Home
Back to Memories
Back to Thoughts

REMOTE COMPUTING-AN EXPERIMENTAL SYSTEM

Part 2: Internal Design

J.M. Keller, E. C. Strum, and G.H. Yang
Development Laboratory, Data System.s Division
IBM Corporation
New York, N. Y.

INTRODUCTION

This is the second of two papers dealing with the experimental remote-computing system. Part 1 described the system as viewed by a user who is unaware that he is jointly sharing the central computer with numerous other users. This paper (Part 2) describes the internal design of the system; with attention focused on those featu res which are of general interest and applicable to the design of other programming systems.

This paper is introduced by a description of the over-all control structure and data organization.. Each of the principal subsystems is then described. The paper concludes with some remarks regarding possible extended applications. An appendix describes in some detail the algorithms used in the decomposition/recomposition of arithmetic expressions.

OBJECTIVES

An operating system servicing numerous online users must meet certain design objectives that might be regarded as secondary or even unnecessary in conventional operating systems or compilers. But these objectives become paramount when the psychological and practical effects of sustained, immediate access to a computer are considered. Thus primary attention must be given to attaining:

1. Immediate error diagnostics;
2. Program alteration without recompiling;
3. Extensive symbolic debugging aids;
4. Ready availability of the source version of the user program ;
5. *A* user program that is:
   a. dynamically relocatable,
   b. easiiy interruptibie, and
   c. storage protected.

SYSTEM ORGANIZATION

Program.s

The experimental remote-computing program is divided into three major system areas (Figure 2.1):

1. The *Scheduler,* which is responsible for maintaining awareness of the total system status and for ordering and assigning tasks to the other system parts;
2. The *Process Control system,* consisting of the Translator, which reduces the user's input statements to an equivalent internal form (see below); the Interpreter, which executes this internal form; and the Process Control program, which regulates these two subsystems on the local level ;
3. The *1/0 Control system,* which is responsible for monitoring and operating all

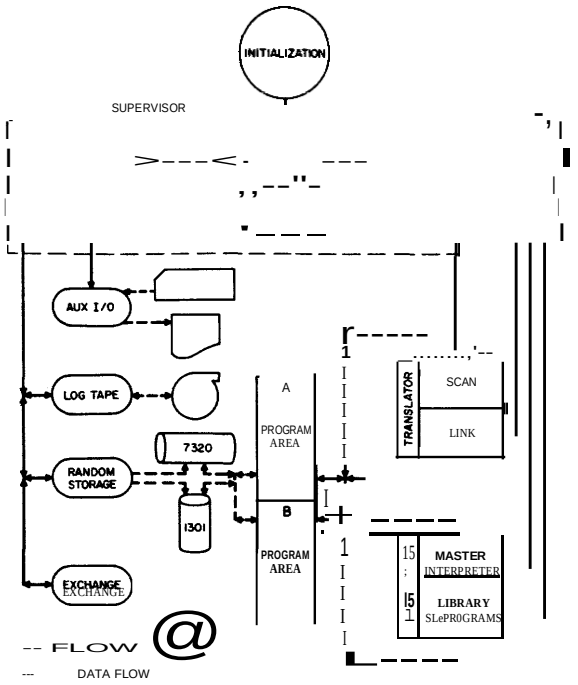

Figure 2.1. General Block Diagram of System.

1/0 attachments, including the communications exchange.

Data Organization

At the system level, there are three principal data constructs:

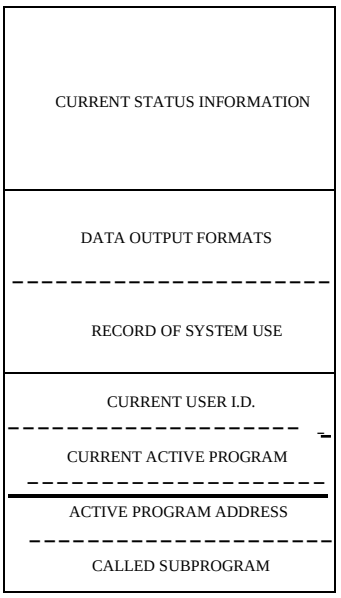

Figure 2.2. Terminal Header.

The Terminal Header

For each terminal in the system, there is a Terminal Header record (Figure 2.2) containing the following information:

1. Current status:
   a. operating mode, i.e., Command or Program (Rf. Part 1),
   b. terminal status, i.e., I/0 wait, busy, or dormant,
   c. control information, i.e., should the system interrupt automatic status (execution) and return to manual status (statement entry) or continue automatic status,
   d. type of terminal component active,
   e. terminal ID,
   f. storage allocation block;
2. Header information for Command mode execution:
   a. formats for data output,
   b. system use records;
3. Temporary locations for random storage access:
   a. current user identification,
   b. name of current active program,
   c. location of active program for this terminal,
   d. name or location of subprogram called by current program.

The Master Block

For each statement in the language (Rf. Part 1), there is a Master Block record (Figure 2.3) containing the following information :

1. A statement type identifier;
2. A statement class identifier;
3. The symbolic, external statement identifier with associated control characters for recognizing the statement name on input and for recreating it on output;
4. Various indicators which denote intrinsic statement characteristics for checking purposes;
5. Addresses for transfers of control to the various major system routines, e.g., Translator, Inter preter ętc.

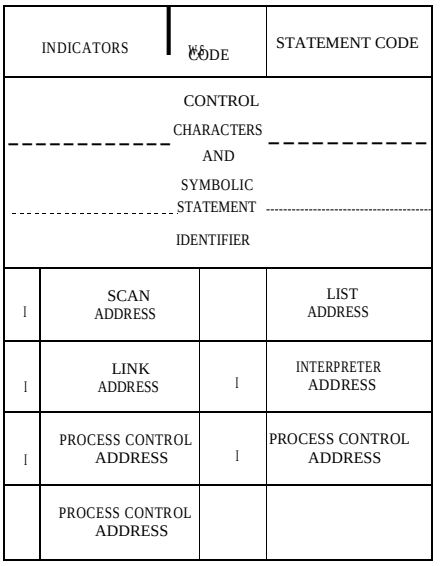

Figure 2.3. Master Block Record.

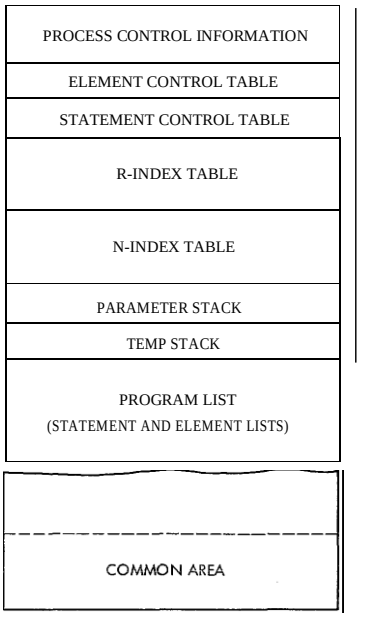

Figure 2.4. User-Program Layout.

The Master Block is used either as a dictionary, when information concerning the statement is needed, or as a switching center, when control flow within the system is dependent upon the statement type. A single record for both of these activities provides considerable flexibility in adding new statements, in modifying control conditions, and in making basic system modifications.

The User Program Layout

For the entire system there are two large, fixed areas (Figure 2.4) reserved for occupation of the various active user programs, Programs brought into these areas are relocated under program control; all I/0 to and from these areas is overlapped. The duration of occupancy is determined either by overstepping a time limit or by the occurrence of one of several specific conditions (see following section on user-program organization).

The layout of the user program is divided into two parts :

1. The statement and element records (see following section) which comprise the user program ; and
2. The header.

The header is further subdivided into two parts:

1. List control and other controls for the program records (see fo!!owing section) ; and
2. Control words used by the Process Control program to keep track of program status.

USER-PROGRAM ORGANIZATION

The User Program

The user's source-pTogram statements are mapped into equivalent internal records, which are classified and controlled by list structures. These records and their controlling elements constitute the user's program (see Figure 2.4). Every statement of the user's program is reduced to an individual statement record; and every element (name or label) is reduced to an individual element record. These records are inserted and chained on lists in the program area in the order of their appearance and creation. Control is maintained through tables of list-control words in the header portion of the user program. All addresses in the user pro- gram are relative to its base in order to facili- tate relocation.

Records

Element Records

An element in the source program is defined as a label, constant, variable, array, or function name. Every source element maps into a fixed-length, internal element record (see Figure 2.5) containing the following information:

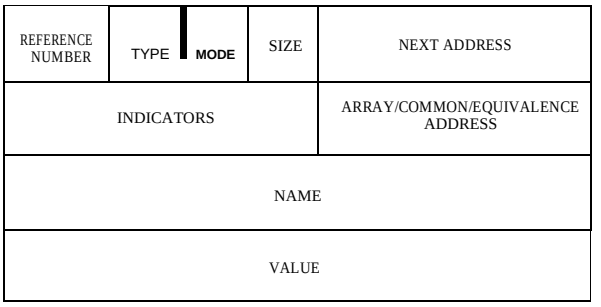

Figure 2.5. Element Record.

1. Reference number-a unique internal numeric identifier, assigned by the system. It is used for all internal referencing by the system.
2. Type-denotes the type of element, i.e., label, constant, variable, array, function.
3. Mode-denotes the mode, real or integer, of elements referring to numeric quantities.
4. Indicators-contains properties attributed to the element by declarative statements and/or execution. These include storage-allocation, and indications of element usage at object time.
5. Name-the external alphanumeric identifier.
6. Value-either the numeric value of the element or supplemental information for an array or function.
7. Next address-address of the next element record.
8. Array COMMON EQUIVALENCE-address of the value, if the element is in COMMON or is an array; or an offset address, if the element is equated to an array.

Statement Records

Every source statement maps into a variable-length, internal statement record (see Figure 2.6), which contains in coded form all information present in the source statement. Each record begins with two standard words containing the foHowing information :

ALTER NUMBER		SIZE	NEXT ADDRESS
STATEMENT CODE	INDICATORS	LABEL	NEXT CLASS ADDRESS
R(C)			
*	R(A)		R(B)
PARAMETER OPERATOR	R(D)		
FUNCTION OPERATOR	R(SQRD		
I	R(C)		temp
+	temp		temp
	R(C)		temp

C = A.*B + C/SQRT(D)

Figure 2.6. Statement Record.

1. Alter number-a unique internal numeric identifier assigned by the system. It denotes the position of the statement relative to all others in the program; it is referenced by the user when modifying the program, manually requesting information, or starting execution.
2. Statement code-identifier of the particular statement type.
3. Indicators-reflects usage of the statement during execution.
4. Label-refers to the associated external statement number, if any.
5. Next address-address of the next statement record.
6. Next class address-address of the next statement record of the same type.

The remainder of each statement record contains one or more words. Their number and content depend on the particular statement type. For example, an arithmetic-statement record contains the macro representation of

the translated expression, while a DO statement contains references to the indexing parameters.

Lists

The objective of providing for alteration of individual statements was the deciding factor in determining the internal record organization. The conventional table-oriented approach appeared much less attractive than the classification of records by lists.[1,2,3,4,5,6]

Most compilers use tables to record information necessary for referencing and validating data usage and control flow. In this system, the same information is kept in the statement and element records. However, organizing these records on lists allows for increased flexibility in the compiling system.[7,8,9,10] For example, deletion and insertion of statements for program modification is easily provided. Time-consuming recompilations become completely unnecessary as a result of this altering provision. In addition, errors resulting from improper control flow and from invalid variable references can be diagnosed earlier in the compilation process than is common with conventional compilers.

Element Lists

Each element record is chained onto one of 26 element lists, each list consisting of all those element records whose symbolic names have the same initial letter. Element records within each list are ordered alphabetically by symbolic name. There are two additional lists which link numeric elements as either integer or real constants. This set of element lists provides two significant features :

1. The symbol look-up is more efficient since only the set of symbols with the same initial letter are considered;
2. Fully alphabetized symbolic cross-reference listings and memory dumps are easily provided.

Statement Lists

Each statement record is chained onto two lists:

1. The entry list consisting of all statement records in source sequence (i.e., ordered by "alter number");
2. One of the class lists, consisting of all statements of a particular class (e.g., arithmetic, control, DO, I/O, allocation declarations, etc.).

The entry list is used both to control execution sequence and to provide the proper ordering when the source program is reconstructed from the internal form.

The class lists are extremely useful in performing checking operations (e.g., checking DO loops for proper nesting and control transfers).

List Control

Every list is controlled by a single control word pointing to the first and last records. For the statement lists there is a small table of control words for statement control (see Figure 2.4). Another similar table controls the element lists. In addition there is also a master table controlling the symbolic names of reserved system symbols: library functions (e.g., SIN, SORT, etc.); built-in functions (ABS, FLOAT, etc.); and system subroutines (DUMP, EXIT, etc.).

Addressing

Two tables exist for control of the element and label identifiers (see Figure 2.4). These are the R-index, or internal-identifier reference table, and the N-index, or numeric-label table. For every element that appears in a program, an entry for its internal identifier, R, is made in the R-index table; similarly, for every statement label, N, an entry is made in the N-index table.

Every element or statement in the program can be accessed in an "asociative" manner by sequentially searching the lists until a match is found for the requested symbolic name or alter number. Each element or labeled statement can also be located in a "direct-look-at" manner [11] by using the internal identifier for the element or label as an entry to the R- or N-index table. Thus the flexibility of associative list searching and the efficiency of direct element fetching are both incorporated in the system.

FUNCTIONAL DESCRIPTION

The Scheduler

The purpose of any real-time, multiprogramming supervisory program is to synchronize, control, and monitor system operation.[12,13,14] The program is responsible for determining what things are to be done, and by whom, to what, where, and when each is to be done. It has the duty of maximizing system throughput and ensuring reliable operation, and, in this case, of maintaining rapid and level response times at the terminal consoles.

At the nucleus of this supervisory structure (see Fig.ure 2.1) is the Scheduler, [15,16] which controls the:

1. Process Control program, which in turn directs the processor routines that translate and execute user programs; and
2. I/O Control program, which coordinates the communications exchange, random storage devices, tape units, reader, and on-line printer.

The Scheduler performs continual sequential sampling of the subsidiary subsystems and maintains pertinent status data in the terminal headers. When data has been received from the terminal, the Scheduler examines the terminal header and decides whether to transmit a request to the random-storage I/O queue to fetch the user program (Program mode), or, if no program is required, save the data in a to-be-processed queue (Command mode).

In either event, the Scheduler passes to the Process Control program all information necessary for processing the input message--such as locations of the terminal and program headers, the location of the input message, and the operating mode of the terminal.

Even if no message has been received for a given terminal, its active program will be fetched from random storage and the Process Control program entered, if the terminal header shows that the program is in the automatic state (i.e., in the process of execution). After each return from the Process Control program under this condition, the Scheduler must determine whether the automatic state should be terminated.

There are two kinds of termination : temporary, and return-to-manual. Temporary interruption frees the system for use by another terminal and may occur for the following reasons:

1. The allotted time interval has expired;
2. Input data is requested:
3. Output buff er is filled;
4. An external subprogram is invoked.

The return to manual status occurs when:

1. .An error condition occurs;
2. A STOP or PAUSE statement is executed;
3. The end of the program is encountered;
4. The user requests an interrupt from the terminal.

The Process Control Program

The Process Control program (see Figure 2.7a) accepts information from the Scheduler and coordinates the activities of the processor programs. All of the appropriate Process Control routines and service routines must be initialized (1) to process the terminal header if the termnial is in the Command mode, or (2) to process various parts of the program header and list if in the Program mode. The Process Control program maintains an action code in

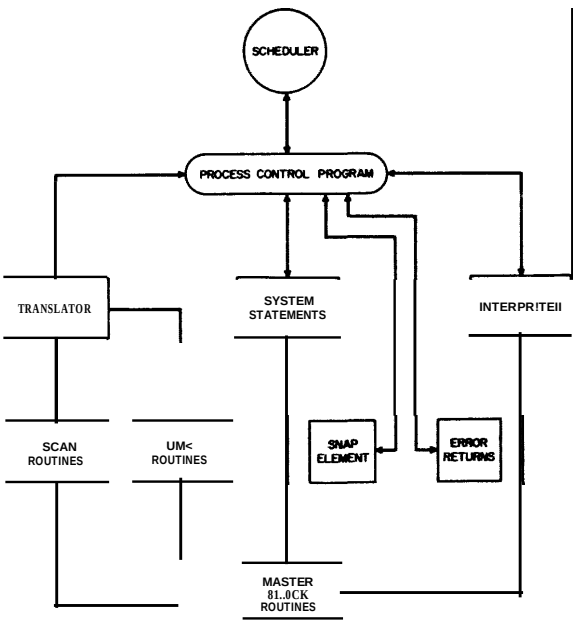

Figure 2.7a. General Diagram of Process Control Flow

each header to determine the next task to be performed upon the user program. The Process Control program may require the input statement just received to be scanned by the translator; it may require continuation of execution, of a DUMP, or of a LIST. The user program may be in the ALTER mode; it may be in the process of being tested for certain conditions which may prohibit further execution. Data for an input statement may be awaited or output of multiple-error messages may be in effect.

When the user program is in the manual mode, the Process Control program has the responsibility of examining the process codes returned from the translator and of taking the necessary action. When the user program is in the automatic mode, execution may be temporarily halted and, in some cases, the program may be returned to manual-mode status (see above). When a subprogram "call" is made, execution halts until the next cycle for this terminal. At that time, the called subprogram becomes the user's active program and is brought into memory in place of the calling program. When a RETURN is effected or if an error occurs, the calling program is reactivated.

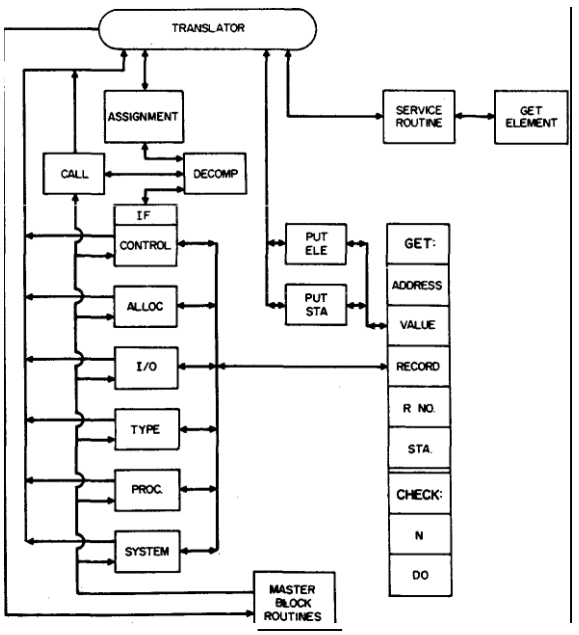

Figure 2.7b. General Diagram of Process Control Flow -**Translator.**

In order that the user may always be aware of the status of his program, condition codes are printed at the terminal whenever a change of status occurs. He is notified when input (a statement or data) is requested; when an error occurs; when and why execution was terminated; when a system statement (see Figure 2.7d) occurs (e.g., DUMP, INDEX, TRACE); and, optionally, when a subprogram call is made. When execution runs off the end of a program, and when a STOP or PAUSE is encountered, he is informed that his request for interruption of execution has been recognized. In short, the Process Control system always knows what is happening in the user program, and continually keeps the user informed of the status of his job. The objective is to provide the remote user with a more complete awareness of his program's status than is obtainable at a conventional computer console.

The Translator

Scan Routines

The Translator (see Figure 2.7b) is responsible for transforming the source-language program to the internal form.[17.18.19] (See Figures 2.5 and 2.6.) A preliminary scan is first used to identify arithmetic statements. For all other statements, the statement operator is collected and used to reference (via a Master Block routine) the corresponding master record. Control then passes to the translation routine for the particular statement type, e.g., GOTO, RETURN, PRINT, DIMENSION, etc.,

Every statement's decomposition goes through the same basic phases to form element and statement records. These involve the use of several service routines to collect the element name, find its record in a list or create a new record, and validate the statement and element usage.

As each element in a statement is collected, a search is made to determine if it has previously appeared in the program. If the element has been previously used in the same statement, the record appears on a current element working list; otherwise, it may be found on the element list in the user's program or, alternately, on the master list of reserved and

system names. If no record is found, then the element is new to the program, and a record is created for it and placed on the current working list. The information put into the record depends on the variable itself and the kind of statement it appears in. The mode indicators for an element depend either on its initial letter or on its appearance in an INTEGER or REAL declarative statement. The type and variable indicators depend on the statement type. A variable appearing in a storage-allocation statement is nagged according to its declaration as an array, common, or equated variable.

If all source elements have previously appeared, no new element records result from the translation of a statement. However, a statement record must always be created. The Master Block record for the statement type provides the statement code. Statements that involve a list of variable, such as DIMENSION, EQUIVALENCE, or COMMON, contain a count of the variables used followed by the R-number of the variables.

Statements which involve a list of labels, such as:

GO TO (5, 6, 7, 8), I or IF (J-5) 12, 3, 12

contain an item count followed by the numeric labels. If the execution of a statement will change the value of a variable, the identifier of that variable is placed in a special field in the statement (see Figure 2.6).

Statements containing arithmetic expressions or input/output lists involve specialized decomposition routines. The master Translator passes to both these routines essentially the same input: a string of words, each word containing either an operator or an R-number. The decomposition routine transforms these elements into an ordered set of arithmetic macros consisting of an operator and two operands in every word. These macros are then returned to the master Translator and added to the statement record (see Figure 2.6). The input/output list-decomposition routine also returns a macro set of executable operations, (A detailed description of the arithmetic decomposition is contained in Appendix I.)

Scan Diagnostics

Throughout the translation-scan phase, checking occurs for syntax and composition-type errors. Illegal statement operators and invalid statement forms are detected early in the translation. Lack of a label on a FORMAT statement or the presence of a label on a declarative (where control may not flow) violate the definition of the statement type. Illegal uses of variables, such . as a simple variable name followed by a parenthesis, are detected by testing indicator bits in the element records. The same checking of element records is used to detect mixed-mode errors in the arithmetic expressions. The number of subscripts following an array name is checked for agreement with the number declared in the DIMENSION statement for that variable. In general, the Translator detects all syntactic errors which are within the context of a single statement and those semantic errors which occur in the use of the elements in the statement.

Link Routines

If the statement has no errors, the Process Control program decides whether to save the statement record and its related element records as part of the user'i program. A statement record is either added to the end of the entry and class lists or, if the .ALTER mode is active, is inserted somewhere into these lists. To accomplish this linking, space for the new record is found, and the address of this area, relative to the program area base, is inserted as the "next" address in the preceding record on the list.

If the statement record is successfully put into the program area, the element records are linked to their respective lists. Every new element record also causes its address (relative to the user program base) to be entered into the R-index table.

Link Diagnostics

Before a new record is actually chained to a list, certain checks for consistency of referencing are made. These are partially accomplished through use of the N-index table, in which all references to labels are recorded. For every label in the program there is a corresponding

entry in this table. In each entry there are two fields: the first specifies the relative address of the labeled statement record ; the second specifies how the label is referenced, i.e., from an 1/0 statement, a DO statement, or a branch type statement. These entries are set up and checked before the statement is linked to the lists.

Examples of., the errors detected at this phase are:

1. Duplication of statement numbers;
2. Referring to a FORMAT statement from a branch statement;
3. Referring to an executable statement from an 1/0 statement;
4. Using an illegal statement as the end of a DO (e.g., a branch type) ;
5. Referring, as the end of a DO, to a statement which precedes the DO.

Another type of consistency error detected at this time is based on ordering of statements. To link a statement into a list, the preceding statement must be available. In the case of an ALTER insertion, the succeeding statement is also available. It is possible, then, to check for violation of such precedence rules as:

1. Declarative statements must precede executable ones;
2. The first executable statement following a branch-type statement must be numbered (i.e., every section of the program should be potentially executable).

It is important to note that all these consistency and precedence errors are reported to the user immediately after the statement is accepted by the system. Most conventional compilers report all composition-type errors throughout the entire program before going on to check for consistency errors. In this system, diagnostics are provided as early as possible.

When the END statement is first linked to the program list, or thereafter at the end of an ALTER sequence, several specialized routines check completeness of control flow and data referencing.

Storage Assignment

The value of a simple variable or a constant is stored in the element record. However, storage for all arrays and any variable appearing in common must be specially assigned. Because of their interaction, all allocation declarations must be entered before storage can be assigned ; on the other hand, storage must be assigned as soon as possible since partial execution of the program may be requested at any time. The Link routine, on recognizing the first executable statement, assigns storage on the basis of all declarative statements, which are linked on the same class list. After an ALTER sequence involving a storage allocation the same operation is .again performed.

Storage Control

When a statement or element record is to be linked to a list, it is moved from a temporary working area to the program area. Space for successive records or data storage is at first assigned sequentially throughout the program area.

When the user deletes (via ALTER) any statement or variable from the program, the associated records are unlinked from the program and chained to a "null" list ordered by size of record. When space is needed for a new record, the null record that best fits (i.e., large enough but with minimal "trim") is selected; this technique prevents wasteful fragmentation of the null-storage areas.

If no record on the null list satisfies the space requirement, but the total size of the scattered null records would provide enough space, then a "squeeze" is performed by moving every record in the program to a contiguous storage area. All references to relative addresses in the program area are then changed to reflect this relocation.

The Interpreter

Execution of the user's program is done in an interpretive fashion[20,21,22,23,24] on a statement-by-statement basis under control of the Process Control program. This control program sends to the Interpreter the address of the statement to be executed. Upon successful

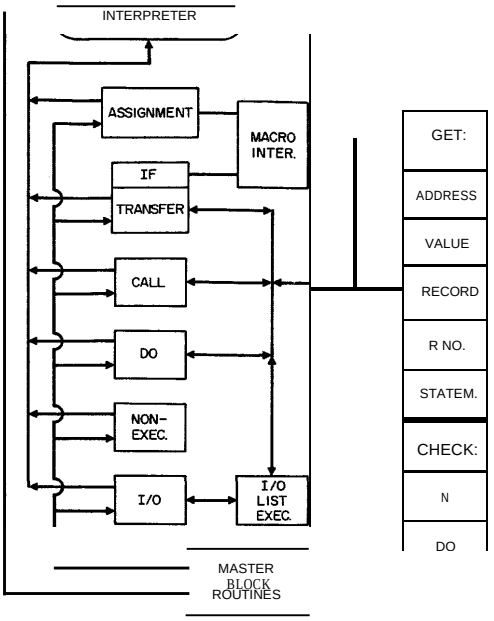

Figure 2.7c. General Diagram of Process Control Flow -Interpreter.

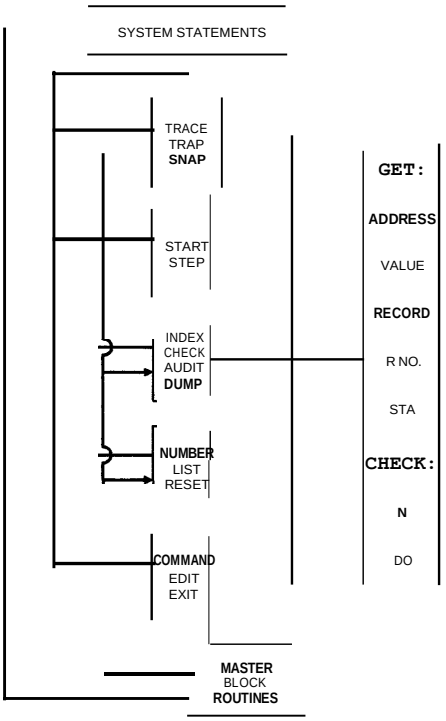

Figure 2.7d. General Diagram of Process Control Flow -System Statements.

execution of this statement, the relative address of the next statement is saved in the program header. At this time, an indicator is turned on in the statement record, showing that the statement has been executed.

The Interpreter can be broken into several parts (see Figure 2.7c) :

1. The master interpreter, which decodes the statement type;
2. The service subroutines used by all statement routines;
3. The macro interpreter used for arithmetic expressions ;
4. The various statement routines.

Decoder

To interpret a statement, a code is fetched from the statement record and, using the Master Block, control is transferred to the appropriate Interpreter routine.

Service Routines

These subroutines are used to fetch element and statement records and to address value words for variables and constants. In the Interpreter, all fetching is done by a direct "look-at" of a table entry for an address. This is in contrast to the associative referencing used in the Translator. Execution speed is considerably increased by this elimination of list searching.

All references to a variable in a statement record are by its internal identifier. This number is used as a key to the R-index table to access the relative address of the element record. Whenever a value word is fetched for use, an indicator for "variable used" is turned on in the element record. Similarly, if a value is stored into a value word, an indicator for "variable set" is turned on.

Macro Interpreter

The evaluation of an arithmetic expression can be expressed in hardware terms; i.e., the system has an instruction repertoire of six two-address instructions, and is equipped with a group of pushdown regist ers. [25,26,27,28,2a] Execution of the statement in the interpretive

mode is analogous to machine execution and involves several steps:

1. Fetch the next niacro (i.e., instruction) to be executed;
2. Fetch operand values;
3. Decode the instruction operation ;
4. Perform the specified operation;
5. Store the result in a push-down stack.

Subscripts of arrays and arguments of f unctions are indicated by a special operator. When this operation is encountered, an entry is made in a push-down parameter stack. Values are fetched from this stack for computing an array address or for passing arguments to a function.

Functions are also indicated by a special operator. When this operator is encountered, the function-element record is fetched. All records for librar'y routines point to their actual machine coding within the system. All other functions are called from random storage.

Statement Routines

Every statement has a particular Interpreter routine associated with it. There are several categories which should be discussed:

1. Arithmetic-the macro interpreter is used to evaluate the expression to the right of the "==" and store its value in the left-hand variable.
2. Branch-the macro interpreter is used to evaluate the arithmetic expression for an IF ; a service routine is used to fetch the value for I in GO TO (. •..), I. The proper transfer point is chosen from the list of numeric labels in the statement. This numeric label is used to access the N-index table for the relative address of the statement to which control should flow.
3. DO loops-the initial execution of a DO statement creates an entry in a push-down stack controlling DO nesting; it also initializes the value of the DO index variable and flags its element record as an active DO index. The execution of the last statement in the range of a DO is detected through checking of an indicator turned on in the translation process. After execution of this last statement, the DO statement is fetched again and its index is tested and incremented. Execution continues with the statement following the DO until the indexing condition is satisfied.
4. Input/Output-an 1/0 macro interpreter is used to compute addresses of values to be passed to the appropriate input/output service routine. A table is generated in the execution process to handle variables in the list controlled by implied DO's.

Execution Diagnostics

Choosing the interpretive approach to execution necessarily means sacrificing speed. For debugging purposes, this is not often a serious impediment-especially since diagnostics are possible for many errors never detected in conventional execution. These include detecting:

1. A value word not being set before used :
2. A subscript value not being valid;
3. A DO index being reset in the range of the DO;
4. A computed GO TO parameter not being in range;
5. The size of an integer exceeding its limits;
6. The existence of an illegal value in an I/0 list with implied DO's.

Input-Output Control System (JOGS)

The prime responsibility of the Input-Output Control system is to select, from the respective queues built up by the Scheduler, the next task or combination of tasks to be performed by the individual 1/0 units. Upon completion of a given task, the Scheduler is notified either directly through program switch indications or indirectly through the terminal header. Before relinquishing control to the Scheduler, the IOCS initiates the next task for that channel device based on the queue information. It also maintains control surveillance over all 1/0 buffer areas to prevent overflow.

The 1/0 attachments consist of disk, drum, magnetic tape, ca.rd reader, on-line printer and communications exchange.

The disk is used as a permanent storage for user programs. The drum serves as a rapid

access storage device for the repeated shuffling of user programs in and out of memory. The magnetic tapes, card reader, and printer are used in a conventional manner.

The communications exchange has some interesting capabilities not found in more conventional 1/0 equipment.

The Exchange

The IBM 7740 communications control system30,31 is used to buffer and control the traffic flow between the communications network and the IBM 7040 computer.

It is a stored-program computer with a rather specialized instruction repertoire designed for real-time applications. The instructions possess powerful logic and data manipulating facilities, through somewhat limited arithmetic capability. Instructions are fixed in size, one instruction per 32-bit word, while data is composed of strings of 8-bit characters. Addressing is at the character level, up to a maximum of 64K characters (i.e., 16K words).

The 7740 program performs several communications-oriented functions. First, it accomplishes line and terminal control by generation, recognition, and manipulation of control characters, in order to establish a connection to the remote terminals, and to determine the operation to be performed. Second, it provides message control, so that· the messages may reach their intended destinations: they are logged in, monitored for correctness, and converted from the various transmission codes to the codes acceptable to the other devices in use. Third, it provides protection to ensure the proper disposition of messages, and to ensure the correction of transmission errors wherever possible.

To simplify these functions, the 7740 has several hardware and programming capabilities not often found in conventional computers.

1. The most striking of these is the ability to operate in an independently controlled hierarchy of modes. In increasing order of priority (that is, decreasing order of interruptability), these are:
   a. The normal mode. The normal activities involved in polling, addressing, and monitoring of all communications devices are conducted in this mode. Because of the large number of lines, processing is on a continuous service basis, whereas a conventional computer attains 1/0 overlap by yielding independent control to the devices and servicing them on an interrupt basis.
   b. The 1/0 mode. This mode is used to control input/output between the 7740 and the 7040. A special uninterruptable state, called copy mode, is used for the actual transmission of information.
   c. The attention mode. This mode is entered when service (not connected with any hardware malfunction) is needed (e.g., servicing the interval timer).
   d. The service mode. This mode is entered if malfunctions are detected.

   Entry to the service, attention, or 1/0 copy modes may be initiated by the machine; entry to any mode may also be initiated by the program. In addition, it is possible to inhibit mode change so that tables or programs used in several modes may be protected (this is analogous to disabling a channel on a conventional computer).

   Associated with each mode is a pair of machine registers which contain the complete status information. Mode change is automatically accomplished by storing this information into the cells associated with the old mode and picking up the corresponding information from the cells associated with the new mode.
2. Time-stamping, essential to control in any communications or real-time environment, is provided for by the interval timer, which is automatically updated by the machine every few milliseconds. This timer is used in conjunction with attention-mode programs to provide a programmed real-time clock, and a programmer-accessible interval timer.
3. Information about each of the communication channels (or lines) is maintained in fixed positions of core storage using two channel-control words, one pair for each line involved. The current status informa-

tion of these words is manipulated by both the hardware and the programs in order to control the flow of information within the system.

4. In order to facilitate the acquisition and transmission of data, the memory of the 7740 is considered by the hardware to be divided into blocks of 32 characters, each of which begins on an 8-word boundary. The first 30 characters of each block are used to store data, while the last two provide a 16-bit chaining address used to indicate where the next block of information is located. These chain addresses, supplied by programming, are used by the hard- ware to advance automatically to the next character location.

   Because storage is not infinite, it is possible to place a special indicator in the chain-address location of any block. When this buff er-block signal is detected by the machine in the process of acquiring a new block, automatic entry into the attention mode occurs, thus enabling the program to accurately control the available storage pool.

CONr.T ,TTnTN REMARKS

The Translator described performs a mapping of a source program to an equivalent, list- - structured, internal form. This method may be called "selective" or "differential" compiling, because statements may be inserted, replaced, and deleted without retranslating the entire source program. In addition, this approach provides rapid, comprehensive reference and diagnostic data. And finally, the process is reversible ; the source program may be regenerated in its original form, or in a related for m.[32]

Interpretive execution provides the means for complete source-language debugging. Information on the dynamic behavior of data use and control flow can be applied to improve optimization of the generated object code.·

The implementation and description of the rem9te-computing system has naturally been done in a time-sharing context. Nevertheless, the techniques used are equally applicable to a conventional compiler operating under a monitor system.

Standard hardware devices in a conventional configuration were adapted to this purpose through programming. However, system performance could be substantially improved by use of a special machine organization designed to perform the same functions.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the contribution of two associates: Miss Harriett Cohen for the stQrage allocation routines, and Mr. Dan Davis for the l/0 list decomposition and interpreter routines.

APPENDIX I-EXPRESSION DECOMPOSITION /RECOMPOSITION

Introduction

The primary purpose of any formula translator is to reduce expressions to a form that provides the fundamental order in which operations should be performed to produce correct results. Implicit in this form should be a record of the order in which partial results are deveioped, accumulated, and reused.

The techniques and traditional programming tools generally applied to accomplish this are [33, 34, 35, 36]

1. Forward scan ;
2. Push-down list;
3. Forcing tables ;
4. Ordered macro list;
5. Implied push-down temporary indications;
6. Chaining and string concatenation.

Forcing tables are used to produce an order of operation based on the real or assumed hierarchy of arithmetical or mathematical operators. Push-down lists in this respect often work on a LIFO (last in-first out) principle. Macros are used as a form which approaches as nearly to a machine-executable form as can be used while retaining its machine-independent structure. In addition to the operator and the operand elements, the macro form often

contains a reference to the temporary result that the operation will produce, such as Tl or T2, implying a push-down order to partial results. String-manipulation techniques of chaining and concatenation are often employed to facilitate translating operations.

Requirements

The Arithmetic Translator includes not only the traditional decomposition but also a recomposition37 phase to restore the statement or expression to its original form from the compressed macro string generated during decomposition. The macro string generated, therefore, must satisfy several requirements:

1. It must be easily interpreted, saving time;
2. It must be compact, saving space;
3. It must be recomposable.

The decomposition translator must detect all errors in logic and syntax. It must supply the number, order, and mode of all operations to be performed by the Interpreter.

The recomposition translator should develop a string in the original sequence; all necessary punctuation must be restored. In short, it must produce a string identical in all respects to the original, except for the removal of redundant parentheses. The resulting string, when decomposed again, should produce a macro string identical to that originally decomposed.

OPERATOR		DECOMPOSITION		RECOMPOSITION	
Symbol	Nome	Left On	Right On	Old On	New On
+	plus	5	5	2	1
	minus	5	5		
	multiply	4	4	3	
I	divide	4	4	3	
	exponentiation	4	3	3	
so	subscript	6	0		
fo	function	6	0	7	
	replacement	7	0	7	
um	*vnory* minus	5			
	comma	6	5	0	
	left poronthesis	6	0	0	0
	right poronthesis	0	6	0	0
0	end of message (EOM)	0	7	0	

Note: The zero code signifies that the operator is illegal when appearing in the specified role.

Figure 2.8. Forcing Tables for Translator.

Techniques

Forcing Tables

The forcing tables in Figure 2.8 are used as follows. The decomposition table is used to cause the• generation of macros based on the relative hierarchy of related or successive operators. If the value for the right operator is equal to or greater than the value for the left operator, then a macro based on the left operator is generated.

The recomposition table is used to decide when parentheses are necessary to maintain the hierarchy implicit in the order of macros previously generated. If the value for the new operRtor is greater than or equal to the value for the old operator, then the string developed around the old operator during a previous concatenation must be enclosed within parentheses before further concatenation can take place.

Push-Down Lists

The lists used in the decomposition translator are the operator and variable lists which hold those elements awaiting further action from a forcing situation. The recomposition translator has an operator list used essentially for the same purpose. In addition, it uses two lists which contain control words of partial strings awaiting further action. The "work list" contains the control words of strings which are to be concatenated into a single string with a single control word. This control word is then placed on the "string list" until a later call for further concatenation is encountered.

M aero Strings

Each macro contains an operator byte and one or two variable bytes. The operator is a basic operation plus an indication of the modes of the variable bytes. Either or both variable bytes may contain a temporary indication. These do not have to be specific temporary indications, since owing to the ordered structure of the macros, both the Interpreter and the recomposition translator use a push-down accumulator for storing and fetching partial re..: sults of execution and partial strings developed through concatenation. For the same reason, no indication need be kept in the macro of the

temporary to be generated by the operator (such as Tl or T2). To save space, macros with temporary indications may be compressed during packing procedures by indicating left and/ or right temporaries in the operator byte, thus eliminating all bytes for temporary indications. For example, when the macro string generated in the example is compressed in this manner, it re3ults in 22 bytes, or, at four eight-bit bytes per word, less than six words of storage. The ln terpreter accesses macros-in order-for execution of the statement, building up temporary re3ults and using them in turn when later macros call for them. An EOM (End of Messa_ge) operator signals the end of the macro list. The recomposition translator accesses the macros and builds up temporary strings in much the same manner as the Interpreter. Also from this simplified, compact macro form it is but a simple step to generate machine-language code; either temporary locations can be implicitly addressed by the machine itself or else explicit storage addresses can be used.

Chaining and Concatenation

In the recomposition translator; when an operand is not an intermediate temporary, it is developed as an element in the output string and placed in an empty word in a pool. It is treated as a one-element string and assigned a

		Mixed 1'kde
	ii.	Mixed fvlode in a Function Argument
	iii.	Illegal Use of Function or Array Name Without Arguments
	iv.	Simple Variable, Constant, or Expression followed by left Parentheses
		Illegal fvlode of Function Argument
	vi.	Illegal Number of Arguments in Function
	vii.	Fixed to Float Exponent
	viii.	Level of Nesting of Functions Exceeds Maximum Number of Eight (B)
	ix.	Illegal Successive Operators
		Illegal Parenthetical Order
	xi.	Uneven Number of Parentheses
*	xii.	General Syntax Error
*	xiii ..	Expression Begins with Illegal Operator
	xiv.	fvlode of Variable Not Set
	xv.	1','ode Not Set For *Arry* Arguments in Function
	xvi.	Number of Parameters in Function Exceeds Declared Maximum
	xvii.	**Number** of Arguments in (Defined) Function Specified lo be Zera
	xviii.	1','ode of Actual Argument is Not Set
	xix.	Illegal Operator in Parameter or Illegal Position for Comma
		A⁰ B**C condition - (illegal in FORTRAN)

.. Only these errors cause an immediate error return.
All others return for further error checki':!S.·

Figure 2.9. Arithmetic Translator Diagnostics.

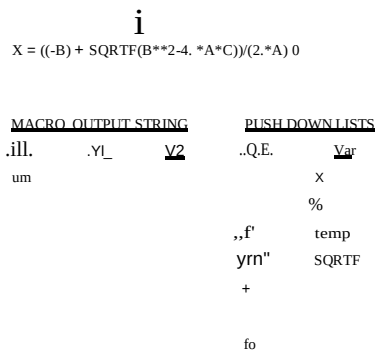

Figure 2.10a. Example of Decomposition-Part I.

control word. When strings are to be joined together, the last word of the first string refers to the linking operator (which is developed in the empty pool), and in turn, the operator refers to the first word of the next or preceding string. The two or more control words are combined into one which references the first and last words of the concatenated chain or string of elements. When the recomposition translator eventually encounters the EOM op- erator, there is only one chain represented by a control word on the string list. This chain or scrambled string is then unraveled into a sequential list of all the elements in the re- composed statement or expression.

Diagnostics

The decomposition performs complete diagnostic checking. Wherever possible, error checking continues even though some errors

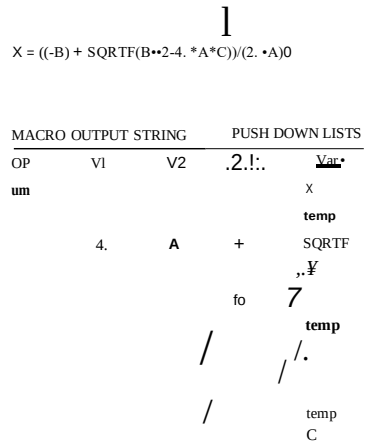

Figure 2.10b. Example of Decomposition-Part II.

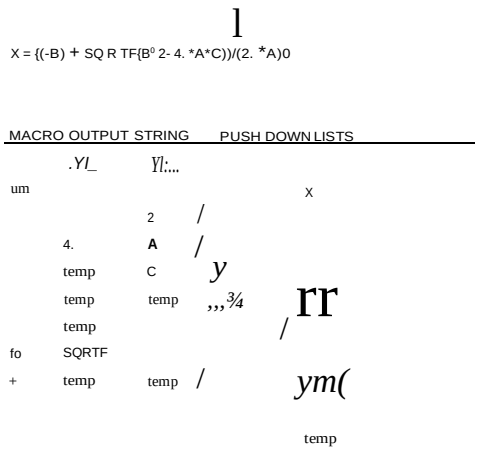

Figure 2.10c. Example of Decomposition-Part III.

have already been encountered. (See Figure 2.9 for a list of decomposition diagnostics.)

Decomposition Rules (Figure 2.10)

1. When all action has been taken with a new operator or variable encountered in the forward scan it is placed on the appropriate push-down list.
2. An array name or function name followed by a left parenthesis generates two additional operators for the operator list: a subscript operator or function operator, and a comma operator for the, initial parameter.
3. When the forcing value of a new operator equals or exceeds the forcing value of the last operator on the operator list, action is taken to output a two- or three-byte macro :
   - **a.** The last operator on the operator list and the last one or two variables on the variable list, depending on the operator, are removed and incorporated into an output macro.
   - **b.** For each macro generated for the output string, except for comma-operator macros, a temporary indication is generated on the variable list.

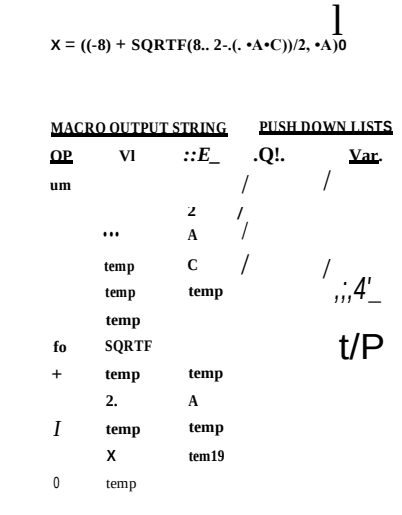

Figure 2.10.d. Example of Decomposition-Part IV.

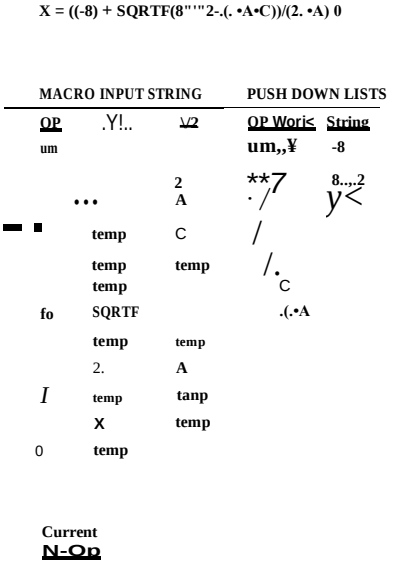

Figure 2.11a. Example of Recomposition-Part I.

Recomposition Rules (Figure 2.11)

1. On each new macro encountered in the forward scan, the right operand (V2), if it exists, is always considered for action before the left operand (VI).
   - a. If Vi of a macro is not a temporary indication, it is developed in a word from the empty pool, and assigned to a control word which is placed on the intermediate work list.
   - b. If V_1 of a macro is a temporary indication, the last control word on the string list is removed and placed on the work list.
2. For any operator in a macro except the comma operator , the last action taken is to

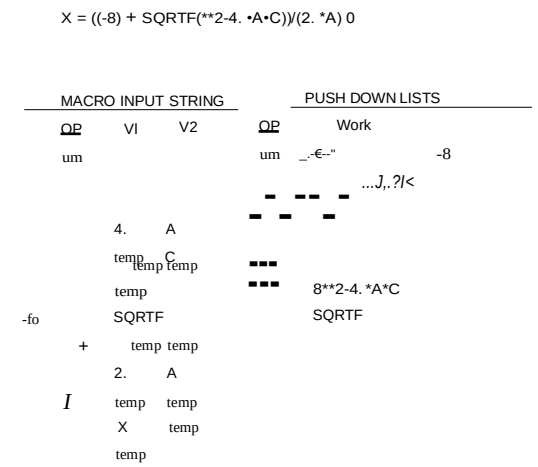

Figure **2.llb.** Example of Recomposition-Part II.

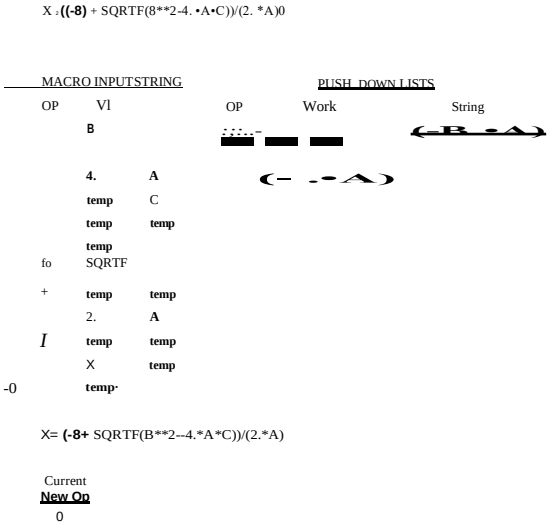

Figure 2.lld. Example of Recomposition-Part IV.

place the operator temporarily on the operator push-down list, and to combine the control words on the work list, linking **tl-u ;-,. C!t-r;nn-C! tnn-at-ha-r in+n nn.n nnn+-.-,-,.1** word, which is placed on the string list.

a. For subscript and function operators, the name and parameter strings referenced on the push-down work list are linked in order, separated by appropriate parentheses and commas.

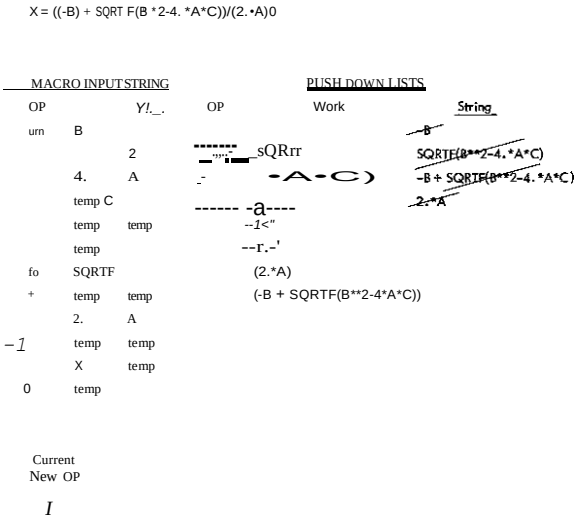

Figure 2.llc. Example of Recomposition-Part III.

b. For arithmetic unary or binary operators, the one or two strings refe''renced on the work list are linked with the operator.

3. Whenever a control word is removed from the string list, an operator is removed from the operator list and tested against the new operator from the current macro.
   a. When the right forcing-value of the new operator equals or exceeds the left forcing-:value of this last operator from the operator list, parentheses are placed at the ends of the string referenced by the control word just placed on the work list.
   b. If the string represents V2, and the left forcing-values of the new and last operators are equivalent, parentheses are placed at the ends of the string.

Extensions

There is no limit to the length of the statement string that can be used as input to this type of decomposition translator.

Any mathematical language based on hierarchical rules of operation-for purposes of computation similar to that in arithmetic formulas-can be decomposed and recomposed just as easily using forcing tables and the other traditional techniques. The macro form pro-

duced could be of quite a different form depending upon the nature of the interpretive scan. It would, of course, have the same implied order of operation. The operators involved need not be only unary or binary operators; they need not be only arithemtical or functional. Boolean operators, logical operators, ternary operators, or any others could be easily handled in this manner.

REFERENCES

1. A. NEWELL (Ed.), "Information Processing Language-V Manual," Prentice-Hall, 1961.
2. H. GERLERNTER, J. HANSEN, and C. GERBERICH, "A FORTRAN-Compiled List-Processing Language," *ACM Journal,* April 1960.
3. A. J. PERLIS and C. THORNTON, "Symbol Manipulation by Threaded Lists," *ACM Communications,* April 1960.
4. A. EVANS, A. PERLIS, and H. VAN ZOEREN, "The Use of Threaded Lists in Constructing a Combined ALGOL and Machine-Like Assembly Processor," *ACM Communications,* January 1961.
5. J. WEIZENBAUM, "Knotted List Structures," *ACM Communications,* March 1962.
6. J. WEIZENBAUM, "Symmetric List Processor," *ACM Communicat ons,* September 1963.
7. R. BROOKER and D. MORRIS, "A General Translation Program for Phrase Structure Languages (Lists)," *ACM Journal,* January 1962.
8. H. W. LAWSON, "The Use of Chain List Matrices for the Analysis of COBOL Data Structures," *Proc. ACM National Conterence,* September 1962.
9. H. D. BAECKER, "Mapped List Structures," *ACM Communications,* August 1963.
10. P. R. KOSINSKI, H. KANNER, and C. L. ROBINSON, "A Tree-Structured Symbol Table for an ALGOL Compiler," *Proc. of ACM National Conference,* August 1963.
11. P. M. SHERMAN, "Table Look-at Techniques," *ACM Communications,* April 1961.
12. L. R. TURNER, A. MANOS, and N. LANDIS, Initial Experience on Multiprogramming on the Lewis Research Center 1103 Computer," *Proc. of ACM National Conterence,* August 1960.
13. N. LANDIS, A. MANOS, and L. R. TURNER, "Initial Experience with an Operating Multiprogramming System," *ACM <:;ommunications,* May 1962.
14. A. B. SHAFRITZ, A. E. MILLER, and K. ROSE, "Multi-level Programming for a Real-Time System," *Proc. EJCC,* December 1961.
15. E. F. CODD, "Multiprogram Scheduling," *ACM Communications,* June 1960, July 1960.
16. E. F. CODD, "Experience. with a Multiprogram Scheduling Algorithm," *Proc. of ACM National Conference,* August 1960.
17. J. WEGSTEIN, "From Formulas to Computer Oriented Language," *ACM Communications,* March 1959.
18. B. ARDEN and R. GRAHAM, "On GAT and the Construction of Translators," *ACM Communications,* July 1959.
19. M. E. CONWAY, "Design of a Separable Transition Diagram Compiler," *ACM Com,-munications,* July 1963.
20. The 701 SP,_eedcoding System, IBM Form Number 24-6059.
21. The 705 Print.System, IBM Form Number 32- 7855, 1957.
22. Bendix Intercom 1000 Programming System, Bendix Computer Division, 1958.
23. W. R. BRITTENHAM, et al., "SALE (Simple Algebraic Language for Engineers)," *ACM Cornmunications,* October 1959.
24. R. E. MACHOL, W. J. ECCLES, and J. C. BAYS, "There's Still a Place for Interpreters," *Proc. ACM National Conference,* September 1962.
25. W. LONGERAN and P. KING, "Design of the Burroughs B5000 System," *Datamation,* April 1961.
26. R. W. BARTON, "A New Approach to the Function Design of a Digital Computer," *Proc. W JGC,* April 1961.
27. J. ANDERSON, "A Computer for Direct Execution of Algorithmic Languages," *Proc. EJCC,* December 1961.

28. A. C. D. HALEY, "The KDF. 9 Computer System," *Proc. FJGC,* December 1962.
29. C. B. CARLSON, "The Mechanization of a Push-Down Stack," *Proc. FJCC,* November 1963.
30. 7740 Communications Control System Principles of Operation, IBM Form Number A22-6753.
31. 7740 Communications Control System Communications Control Package, IBM Form Number C28-8160.
32. J. J. ALLEN, D. P. MOORE, and H.P. ROGOWAY, "SHARE Internal FORTRAN Translator (SIFT)," *Datamation,* March 1963.
33. K. SAMUELSON and F. L. BAUER, "Sequential Formula Translation," *ACM Communications,* February 1960.
34. H. D. BA;ECKER, "Implementing a Stack," *ACM Communications,* October 1962.
35. C. L. HAMBLIN, "Translation to and from Polish Notation," *The Computer Journal,* October 1962.
36. R. J. EVEY, "Application of Pushdown-Store Machines," *Proc. FJGC,* November 1963.
37. K. IVERSON, "A Programming Language" (Chapter 5), John Wiley & Sons, Inc., 1962.