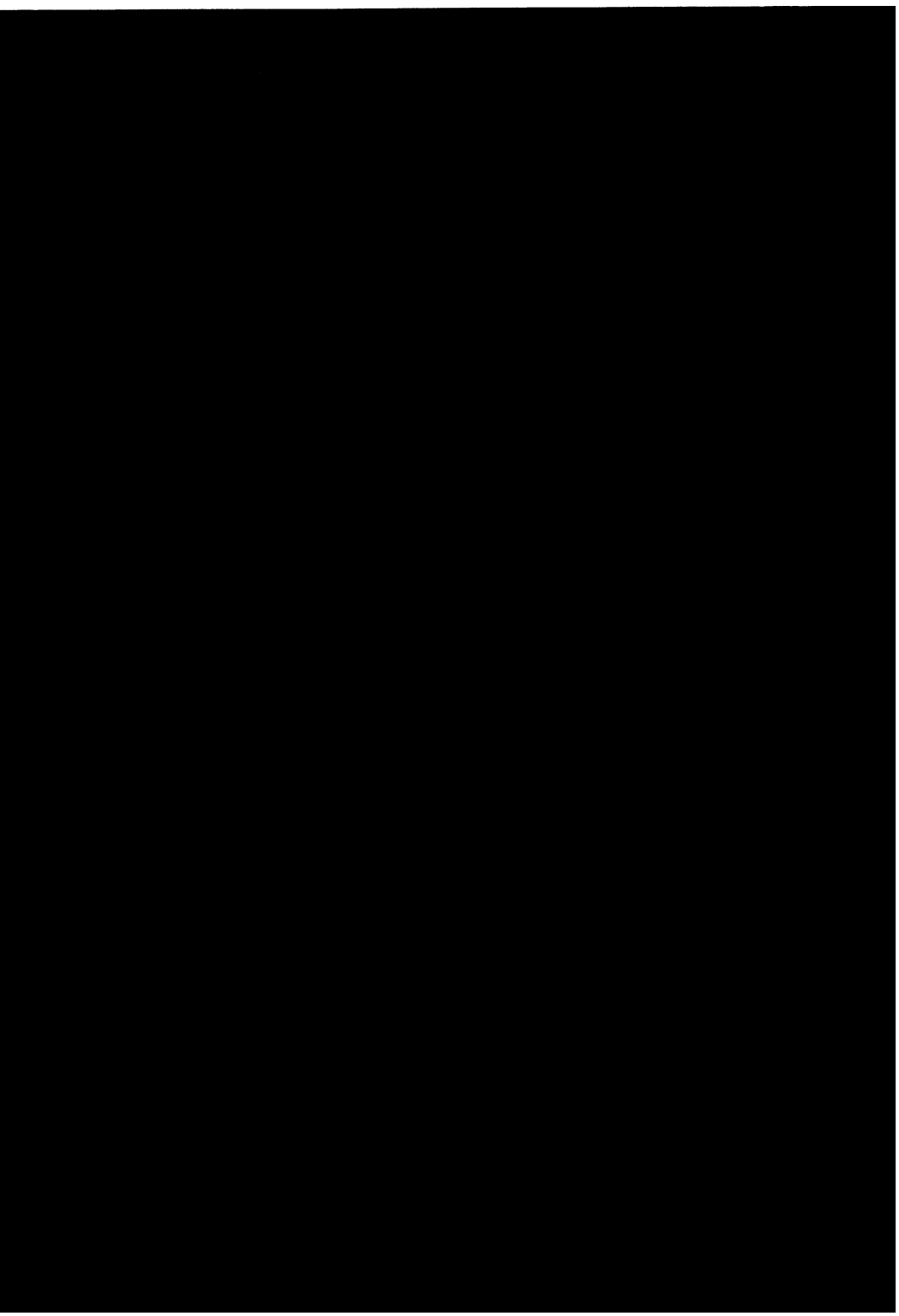


This blank page was inserted to preserve pagination.

CALCULAID: AN ON-LINE SYSTEM FOR ALGEBRAIC COMPUTATION AND ANALYSIS

by

MAYER ELIHU WANTMAN

B.A., Harvard College, 1961

Submitted in Partial Fulfillment
of the Requirements for the Degree
of Master of Science in
Industrial Management

September 1965

Signature of Author . . . . . Mayer E. Wantman . . . . . .
Alfred P. Sloan School of Management, September 15, 1965

Certified by . . . . . .
Thesis Supervisor

Accepted by . . . . . .
Chairman, Departmental Committee on Graduate Students

"Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government."

CALCULAID: AN ON-LINE SYSTEM FOR ALGEBRAIC COMPUTATION AND ANALYSIS

by

MAYER ELIHU WANTMAN

Submitted to the Alfred P. Sloan School of Management on September 15, 1965 in partial fulfillment of the requirements for the degree of Master of Science in Industrial Management.

ABSTRACT

OPS is an on-line system developed by M. Greenberger et. al. at Project MAC. The present work provides a powerful and simple way to perform numerical manipulations and calculations within OPS. The program package is called CALCULAID.

A method of executing algebraic assignment statements, of which MAD and FORTRAN assignments are a subset, is provided. When this assignment-statement ability is coupled with other features of the OPS system, such as unconditional transfers, general conditionals, and array and function declarations, most of the ability of a compiler language is provided. Because the programs written in OPS are executed interpretively, OPS-3 programs can be changed and re-run immediately, without being recompiled.

The other elements of CALCULAID are a program for creating multiple linear regression models, rank-ordering and counting data, and finding roots to polynomial equations in one unknown.

The applications of CALCULAID to the analysis of a round-robin scheduling model and to a process-control problem are discussed, and conclusions regarding the suitability of running computational programs in an interpretive mode are drawn.

Thesis Supervisor: Martin Greenberger
Title: Associate Professor of Industrial Management

September 15, 1965

Professor William C. Greene
Secretary of the Faculty
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Dear Professor Greene:

In accordance with the requirements for graduation, I herewith submit a thesis entitled "CALCULAID: An On-Line System for Algebraic Computation and Analysis."

Thanks are due to Professor M. Greenberger, who was an advisor in name and in deed, and to Professor D. Carroll, who served on my committee and made many helpful suggestions. Conversations with other members of Project MAC, especially those from the Sloan School, provided valuable assistance in detecting logical flaws and clarifying obscurities.

The patience shown by Mrs. E. Schneider, who typed this thesis, was remarkable. I will never know how she managed to smile when the insertion of one word forced the retyping of entire pages.

Sincerely yours,

Mayer Elihu Wantman

Mayer Elihu Wantman

Restricted Distribution

This thesis has been given limited reproduction with the permission of the Alfred P. Sloan School of Management. However, this copy is issued with the understanding that none of the data resulting from this investigation will be used for advertising or publicity purposes, and that the thesis is solely for the confidential use of the organization or individual to whom it is addressed.

TABLE OF CONTENTS

	Page
ABSTRACT	2
LETTER OF TRANSMITTAL	3
CHAPTER I. INTRODUCTION	7
The OPS System	9
CHAPTER II. OPERATIONS WITHIN CALCULAID	11
ASSIGNMENT STATEMENTS	11
Simple Arithmetic Statements	14
Arrays	14
Function Calls	15
Array Operations	16
Literal Vectors	16
Infix Operations	17
Logical Operations	22
COMPUT	23
SETBCD	23
REGRESSION MODELS	24
Data Storage	24
Model Specification	25
ROOTS TO POLYNOMIAL EQUATIONS	28
RANKING AND COUNTING	30
CHAPTER III. APPLICATIONS	32
Analysis of Round-Robin Scheduling in Time-Sharing	32
The Scheduling Algorithm	32
The Analysis	33
Model of a Cheese-Production Process	37
The Process	37

TABLE OF CONTENTS (Continued)

		Page
APPENDIX 1.	INTERNAL CHARACTERISTICS OF SET	42
	BRKUP	42
	MAKLST	43
	Polish Lists	44
	Modifications	45
	EVAL	47
	Standard Polish Lists	47
	Storing of Results	48
	Indexing and Iteration	48
	Special Operations	49
	Suggested Improvements to SET	50
APPENDIX 2.	SUITABILITY OF SET FOR EXTENSIVE COMPUTATIONS	51
	. . . . .	
REFERENCES		53
ASSOCIATED REFERENCES		53

Chapter I

INTRODUCTION

This work was performed within the framework provided by Project MAC, an experiment in multiple-access computer systems. Project MAC provides the means whereby several persons may use the computer at what appears to them to be the same time, but which in reality is a single-server queue in which everyone lines up for service. A user places himself in the queue by issuing a command to run either one of the system programs (such as logging in to the system or compiling a program) or one of his own programs, which may be anything he has previously written.

Each user has space reserved for him in a disk file which he can access directly from his console. It is from this collection of programs in his own directory that he may select a program to be edited, compiled, or run.

The great advantage of a system such as Project MAC's is that each user has a fairly direct line to the computer, but when he is not using the machine someone else may do so. If none of the "on-line" users has any programs to be run, the system supervisor recognizes this fact and runs programs that have been submitted in the traditional way for batch processing. Thus there is no penalty to the user if he wishes to take some time to think about what to do next.

The Project MAC system, known as CTSS (Compatible Time Sharing System) [1], provides a wide spectrum of commands for program compilation and file manipulation. Since the inception of the project in July, 1963 progress has been made in the area of Multiple Access Computers. With the introduction of a new file system in August, 1965 users were given significantly better file-handling commands. The OPS system [2], operating within CTSS, gives the ability to structure commands and change programs as they are being run.

The program "package" called CALCULAID provides a convenient way to specify and execute requests for mathematical computation and manipulations. It runs entirely within the OPS-3 system. In addition to a general assignment-statement capability, such as the execution of the statement

```
X = Y + A * (Z + 5)
```

several more specific programs have been provided. These include a multiple linear or simple polynomial regression program, a solver of polynomial equations in one unknown, a count-by-interval program, and a program to rank-order a given sequence of numbers.

These programs can be used singly, to perform individual tasks, or they can be combined together in "programs", where each step of the program is one of these macro operations. The system can be used either as a very large and fast "desk calculator" or as a powerful programming device which allows simultaneous creation and execution of programs.

A brief introduction to the OPS-3 system will be given, followed by a detailed description of the elements of CALCULAID.

The OPS System

The reader interested in a detailed description of the OPS system is referred to the OPS-3 manual [2]. There he will find extensive explanations of all system programs and functions. The brief introduction to OPS given here serves as background to this thesis.

OPS extends the Project MAC time-sharing system in the areas of command structuring and data handling. The OPS user finds it easy to integrate his own commands into the system and write programs consisting of these commands. Each step of the program consists of one operator. This program of operators (called a compound operator or KOP) can then be used in the same way as an individual operator giving the user great flexibility in tailoring the command structure to his individual needs. Steps that are performed repeatedly can be made into a KOP and called like a subroutine.

A call to a KOP can be a small step, if the KOP has only a few simple operators, or a very large one, if the KOP contains calls to other KOP's which call further KOP's, etc. In OPS, this hierarchical command structure can be built up on-line, with frequent checking of intermediate results.

OPS provides a convenient facility for data storage and retrieval. It uses common storage as a data area that can be dynamically allocated by the user from the console, while the program is running. One can define cells, vectors, matrices, 3-dimensional arrays and functions, in fixed, floating point, or alphabetic mode. Programs have been provided within the OPS system which allow symbolic reference to these data

during execution. The user need not know exactly where the symbols are stored, or if their location changes from one execution to the next, as they are looked up in the symbol table immediately prior to execution.

Data can be saved on and retrieved from the disk by using standard OPS programs. This allows you to progress to some point in a calculation, save intermediate results, and try something new. If the attempt is unsuccessful and destroys the data, the intermediate results can be easily and quickly restored.

Chapter II

OPERATIONS WITHIN CALCULAID

ASSIGNMENT STATEMENTS

One conclusion implicit within the Project MAC time-sharing system is that an individual command should initiate a significant amount of work, without putting an undue strain on the user. There has been considerable emphasis on writing programs that are powerful and easy to use.

The same policy has been followed in the construction of SET, the assignment-statement executor. SET executes the standard arithmetic operations +, -, *, and /. The special feature of SET is that operations can be specified for entire arrays. SET does a considerable amount of interpretation and processing to determine indices and control loops, thereby relieving the programmer of his responsibilities in this area.

Relieved of the tedious job of index- and subscript-tending, the user has more freedom to think about his own problem instead of concerning himself with the "housekeeping" chores. The computer does drudgery with far less fatigue than do humans.

An assignment statement is a line of the form

```
SET A = X + Y + Z * SQRT.(C+D)
```

The input line is analyzed, parsed into the proper groupings, and executed. The symbols on the right-hand side are looked up in the OPS symbol table, the indicated operations are performed, and the result is stored in the location A (also looked up in the symbol table).

first row and second column of A. One very useful property of SET is that if any of the variables mentioned in an assignment statement are arrays, the statement is executed for all possible elements in the arrays. For instance, if B is a vector of 10 elements, the statement

```
SET B = 0
```

sets all 10 cells of B to zero. If A is a vector of 10 elements, the statement

```
SET A = B
```

places the first element of B in the first cell of A, the second element of B in the second cell, etc.

If, however, A is only 5 cells long, the process stops after only 5 transfers have been made. In general, the range of operation is limited by the smallest array mentioned. Single cells and explicit numbers are expanded as far as is necessary, as in

```
SET B = 2
```

which sets all the cells of B to the value 2.

```
SET B = DELTA
```

fills B with whatever value DELTA has when the statement is executed.

The same rules apply to 2- and 3-dimensional arrays. If A is a 5 x 10 matrix and B is a 10 x 5 matrix, only the overlapping 5 x 5 part is affected by the statement

```
SET A = B
```

As soon as the first row of B is exhausted SET begins to transfer the second row. After five rows have been thus transferred no more rows are available in A and the process is terminated.

Rows and columns of matrices may be treated as vectors. A(1) means the first row of A, while A(0,1) means the first column of A. The reason for the 0 subscript is that SET recognizes a subscript of 0 to mean a subscript over which an iteration is to be performed. An omitted subscript is also considered to be an iteration subscript. Thus A(I) and A(I,0) mean exactly the same in a SET statement. The 0^{th} cell of an array is referenced by setting a variable, say X, to zero and then referencing A(X).

The only limitation placed on subscripts is that a subscripted variable may not be used within a subscript. A(B+C/D) is allowable (if B, C, and D are cells), but A(I(1)) is not. The limitation is necessary because of the way in which iterations are handled. Any other kind of computation, including calls to functions, may be included in a subscript.

It is often convenient to have a vector of the positive integers, for use in computing weights or averages. An implicit vector of integers called INTEGR has been included within SET, and it may be used like any other vector in a SET statement. If A is a vector of 10 elements, the statement

```
SET A = INTEGR
```

will put the numbers 1,2,3,4,...,10 into A. Modes (integer or floating point) are again handled properly by SET.

Simple Arithmetic Statements

The four arithmetic operators +, -, *, and /, meaning addition, subtraction, multiplication, and division respectively, and .P., meaning raise to the power are allowed. For example, the statement

SET A = (B+C) * (D-E.P.2)/5

would be executed as follows: If B=1, C=4, D=3, and E=6

$$A = (1+4) * (3-(6)^2)/5$$
$$= 5*(3-36)/5$$
$$= 3-36$$
$$= -33$$

The number -33 would be placed in A. SET performs all necessary mode conversions. If some of the symbols on the right happen to be stored as integers, they are converted to floating-point numbers before any calculations are performed. If A is declared to be integer mode in the symbol table, the result will be converted to integer form before it is stored.

The MAD [3] convention of bracketing all operators (with the exception of +, -, *, and /) with periods has been followed. Thus the power operator, as already mentioned, is ".P.". Bracketing makes operators easily recognizable, and allows simple extension of the operator vocabulary without using special characters.

Arrays

Subscripts are handled just as in FORTRAN and MAD. A(1) indicates the first element of vector A. A(1,2) indicates the element found in the

Function Calls

Calls can be made to all functions except those that return a result through the calling sequence. The reason a function cannot change its arguments is that it is called with pointers to copies of all its arguments, not with pointers to the arguments themselves. Therefore when a function attempts to change something in its calling sequence, it changes the copy, not the original cell with which it was called.

Calls are made as in MAD, with a period between the name of the function and the parentheses enclosing the arguments. Arguments may be computed, and the number of arguments is limited to 10. If you wish to use a function that has not yet been loaded, type

```
LOADO ALPHA
```

where ALPHA is the name of the file containing the desired subroutine. Then type

```
ENTERS FUNCT BETA I
```

where BETA is the entry point to the subroutine (it may be the same as ALPHA) and I is the number of arguments the function expects. Detailed descriptions of these loading and declaration statements are contained in the OPS manual [2]. Now SET has all the information it needs, and a statement of the form

```
SET A = BETA.(C,D+G(3),X)
```

will be properly executed. A call to a function may not contain another function call. For example,

```
SET A = SQRT.(C+SIN.(B))
```

is not allowed.

Array Operations

SET will execute a statement for all elements contained in argument arrays. It can also perform matrix multiplication, take first differences of vectors, or "compress" one vector with another. This gives the user mathematical power beyond that offered by a compiler language. A single statement can do what would take 5 statements in MAD, and the chances for error are reduced, especially since the programmer can be unconcerned with the dimensionality of arrays. If he specifies the multiplication of one matrix by another, SET will check to see that the number of columns in the first agrees with the number of rows in the second. If they do not agree, an appropriate message is printed.

Literal Vectors

If a vector is to be used only once, say for a logical compression, it is bothersome to have to enter it in the symbol table and then do a "TYPE IN" or a series of SET's. Literal vectors allow a vector to be typed directly into a SET statement. To enter a set of values into VEC0, type

```
SET VEC0 = (2,4,A,C+D,VEC1(5))
```

Any computations within this "literal vector" are allowable, except that none of the elements can be of dimension greater than 0. Thus

```
SET VEC0 = (A+B*SIN.(5),B,C,LOG.(2),VEC1.M.VEC2)
```

is legal, but

```
SET VEC0 = (A,B,C,VEC1+VEC2,D)
```

is not.

These console vectors may be used nearly anywhere in a SET statement. The restrictions are noted below.

Infix Operations

Matrix multiplication is best explained by example. Assume

```
A    :  cell
VEC0:  vector of 5 elements
VEC1:  vector of 10 elements
VEC2:  vector of 10 elements
VEC3:  vector of 20 elements
MAT1:  matrix 5 x 10
MAT2:  matrix 10 x 10
MAT3:  matrix 10 x 10
MAT4:  matrix 10 x 20
```

The statement

```
SET A = VEC1.M.VEC2
```

computes the inner product of VEC1 and VEC2 and puts the result in location A.

```
SET A = VEC1.M.VEC3
```

is an error since VEC1 and VEC3 are not of the same dimension. Constants may be used in matrix multiplications.

```
SET A = VEC1.M.1
```

would put the sum of all elements of VEC1 into A. One possible application for this statement is the computation of sum-of-squares. One would use a sequence of statements like

```
SET VEC2 = VEC1*VEC1
SET A = VEC2.M.1/10
```

The first statement squares the elements of VEC1, and the second sums them and divides to obtain the average squared deviation. These two statements could not be written

```
SET A = (VEC1*VEC1).M.1/10
```

because SET does not have the temporary storage that is required. If the RMS were desired we could either add the statement

```
SET A = SQRT.(A)
```

or change the second statement to read

```
SET A = SQRT.(VEC2.M.1/10)
```

Matrices can be pre- or post-multiplied by vectors of the proper dimension.

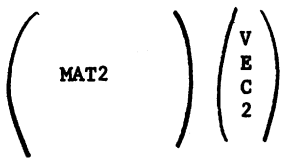
```
SET VEC1 = VEC2.M.MAT2
```

performs the multiplication indicated by

$$\begin{pmatrix} \text{VEC2} \end{pmatrix} \begin{pmatrix} \\ \text{MAT2} \\ \\ \end{pmatrix}$$

```
SET VEC1 = MAT2.M.VEC2
```

performs the multiplication



The statement

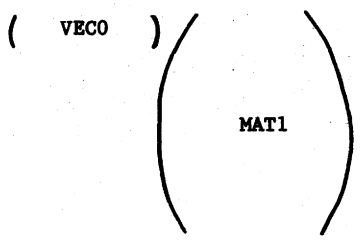
```
SET VEC0 = VEC2.M.MAT1
```

is allowable, since VEC2 and MAT1 are compatible, but the operation will be carried out for only the first 5 elements of VEC2 and the first 5 columns of MAT1. At that time there is no more space in VEC0 and the statement is terminated.

The order of arguments in matrix-algebra is important.

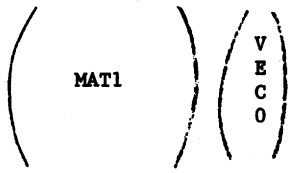
```
VEC0.M.MAT1
```

is allowable, since we are performing



MAT1.M.VEC0

is not allowable, since it implies

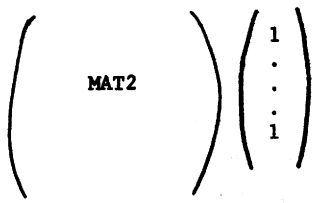


The 10 elements in each row of MAT1 are incompatible with the 5 elements in VEC0.

Matrix multiplication by a constant is handled in a special way. If the left side of the SET statement is a vector, the constant is considered to be a vector also; if the left side is a single cell, the matrix is summed over all its elements and multiplied by the constant. For example

SET VEC1 = MAT2.M.1

performs



That is, VEC1 is set to the row sums of MAT2.

```
SET VEC1 = 1.M.MAT2
```

places the column sums of MAT2 into VEC1. The number 1 is expanded as far as is necessary to satisfy MAT2.

The statement

```
SET A = 2.M.MAT1
```

multiplies each element of MAT1 by 2 and cumulates the total into A. This seems the most legitimate interpretation of an operation which is not defined in standard matrix algebra.

NOTE: There is no possibility of confusing the period associated with an operator with a decimal point attached to a number.

```
1.5.M.VEC1
```

will be interpreted correctly, as will

```
VEC1.M..5
```

Since the matrix multiplication operator automatically transposes a vector if it is required, there is no particular need to transpose vectors explicitly. .T., which will transpose a matrix, has been included within SET.

```
SET MAT2 = .T.MAT3+5
```

will transpose MAT3, add 5 to each element, and put the result in MAT2. If .T. is given a non-matrix operand, SET will deliver an appropriate message.

An implicitly transposed matrix may not be used as an operand to ".M."

```
SET VEC1 = (.T.MAT1).M.VEC0
```

is not allowed. To perform the calculation, define a new symbol, say MAT5, of dimension 10 x 5. Then type

```
SET MAT5 = .T.MAT1
SET VEC1 = MAT5.M.VEC0
```

First-differences can be taken with the .D. operator. It has a single operand, which must be the name of an internally-stored vector (again, not a literal vector). It returns the series of first differences between elements. Suppose VEC1 = .5,3.4,2.9,13.1,12.9,15.7,9.3. The statement

```
SET VEC2 = .D.VEC1
```

would place 2.9,-.5,10.2,-.2,2.8,-6.4 in VEC2. Note that the vector generated by .D. has one less element than its operand.

Logical Operations

SET can perform the logical compression specified by Iverson [4], which uses one array as a mask and another as the operand. The operator is called .C. and is used in expressions of the form

```
VEC1.C.VEC2
```

If an element of VEC1 is positive, the corresponding element of VEC2 is included; if the element of VEC1 is zero or negative, that element in VEC2 is not included. For example, suppose VEC1 were 0,1,1,0,0,0,1,1,1 and VEC2 were 1,2,3,4,5,6,7,8,9,10.

SET VEC0 = VEC1.C.VEC2

sets VEC0 to 2,3,7,8,9. If VEC0 is less than 5 elements long, the statement terminates before all 5 selections are made. As with matrix multiplication, .C. will not accept computed arguments.

Even if the compressing vector has only one non-zero element, SET expects to find a vector on the left-hand side of the = sign. At the time SET is checking the dimensionality of arguments it has no information about the contents of the compressing vector.

COMPUT

SET can be used as a desk calculator. COMPUT followed by any algebraic expression that can appear to the right of an = sign in a SET statement causes the result of the computation to be printed.

COMPUT 2*(Y-Z.P.2)

will print the result of the computation. The number will not be stored in the computer.

SETBCD

BCD information may be entered into single cells by SETBCD.

SETBCD A = CAT

places the word CAT into the cell A. No operator except the = may appear in a SETBCD statement.

REGRESSION MODELS

The most popular kinds of models are linear models since they are most tractable analytically, even though only an approximation to the real world. A typical modeling approach involves theorizing about the behavior of the system and deciding which variables are relevant, gathering experimental data concerning the behavior of the variable, and seeing how closely the data fit the model.

At this point, the researcher can decide the system is no longer interesting; he can theorize a different kind of model; he can gather more data and try again; he can try various subtle manipulations with the model he has just created; or he can decide the model is good enough and go on to further work.

The regression program in CALCULAID is intended to assist in the last steps of the initial experimentation and in the subtle manipulations in the evaluation stage. The program is easy to use, and allows the user to make a change in the model, run it, evaluate it, make another change, etc. He will continue the cycle of modification and evaluation until he is satisfied with the results.

Data Storage

FIT (the regression program) assumes that its data are stored in the following fashion: each variable, dependent or independent, must be stored in its own vector named to have mnemonic value to the user. The first element in each vector should represent an observation taken at the same

time as the first elements of all the other variables. Thus each vector is a time series of a different variable. It is crucial that the vectors all be of the same dimension, and that there be no unused locations at the ends of the vectors. FIT has no way of knowing how many points to include in the analysis except by examing the dimensionality of the vectors.

Model Specification

Suppose you were interested in a system which had 10 interdependent variables, and you had gathered data appropriate to this analysis. For ease in writing, assume the names of the variables (and the names of the vectors in which they are stored) are A,B,C,...,J. Initially you might want to theorize that A is really the independent variable, and all the others are independent. However, B and C are the most important ones, as they seem to influence A most directly. Type in

FIT A TO B C

This will perform the following analysis:

Assume A is related to B and C linearly. That is, A is governed by a relationship $A=c_1B+c_2C+c_3$. FIT analyzes the vectors A,B, and C, and chooses those values for c_i which give the most consistently close approximation to A. The criterion for consistency is that the sum of the squared deviations of A (observed) from A (calculated) is a minimum. The results are typed out in equation form, along with the calculation of the error. After examining the results you may want to include more variables in the model to reduce the error, or try a different combination of variables.

If some of the data points are suspect, you may want either to omit them from the analysis, or reduce their role in the model. This is accomplished by defining a vector, say W, which is of the same dimension as the variable vectors. Each element of W should be set to indicate the relative role each observation should play in determining the coefficients of the model. This weighting vector is called into play by including the word WEIGHT in the list of variables and following it by the name of the weighting vector, as in

```
FIT A TO B C WEIGHT W
```

or

```
FIT A TO B WEIGHT W C
```

The placement of the word WEIGHT is not important, as long as it occurs after the word TO. The two analyses, with the weights and without them, can then be compared.

A vector into which residuals will be placed can be specified in the same way as the weighting vector. The key word is RESIDS, followed by the name of a vector of at least the dimension of the variable vectors.

```
FIT A TO B RESIDS C
```

will store in C how much each observation of the dependent variable differs from the value predicted by the model. It would be easy to change the weight attached to a given point and re-examine the residuals to determine how much the error has changed.

When a FIT is performed the mean square error is printed. Also printed is an "unbiased" value of the error which takes account of the

fact that each additional variable removes one degree of freedom from the system. For example, if we had three data points and three variables, an equation of the form

$$A=c_1B+c_2C+c_3$$

could be found which would fit the three points exactly, no matter what the observed values were. The unbiased estimate then is obtained by multiplying the original estimate by N/(N-V-1), where N is the number of data points and V is the number of variables in the analysis. In the example above, N/(N-V-1) is undefined, since N-V-1=0. If we fit A to B alone, the original estimate of error would be multiplied by 3/(3-1-1)=3.

Simple polynomial regression is a special case of linear regression, where the independent variables are all powers of a single variable. The specification of polynomial regression is

```
FIT A TO B DEGREE N
```

The word DEGREE is recognized by FIT and can appear anywhere on the line after the word TO. N may be specified symbolically or literally, and vectors of weights and residuals may be specified.

```
FIT A TO B WEIGHT W RESIDS R DEGREE 2
```

is quite acceptable. A will be fitted to a second-degree polynomial in B, using weights W. The residuals will be placed in R.

ROOTS TO POLYNOMIAL EQUATIONS

The operator ROOT finds a root of a polynomial equation in one unknown. The method is the Newton-Raphson technique of computing the tangent at a particular point and using that as a local approximation to the curve.

The solution is iterative, and it is possible to have a non-terminating loop. To avoid this, the user is asked to specify some maximum number of iterations. He is also asked to specify the tolerance or precision of the solution. The precision is the difference between successive approximations.

The parameters of ROOT are as follows:

1. The power, P, of the highest-order term in the equation. This need not be an integer. It is assumed that all succeeding powers differ from this one by 1. That is, the equation is of the form

$$a_1x^P + a_2x^{P-1} + a_3x^{P-2} + \ldots$$

2. The name of the array in which the coefficients are stored. The coefficients of the highest powers are first, with the constant term last. For example, if the equation were

$$5x^{3.5} + 3x^{2.5} + 7x^{1.5} - 8 = 0$$

the coefficient array would contain 5,3,7,-8. The coefficient array must be properly dimensioned in the symbol table. In this case, the vector must contain 4 cells.

3. The starting value. This is the value used by ROOT in starting the analysis. It could be selected from a graph

of the curve, or from some other prior feeling about the nature of the roots. If there are several roots to the equation they can be determined by trying different starting values.

4. The tolerance. The analysis is complete when the absolute value of the difference between successive approximations is less than this number.
5. The maximum number of iterations. If the tolerance criterion is not satisfied before the maximum number of iterations has been performed, ROOT returns a "fence" as the value.

Solution of the equation in the example mentioned above is accomplished in the following way. The statement

```
SET COEFF = (5,3,7,-8)
```

will store the coefficients in COEFF, and executing

```
ROOT 3.5 COEFF SVAL TOL MAXIT
```

will find the root of the equation. The numbers stored in SVAL, TOL, and MAXIT will be used as starting value, tolerance, and maximum iterations respectively. The last three variables may be entered directly, as in

```
ROOT 3.5 COEFF 7.5 .00001 10000
```

or they may be omitted entirely, in which case SVAL=0, TOL=.001, and MAXIT=1000 will be used.

```
PRINT ROOT 3.5 COEFF
```

will print the value on the console.

RANKING AND COUNTING

Two operators have been provided to generate information about the relative magnitudes of experimental data. The first, called RANK, rank-orders elements of one vector, A, into a second vector, B. The largest element in A is the first element of B, etc. The mapping (which element of A was the largest, which second largest, etc.) may be specified as the third parameter of RANK. The three parameters are

1. the original vector
2. the resultant vector (* if not desired)
3. the vector of the mapping required to obtain the second vector from the first.

For example, if A contained 8,9,3,7,4,6,2, execution of the line

```
RANK A B C
```

would place 9,8,7,6,4,3,2 into B and 2,1,4,6,5,3,7 into C.

The second operator, called COUNT, places into vector B information as to how many elements of vector A fall into intervals specified by a third vector C. The intervals may be specified at the console directly. For example, suppose A contains 3,100,18,25,16,75,22 and

```
SORT A B C
```

is executed. If C contains 0,25,40,80,200 the first four elements of B are set to 4,1,1,1. 3,18,16 and 22 all fall in the interval 0-25, 25 falls in the interval 25-40, 75 falls in the interval 40-80, and 100 falls in the interval 80-200.

If C contains 0,200,25, B is set to 4,1,0,1,1,0,0,0. If the third element of C is less than the second, C is interpreted as specifying a range and a spacing. In this case, intervals between 0 and 200 with spacing of 25 have been specified.

If any elements of A fall outside of the intervals specified by C, an appropriate message is typed on the console.

Chapter III

APPLICATIONS

Analysis of Round-Robin Scheduling in Time-Sharing

An analysis of round-robin scheduling of requests for the services of a multiple-access computer affords an example of the use of CALCULAID. The analysis [7] was successful and was carried out quickly and easily with CALCULAID.

The Scheduling Algorithm

The scheduling algorithm works as follows: a first-in, first-out queue of users is maintained. When a user desires the services of the computer he is placed at the end of the queue. The computer services users by removing the user at the head of the queue for either a basic time unit, called a quantum or to completion of his request which ever is shorter. When a user's request is finished, he is removed from the queue. If his request is not finished the balance is placed at the end of the queue, and he must wait until every other person in the queue has been run before he gets another turn.

The sizes of requests are assumed to be distributed exponentially, as is the distribution of think times (the time between the completion of a user's request and his initiation of a new one). These assumptions are supported by Scherr [5].

The Analysis

Four KOP's, or compound operators, are used in the calculation. The first, SETUP, asks for information (input parameters) and does preliminary calculations. It calls PROBS, which calculates a vector P where P(I) is the steady-state probability of having I users in the queue for completion of pending requests. PZ is the probability of the queue being empty.

The calculation of the P(I) is done in the loop beginning on line 30 and terminating on line 60. PZ is calculated on line 80, and Z is set equal to the sum of PZ and P_i by the assignment

```
Z=PZ+P.M.1
```

Z is then used to normalize P and PZ so that the probabilities sum to one. The most likely number of persons in the queue can be determined by ranking P (line 110). The first element of MAP is the index in P of the largest element of P. If PZ is larger than P(MAP(1)) then it is most likely that the queue will be empty.

CYCLE calculates YZ, the steady-state wait of the user for his first quantum, and YL, the steady-state wait for all succeeding quanta. The waits are obtained by calculating the number of people in the queue and multiplying by the average request size. COSTS computes the average cost accrued in the system for each request. Note the call to the exponentiation routine on line 10. After the costs have been calculated, control returns to OPS. Another run may be made by calling SETUP again.

```
      KOP SETUP
 10   SET I = 0 , II = 0
 20   TEXT HOW MANY Q
 30   TYPE IN FLOAT NQ
 40   TEXT ENTER THE Q
 50   TYPE IN FLOAT Q 1 Q NQ
 60   TEXT HOW MANY GAMMA
 70   TYPE IN FLOAT NG
 80   TEXT ENTER THE GAMMA
 90   TYPE IN FLOAT GAMMA 1 GAMMA NG
120   SET I = I + 1
130   IF I .G. NQ
140   RETRNK
150   CALLK PROBS
160   CALLK CYCLE
170   TEXT FOR A Q OF
180   TYPE OUT FLOAT Q I
230   SET II = II + 1
240   IF II .G. NG
250   GOTO 300
260   CALLK COSTS
280   GOTO 230
300   TEXT SINGLE LEVEL COSTS ARE
310   TYPE OUT FLOAT CE 1 CE NG
320   SET II = 0
330   GOTO 120
```

```
    KOP PROBS
 20 SET J = 1 , P ( N ) = 1
 21 SET X = EXP . ( - SIGMA * Q ( I ) )
 23 SET X = ( 1 - X ) / SIGMA
 25 SET SIGMAP = ( 1 - X ) / ( S + V )
 30 SET P ( N - J ) = P ( N + 1 - J ) * SIGMAP / ( ALPH * J )
 40 SET J = J + 1
 50 IF J .L. N
 60 GOTO 30
 80 SET PZ = P ( 1 ) * SIGMAP / ( ALPH * N ) , Z = PZ + P .M. 1
 90 SET P = P / Z , PZ = PZ / Z
100 SET RT = N / ( SIGMAP * ( 1 - PZ ) ) - 1 / ALPH
110 RANK P * MAP
120 TEXT MOST LIKELY NUMBER IN QUEUE IS
130 IF P MAP 1 .G. PZ
140 GOTO 170
150 SET LIKELY = 0
160 GOTO 180
170 SET INDEX = MAP(1), LIKELY = P(INDEX)
180 TYPE OUT FIXED LIKELY
190 RETRNK
```

```
    KOP CYCLE
10  SET Z = 1 - P ( 1 ) / ( 1 - PZ )
20  SET YZ = ( S - Q ( I ) * X ) / SIGMA + V * ( S + V / 2 )
30  SET YZ = Z * YZ / ( S + V )
40  SET LB = N / ( 1 - PZ ) - SIGMAP / ALPH - 1
50  SET Y1 = YZ + ( LB - Z ) * ( S + V )
70  SET WP = YZ / ( 1 - ( S + V ) * SIGMAP * ( 1 - PZ ) )
80  RETRNK
```

```
    KOP COSTS
10  SET Z = EXP . ( ( SIGMA + GAMMA ( II ) ) * Q ( I ) )
20  SET CE ( II ) = Y1 + LB * ( S + V ) / ( Z - 1 ) +
                    V / ( 1 - 1 / Z )
30  RETRNK
```

Model of a Cheese-Production Process

Food production is frequently not well understood, because there are often too many variables in a real-world process to allow an analytic solution. An approximation in the form of a model may still be useful, and CALCULAID can help in the formulation of the model.

The Process

Cheese-making is an art, say those who make cheese, because there are so many intangibles. The milk used may vary in age or in butterfat and solids content. There are variations in process temperature and the amount of yeast used may vary. Air temperature may affect the process. The dependent variable is the yield; that is, what percentage of the milk turns into cheese.

The table below contains observations which might have been taken over a period of several months. We will synthesize from the data a model that will help to predict future yields. The model could then be used in a linear program to optimize yields.

The variables are stored in vectors of 20 elements each. Age of the milk is in the vector AGE, butterfat content in FAT, solids content in SOLIDS, process temperature in TEMP, amount of yeast in YEAST, air temperature in AIR, and percentage yield in YIELD. It should be mentioned that these data are purely hypothetical and almost certainly do not reflect reality.

Age of Milk	Percent Butterfat	Percent Solids	Process Temperature	Amount of Yeast	Air Temperature	Yield
10	4.3	2.3	75.0	8.7	72	15.42
6	6.2	3.3	73.4	8.6	73	13.74
4	7.1	2.5	74.0	7.8	85	11.98
8	5.4	3.4	73.7	7.9	84	14.43
3	5.5	3.1	76.3	8.5	90	9.11
7	3.9	2.7	76.2	8.5	74	11.54
14	4.4	2.7	74.3	8.3	77	19.49
11	5.1	2.4	75.8	8.4	70	16.80
6	4.7	2.8	75.6	7.8	65	11.10
7	4.8	3.2	76.1	8.1	68	12.23
2	5.9	3.0	74.7	8.6	74	9.05
13	4.2	2.4	73.9	8.4	75	18.47
9	4.8	2.1	75.2	8.2	80	14.56
8	5.3	2.4	75.8	7.7	83	13.58
12	5.5	2.3	73.3	7.9	81	18.65
13	3.8	2.8	76.9	8.0	88	16.93
5	4.0	2.5	74.0	7.7	77	9.82
9	4.9	3.6	73.8	8.4	77	14.90
10	4.7	3.5	74.6	8.3	70	15.70
7	5.7	2.6	74.7	8.5	72	13.79

In the illustration, user input will be represented as lower case, computer output as upper case, and author comment in parenthesis. It shows both the versatility of OPS and the particular abilities of FIT.

```
fit yield to age

YIELD =
    .8744    AGE +
   6.8947
RMS DEVIATION = .89716
CORRECTED DEVIATION = .92047

OK
```

(To see if age is an important variable, try fitting yield to butterfat content.)

```
fit yield to fat

YIELD =
   -.9720    FAT +
  18.9341
RMS DEVIATION = 2.91177
CORRECTED DEVIATION = 2.8741

OK
```

(The RMS deviation is considerably higher, indicating that age is more important than fat content. To see if fat has much of an effect, fit yield to both age and butterfat.)

```
fit yield to age fat

YIELD =
   1.0305      AGE +
   1.2055      FAT +
   -.4255
RMS DEVIATION = .31955
CORRECTED DEVIATION = .33684

OK
```

(The corrected RMS deviation was reduced from .920 to .337, a factor of nearly 3. These two variables have a considerable effect on yield. See if the solids contents has much effect.)

```
fit yield to age fat solids

YIELD =
   1.0339      AGE +
   1.2054      FAT +
    .1061   SOLIDS +
   -.7475
RMS DEVIATION = .31645
CORRECTED DEVIATION = .34323

OK
```

(The corrected deviation has increased, indicating that yield is not a function of the solids content. Try another factor, say the amount of yeast present.)

```
fit yield to age fat yeast

YIELD =
   1.0312      AGE +
   1.2117      FAT +
    .5541    YEAST +
  -5.0141
RMS DEVIATION = .26652
CORRECTED DEVIATION = .28909

OK
```

(The deviation has decreased somewhat, though not strikingly. Include air temperature instead of yeast.)

```
fit yield to age fat air

YIELD =
   1.0338      AGE +
   1.2364      FAT +
   -.0157      AIR +
    .5979
RMS DEVIATION = .30300
CORRECTED DEVIATION = .32865

OK
```

(Again the deviation has increased, indicating air temperature is not crucial.)

This analysis indicates that the yield is related most strongly to the age of the milk, its butterfat content, and the amount of yeast used. The analysis could be carried further, trying transformations of variables, more variables, or weighting some observations more than others.

APPENDIX 1

Internal Characteristics of SET

The group of subroutines known as SET executes assignment statements in the following steps:

I. Parsing of the input line into meaningful subparts

II. Creation of a modified "Polish list" (changing of the string from infix to suffix notation)

III. Evaluation of the Polish list

The names of the subroutines which perform these functions are BRKUP, MAKLST, and EVAL. The routine which calls the subroutines in the proper order is called SET. Let us consider the routines in order.

BRKUP

BRKUP (short for BREAK UP) takes the input statement and delivers a line which is broken up into meaningful strings. The break characters +, -, *, /, (,), ., =, and , (comma) are recognized. In addition, strings of less than 5 characters which are bracketed by "." are recognized as individual entities. A space is recognized as a break character but is not stored explicitly.

The string

X=A+SQRT.(C+D)

would be broken up into

X = A + SQRT . (C + D)

The string

VAR1=LOG.(B.P.3)

would be broken into

VAR1 = LOG . (B .P. / 2)

There is no possibility of confusing the . associated with an operator with the decimal point associated with a number. The string

.P..3

is broken up into

.P. .3

and

.P.5

is recognized as

.P. 5

MAKLST

This subroutine accepts the broken-up string supplied by BRKUP and creates a modified Polish list. In the process it checks the symbol table for the dimensionality of the symbols appearing in the statement and assures that the statement is dimensionally correct.

Polish Lists

A Polish list is a prefix or suffix specification of a computation, and since SET uses suffix notation, that is what we will consider here. The list is constructed by assigning to each possible operator a precedence, which is an indication of its binding strength. For example, * (multiplication) has a higher precedence than +, and in the expression A+B*C the * will be performed before the +.

The rules by which a Polish list is constructed from an infix-notated expression are as follows:

1. If the next element in the expression is a symbol, put it into the Polish list
2. If the next element in the expression is an operator
   A. If the precedence of the operator is greater than or equal to the precedence of the last operator in the operator list, put the current operator into the operator list
   B. If the precedence of the operator is less than the precedence of the last operator, remove the last operator from the stack and put it into the Polish list. Check the precedence again, and continue to remove operators until either condition A is satisfied or there are no more operators left.
3. When the end of the expression is reached, put all operators into the Polish list.

Consider the expression

A+B*C-D

The first element encountered is A, a symbol, so it is put into the Polish list. +, with precedence 1, is put into the operator list. B is placed into the Polish list, and the status is

Polish list A B
Operator list +

The operator * has precedence 2, which is greater than 1, so it is put into the operator list and C is put into the Polish list, giving

Polish list A B C
Operator list + *

The - is reached, and we go to step 2 B. * is removed from the operator list and appended to the Polish list, and - is added to the end of the operator list. We have

Polish list A B C *
Operator list + -

D is added to the Polish list, and all operators are put into the Polish list (step 3). The final status is

Polish list A B C * D - +

and the operator list is empty. The evaluation of this list will be discussed in the next section.

Modifications

Some of the features of SET and of the OPS system require that more information be kept in the Polish list, information concerning subscripts of arrays, literal vectors, and function calls.

Any symbol that is dimensioned in the Symbol Table receives special treatment. It is followed in the Polish list by the word SUB, which indicates that everything between it and the word SUB. , which must appear later, is part of a subscript. Each subscript is followed by the word FIXED, meaning the subscript is not to be iterated, or the word VARY, indicating it is to be iterated. SUB. is followed by a word which tells how many iteration subscripts the symbol has.

If A is a vector, A(1) appears in the Polish list as

A SUB 1 FIXED SUB. 1

If A is a matrix, A(1) appears as

A SUB 1 FIXED 1 VARY SUB. 1

and the second subscript of A is marked for iteration (omitted or zero subscripts are assumed to be iteration subscripts).

A(B+C,5) appears as

A SUB B C + FIXED 5 FIXED SUB. 1

and this will be correctly evaluated by EVAL.

Literal vectors are handled a little differently. The word CNSL. means that everything between itself and a matching CSL.. constitutes a literal vector. The CSL.. is followed by the number of elements in the vector. The literal vector

(1,3,7,P)

is

CNSL. 1 3 7 P 1 VARY CSL.. 1

The 1 VARY pair is used by EVAL in its iteration indexing, and the 1 following CSL.. indicates the literal vector has 1 iteration subscript (as do all literal vectors). The vector

(B,C+3*F,A(5))

would appear as

CNSL. B C 3 F * + A SUB 5 FIXED SUB. 1 CSL.. 1

Function arguments are bracketed by FUN. at the beginning and a . at the end.

SQRT.(A)

becomes

SQRT FUN. A .

in the Polish list.

The modified Polish list is then delivered to the evaluation program.

EVAL

The program EVAL evaluates the modified Polish list prepared by MAKLST. A description of the evaluation of a standard Polish list will be followed by an explanation of the peculiarities of EVAL.

Standard Polish Lists

The expression

A + B * C - D

is converted into the Polish list

A B C * D - +

EVAL simply examines each element of the list in turn. If the element is a symbol or a number EVAL stores its value in an evaluation list. If the element is an operator, EVAL executes the operator.

In the present example, EVAL would encounter A, B, and C and their values would be stored in the evaluation list. When EVAL finds *, the last two elements in the list are removed and multiplied together, and the product is put back on the list. It now contains A and B*C. D is put on the list, and when - is found D is subtracted from B*C. The evaluation list contains A and B*C-D. + is picked up and executed by EVAL, giving A+B*C-D.

Storing of Results

EVAL must recognize that the execution of = is not like the execution of other operators. The first symbol in the Polish list is looked up but instead of putting its value in the evaluation list, its address in the computer is inserted. Then when the = is executed the result is stored in that address.

Indexing and Iteration

An internal function within EVAL, called INIT, uses the VARY words in the Polish list as indicators of iteration subscripts. It works like a speedometer, iterating on the last subscript until the symbol table says some array dimension is exceeded. It then increments the next higher subscript by 1 and resets the lowest level indices to 1.

Every subscript of a particular level is incremented at the same time, and if any variable size is exceeded, execution of that iteration is terminated. The stopping criterion for the entire statement is the inability to increment any level of subscript and still stay within the size limits of all variables.

Special Operations

Certain operations in SET, namely .M. and .T., must have more information than any of the others, which need only two floating-point numbers for their proper execution. .T. must have available the name and the indices of the matrix it is transposing. This information is determined by examining the Polish list, and the symbol is re-evaluated with the indices reversed. The new value then replaces the old one in the evaluation list.

Execution of .M. is more complicated than that of .T. .M. must make decisions concerning the legitimacy of arguments and the dimension of results, and must handle iterations. This task is complicated by the fact that some of the iterations are handled by INIT and some are handled by special coding within the .M. section of the program. .M. examines the Polish list to determine the names and present indices of its operands and then performs the necessary multiplying and summing. This may involve multiplying one vector by another, a vector by a row or column of a matrix, a constant by a vector, etc.

Suggested Improvements to SET

SET uses no significant temporary storage and as a result all operations having to do with arrays are extremely slow. If A, B, and C are arrays and

```
SET A = B + C
```

is executed, the entire statement must be executed once for every element in B and C. The operations must be analyzed for every iteration.

It would be more efficient to analyze the addition operation once, and perform the computations on the arrays at that time. CTSS routines can provide the temporary storage space required.

The disadvantage of using temporary storage is that execution of the individual operators becomes more complicated; execution of an arithmetic operation would involve more than processing two numbers. The speed tradeoff between faster handling of arrays and slower execution of operations would have to be determined by writing and testing the new program.

Using temporary storage will remove the present restriction that some operations (.D., .C., and .M.) will not accept computed arguments. These operations will no longer go to the Polish list to pick up arrays, but will work, like other operations, on items in the evaluation list. The difference is that in using temporary storage, the items may be entire arrays instead of being restricted to single cells or numbers.

APPENDIX 2

Suitability of SET for Extensive Computation

SET is an interpretive program, and each time a statement is executed it must be completely analyzed. In addition, the analysis concerned with array operations is quite wasteful if arrays are not being referenced. For example, execution of

```
SET A=0
```

involves the execution of several hundred machine instructions. A compiled program would require two instructions.

It is difficult to say exactly what the execution-time ratio between a compiled program and a program of SET's would be, but for a small sample of test programs the ratio was about 1:100. A compiled MAD program that took one or two seconds would take two or three minutes with SET's. In the CTSS time-sharing environment this is not at all desirable, because three minutes of processor time would involve about half an hour of waiting at the console. A wait of that duration would be defeating to the CTSS aim of high-frequency user interaction, as well as being very wasteful of machine time.

SET continues to be quite useful as a fast calculator, performing extended computations on-line, and as a debugging tool. However, after it is determined that a program (KOP) is running correctly, it is advisable to change it to a compiled program. This can be done in two ways.

The most obvious technique is simply to manually translate the KOP into any standard language for which a compiler is available. One could write a program which would look like the KOP version, using the same symbol names, etc. All SET statements which referenced arrays would have to be written out explicitly as loops, with the accompanying high probability of error in indexing. It may even be hard to find out just what the dimensions of the symbols are, as SET looks them up at execution time.

An easier conversion technique is MADKOP [6]. This program transforms a KOP into a MAD program which can then be compiled and loaded just like any other operator. The advantages to MADKOP are several. First, the entire translation, compilation and loading takes but a few seconds of processor time, instead of perhaps an hour of human time. Second, MADKOP can examine the symbol table as it is writing the MAD program, and will program SET's iterations automatically. This relieves the user from having to worry at all about indexing.

MADKOP provides a gain of about 25:1 in execution time over straight interpretation. The full gain of 100:1 is not realized because MADKOP does not write in general the most efficient possible programs. It is fast, however, and relieves the user of most of his indexing responsibilities. A KOP should be compiled early in the debugging stage. If a KOP is to be executed often enough that it will use 10 seconds of processor time, it is almost certain to be advantageous to compile it. The compilation and loading would take about 5 seconds, and total execution time would be less than half a second.

References

1. The Compatible Time-Sharing System, A Programmer's Guide, Second Edition, The MIT Computation Center, P. A. Crisman, Editor, The MIT Press, Cambridge, Massachusetts, 1965.

2. Greenberger, M., et. al. The OPS-3 System for On-Line Computation and Simulation. The MIT Press, Cambridge, Massachusetts, 1965.

3. Arden, Galler, and Graham. Michigan Algorithm Decoder (MAD). University of Michigan Press, Ann Arbor, Michigan, 1963.

4. Iverson, K. A Programming Language. John Wiley and Sons, Inc., New York, 1962.

5. Scherr, A. L. An Analysis of Time-Shared Computer Systems. Project MAC, MAC-TR-18 (Thesis), June 1965.

6. Morris, James. "Interpretive Systems in On-Line Programming". Unpublished Master's Thesis, Alfred P. Sloan School of Industrial Management, MIT, September 1965.

7. Greenberger, M. The Priority Problem. A forthcoming Project MAC Technical Report.

Associated References

Hellerman, H., "Experimental Personalized Array Translator System", Comm. of the ACM 7 (July 1964).

Hamblin, C. L., "Translation to and from Polish Notation", The Computer Journal (October 1962), pp. 210-213.

Samelson, K. and Bauer, F. L., "Sequential Formula Translation", Comm. of the ACM (February 1960), pp. 76-83.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-20

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 54 (60-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☒ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP! (1-54) UNN'ED TITLE PAGE, 2-3, UNN'ED DISTRIBUTION,
UNN'ED TABLE OF CONT, 6-53, UNN'ED BLANK
(55-60) SCANCONTROL, COVER, DOD, TRGTS (3)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 1/8/96 Date Returned: 1/11/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

CALCULAID: An On-line System for Algebraic Computation and Analysis

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Master's Thesis, Sloan School of Management

5. AUTHOR(S) *(Last name, first name, initial)*

Wantman, Mayer Elihu

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
September 1965	54	10

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
Office of Naval Research, Nonr-4102(01) b. PROJECT NO. Nr-048-189 c. d.	MAC-TR-20 (THESIS) 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D. C. 20301

13. ABSTRACT

OPS is an on-line system developed by M. Greenberger et. al. at Project MAC. The present work provides a powerful and simple way to perform numerical manipulations and calculations within OPS. The program package is called CALCULAID, and provides a method of executing algebraic assignment statements, of which MAD and FORTRAN assignments are a subset. When this assignment-statement ability is coupled with other features of the OPS system, most of the ability of a compiler language is provided. Because the programs written in OPS are executed interpretively, OPS-3 programs can be changed and re-run immediately, without being recompiled.

The applications of CALCULAID to the analysis of a round-robin scheduling model and to a process-control problem are discussed, and conclusions are drawn regarding the suitability of running computational programs in an interpretive mode.

14. KEY WORDS

Computer
Machine-aided cognition
Multiple-access computers
On-line computer systems
Real-time computer systems
Time-sharing
Time-shared computer systems

DD FORM 1 JAN 64 1473 (M.I.T.)

UNCLASSIFIED

Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 1/8/1996

M.I.T. Libraries
Document Services