, P-3131, April 1965.

-----, *JOSS: Experience with an Experimental Computing Service for Users at Remote Typewriter Consoles*, The RAND Corporation, P-3149, May 1965.