1969

A Mathematical Problem Solving Language and its Interpreter

Lawrence R. Symes

Report Number:
69-044

A MATHEMATICAL PROBLEM SOLVING LANGUAGE AND ITS INTERPRETER

Lawrence R. Symes

May, 1969

CSD TR 44

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

HISTORICAL REVIEW

Characteristics of Mathematical Problem Solving Systems

During the last several years considerable effort has been expended designing and implementing systems which are intended to provide extended capabilities for persons with mathematical problems to solve. These systems can be classified as problem solving systems for applied mathematics.

Before the advent of these systems the research scientist or engineer used a procedural language such as Fortran or Algol when he employed the computer to aid him in solving a problem. Both of these languages, although they resemble mathematical notation much more closely than machine language, are somewhat artificial and contain many unnecessary, from the user's point of view, rules. The artificial appearance and the rules must be mastered before the language can be used. Therefore the scientist or engineer is diverted from his main purpose into becoming a programmer. Even after he has learned the language, its complexity increases the probability of error, and thus reduces his efficiency.

In addition to these difficulties, the user with a mathematical problem had to use program libraries in order to obtain routines for solving commonly occurring problems. These libraries frequently were inadequate and almost always confusing. The routines often were poorly documented and performed little or no monitoring of the accuracy of the results. Thus the user of such a library had to know enough numerical analysis to select the best method for solving his problem and to determine the accuracy of the results.

The problem solving systems which have been designed and implemented attempt to remove some or all these problems and to offer several other desirable features. They have in a sense endeavored to have man do what he is best equipped to do and to have the computer do what it is best equipped to do.

The designers and implementers of these mathematical problem solving systems have utilized six general techniques to assist the user in stating and solving his problem.

First, the source language used to present a problem to the computer is similar to normal "text book" mathematical notation. This permits one to use the system without first having to intensively study the input language. It also reduces the probability of user programming errors, because the user is familiar with

the notation and it is simpler.

Second, clerical statements used for dimensioning arrays and declaring variables are removed from the source language. These are tasks which the computer can easily perform but which are a constant source of errors if the user does them.

Third, the special purpose languages permit the direct manipulation of quantities other than scalars. These may include numeric arrays, functions, and arrays of functions. This further allows the source language to resemble more closely "text book" form. This again leads to fewer statements and hence fewer opportunities for errors in a program.

Fourth, solve statements are included in the source language. These statements permit the user to state a problem he wishes to solve in a concise, natural form. The user may include parameters such as initial values, the accuracy desired, the method he would like used, and he may omit any or all of the additional parameters. The solve statements invoke routines from a built-in library. They attempt to solve the user's problem automatically. They request additional information as needed and monitor the accuracy of the results in order to insure that it remains within the specified limits. The inclusion of these solve statements greatly reduces the burden normally imposed on the user. To solve commonly occurring

problems with the aid of the solve statements, the user is only required to know how to define the equations for the problem; he is not required to know the numerical analysis involved or even the method used. The method is selected by the system and the accuracy of the results is assured.

Fifth, on-line communication between the system and the user is provided. Teletypes, graphical display devices and specially designed consoles are used. The use of these devices bring the computer and the user closer together and consequently improve the user's efficiency.

Sixth, incremental execution of a program is allowed. This, combined with the use of on-line terminals, creates a closed loop between the user and the system. The user is able to monitor his program during execution and the system is able to request information from the user and point out errors when they arise. This eliminates much of the time that is wasted in preparing and submitting runs of a program which are unproductive because the user tried several fruitless cases, has an incorrect program, or has forgotten to initialize a variable.

Five Systems

Five of the mathematical problem solving systems which have been designed and implemented are: the Culler-Fried System (see [1], [20]), the Klerer-May

System (see [7], [8], [9], [10], [20]), the MAP System (see [6], [20]), the Lincoln Reckoner System (see [5], [20], [25], [29]), and the AMTRAN System (see [20], [15], [30]). All of these systems have attempted to make the computer more accessible to the engineer and research scientist. But each system is unique in the sense that they do not equally employ the six general techniques mentioned above. Each of the systems, however, is general enough so as not to be restricted to accepting problems in a single field. We examine some of the unique features of each of these five systems.

Culler-Fried System

Work began on the Culler-Fried System in 1961 at Thompson Ramo Wooldridge and has been continued at the University of California, Santa Barbara.

The source language for the Culler-Fried System is a form of prefix polish notation, with the operator preceding the operands. Thus the source language does not resemble normal mathematical notation and consequently, for the novice, the system is difficult to use.

Some clerical statements exist in the Culler-Fried System. Before a vector is referenced, for example, storage must be allocated for it by the use of a special operator.

A level number is associated with each operator. The level number specifies the type and structure of the

operands. Thus the level number is a type of implicit declaration.

Functions may be manipulated in the Culler-Fried System by loading an X Register with the vector for the independent variable and a Y Register with the vector for the dependent variable.

The user has several buttons on the console to which he can assign routines he has constructed. This provides a means of adding operators to the system at execution time.

Klerer-May System

The Klerer-May System has been developed at Columbia University's Hudson Laboratory and has been available since 1963.

The most striking feature of the Klerer-May System is its input format. It is the only system of the five to employ two dimensional input. Initially the System used a Flexowriter, but work has been done to incorporate a graphic display device into the system. By using two dimensional input the source text appears in its standard mathematical form. This results in self-documenting programs, but requires the user to learn how to manipulate a more complicated input device.

Variables are one character in length. This permits implied multiplication without having to include rules

such as a blank must appear between the variable names. If a user wishes to employ variable names of more than one character he must declare them to be special variables.

All variables have an initial value of zero. Therefore the user never can have an undefined variable in his program. This feature is of debatable value in an on-line system. For with an on-line system the user has the ability to assign values to variables he has forgotten to initialize at execution time, but if all the variables are initialized to zero this error cannot be detected.

The Klerer-May System is designed to be self-teaching. One of the ways this is accomplished is to give the user the system's interpretation of his program in a Fortran-like intermediate language.

When a program is compiled into internal text, semi-automatic dimensioning of arrays is performed. The bounds of arrays are determined from the maximum values assumed by their indices. When the values of index bounds are dynamic the user must include a declaration statement specifying what the largest values of the index bounds will be. Therefore the size of arrays is established at compile time and is not dynamic during execution.

The Klerer-May System has received a limited amount of effort on solve statements and the associated routines for automatic numerical analysis. Instead, high level

operators such as integration have been included in the system.

The MAP System

MAP (Mathematical Analysis without Programming) operates within the Massachusetts Institute of Technology Compatible Time Sharing System. It has been used for research and teaching at M.I.T. since mid 1964.

The most distinctive feature of MAP is the dialogue between the user and the system. After each statement the system responds with the message "COMMAND PLEASE" to signify that it is ready to continue. This feature is reassuring to the novice, but can be annoying to the experienced user..

To insure that the user interacts frequently with the system no logical operators have been included in the language. This forces the user to interact each time he wishes to compare the values of two variables.

The linear source language uses the normal infix notation. Equations and statements resemble normal mathematical notation except for the fact that equations must be enclosed in parenthesis.

The system is best suited for the manipulation of functions of one variable. Functions are defined by symbolic formulas, but the values for the functions are stored in tabular form. When a function is defined the user is asked to specify the domain of the function and

the interval size for the tabular values. The system keeps track of the function and the independent variable automatically.

When a function is defined in terms of another function, the system determines the domain of the function being defined from the domain of the function appearing in its definition.

Functions which are defined in the system (eg. sqrtf) use the convention established in Fortran II of having the name of the function ending in an f.

The Reckoner System

The Lincoln Reckoner System has been developed at the Lincoln Laboratory and has been used by the laboratory staff since early 1966.

The Reckoner System uses a specially constructed terminal consisting of the Lincoln Writer and a CRT display for graphic output. Hard copy graphical output is available from an on-line Xerox printer.

The system consists of a library of routines and the language consists simply of a set of subroutine calls. Each statement is the name of a routine followed by the names of the operands. Routines are available to permit the user to define his own routines.

The main emphasis of the routines is in the area of mathematical computations on arrays of data.

The various routines are controlled by a supervisor called the Mediator. The routines are designed in such a fashion that the results of one routine are available to any other routine in the system. As soon as a statement is entered into the system, control is turned over to the appropriate routine. The user is able to start typing in his next statement while the previous statement is being executed. This overlap results in a decrease in response time and only causes confusion if an error is detected in the previous statement.

Functions cannot be manipulated directly in the Reckoner System. However, they may be manipulated as two vectors, with the user responsible for the dependency between the dependent and independent variables.

The AMTRAN System

AMTRAN (Automatic Mathematical TRANSlation) has been developed at the NASA Marshall Space Flight Center, and has been available there since early 1966.

AMTRAN has employed several types of terminals: teletypes, typewriters, and some specially constructed terminals. The current terminal consists of a keyboard, a typewriter, and two CRT display devices. Statements are entered by means of the keyboard which consists of a standard typewriter keyboard, supplemented by a number of user assignable keys and a set of special function operator keys with labels such as SIN, d/dx, etc.

One scope is used to display the user's program as he constructs it. When he is satisfied that the line he has constructed is correct, he releases it to the system. At this time a copy of the statement is put on the typewriter. Computed results may be put on either the scope or the typewriter. The second scope is used for graphical output.

This type of terminal has the advantage of providing the speed of CRT display devices to communicate with the user, while also providing a hard copy of the program listing.

The user assignable keys on the console are available for creating new operators. The user can construct a routine in AMTRAN and then assign it to such a key. In this way the number of operators can be expanded. This feature was derived from a similar feature in the Culler-Fried System.

The AMTRAN source language employs infix notation for operators. The several function operator keys provide a wide variety of high level operators. Implied multiplication is permitted if a variable name is entered through a special function key.

The notation used for an element of an array is not natural, A SUB 1, refers to the first element in the array A. However, arrays may be manipulated directly as a unit.

Functions may be defined by a symbolic formula, but the independent variable must be a vector. The values of the function are then computed for the values of the independent variable, and the function name then denotes a vector. The user is responsible for keeping track of the dependency between the dependent and the independent variables. To obtain a value of a function the function name is referenced with a subscript.

AMTRAN has included several solve type statements. Some of the solve statements in the system deal with solving systems of simultaneous or differential equations, determining the zeros of functions and performing interpolation.

Other Systems

Several additional systems have been developed in the area of mathematical problem solving. Some of them are: EASL (see [21], [22]), POSE (see [21], [22]), APL (see [33]), VENUS (see [12]), REDUCE (see [4]), JOSS (see [24]), ICES (see [14], [15]), and MATHLAB (see [2]).

Table 1. Comparison of Features of Six Systems

FEATURE	SYSTEM CULLER FRIED	KLERER MAY	MAP	RECKONER	AMTRAN	NAPSS
TWO-DIMENSIONAL INPUT	NO	YES	NO	NO	NO	NO
SELF-EXPLANITORY SOURCE LANGUAGE	NO	YES	YES	NO	YES	YES
FUNCTION MANIPULATION	YES	NO	YES	POOR	YES	YES
AUTOMATIC ARRAY ARITHMETIC	YES	NO	NO	YES	YES	YES
IMPLIED MULTIPLICATION	NO	YES	NO	NO	YES	YES
COMPLEX ARITHMETIC	YES	NO	NO	NO	YES	YES
BUILT-IN HIGH LEVEL OPERATORS	NO	YES	NO	NO	YES	YES
ITERATION VARIABLES	NO	NO	NO	NO	NO	YES
EXTENDED ARRAY EXPRESSIONS	NO	NO	NO	NO	NO	YES
SYMBOLIC ASSIGNMENT STAT.	NO	NO	NO	NO	NO	YES
DETECTION OF UNDEFINED VARS.	YES	NO	YES	YES	YES	YES
SOLVE STATEMENTS	NO	NO	NO	YES	YES	YES
LOOP CONTROL	YES	YES	NO	POOR	YES	YES
EXTERNAL PROCEDURES	NO	NO	NO	NO	NO	YES
INTERNAL PROCEDURES	YES	YES	YES	YES	YES	YES

Table 1 (cont'd.)

FEATURE	SYSTEM					
	CULLER FRIED	KLERER MAY	MAP	RECKONER	AMTRAN	NAPSS
LOCAL VARIABLES	NO	NO	NO	YES	NO	YES
GLOBAL VARIABLES	YES	YES	YES	YES	YES	YES
ACCESS TO PROCEDURAL LANG.	NO	NO	MAD	OWN	NO	NO

THE NAPSS LANGUAGE

General

The NAPSS language offers the user a language in which he can manipulate directly the basic mathematical entities. These include real and complex numbers, functions (which may be symbolic or tabular), vectors and matrices (whose elements may be numbers or functions), and equations composed of any of the preceding objects. The language is designed for a standard conversational terminal, teletype or graphic console, and has 63 characters. It attempts to resemble "text book" mathematical notation as closely as possible within the constraint of a linear notation. It eliminates many of the artificial rules that are included in other languages, without imposing on the language's flexibility or power.

NAPSS is intended primarily as a problem statement language for use in a conversational, incrementally executing mode. However, it allows the user to construct internal and external procedures and thus NAPSS also has the power of a procedural language.

NAPSS incorporates automatic procedures to solve basic mathematical problems such as systems of linear equations, boundary-value problems, and zeros of functions. The user need only supply the system with a description of the problem and ask for its solution. The system then automatically solves the problem by means of polyalgorithms. During the solution, it monitors the accuracy of the results.

A detailed description of the syntax and semantics of the NAPSS language is given in [28]. Here we examine some of the unique features of the language.

Arithmetic Expression

The arithmetic expression in NAPSS allows the direct manipulation of scalars, vectors, arrays and functions. The operators (+, -, /, *, ↑) have their normal mathematical meanings and operate on the operands without any regard to type or mode. For example, a real array may be multiplied by complex scalar and the result is a complex array. Combinations which are not defined mathematically are not permitted. It is not permissible to multiply an n by m array A times a 1 by m vector V because they do not conform. But the arithmetic expression, A * V', is valid where ' denotes transposition.

Implied multiplication may be used in arithmetic expressions in NAPSS where no ambiguity arises. Ambiguities stem from the fact that variable names may be

more than one character in length. Blanks are significant in NAPSS to allow for implied multiplication. Example

2A, A2 ÷ C, and A B + C

mean

2 * A, A2 + C, and (A * B) + C respectively

There are a number of operators in addition to the five basic operators mentioned above. Some of them are: // integer division, || absolute value, ' derivative of univariate functions, ' transposition of arrays and vectors, ∫ integration, and DER partial differentiation. Examples of these operators are:

i) DER(X↑3 + A Y + G(X)) / (X↑2, Y)|X←2, Y←4)

in NAPSS denotes

$$\frac{\delta^3(X^3 + A*Y + G(X))}{\delta^2 X \delta Y}\Bigg|_{\substack{X=2\\Y=4}}$$

ii) |F''(3.5) ∫(X↑2 + G(X)) / X, (X←1 TO 3)|

in NAPSS denotes

$$\left| \left(\frac{d^2F(X)}{d^2X}\Big|_{X=3.5}\right) \left(\int_1^3 \frac{X^2 + G(X)}{X}\, dX\right) \right|$$

There are several methods for constructing vectors and arrays in arithmetic expressions:

i) (1,-3,2,6,-10)

ii) (1,2,...,20)

iii) (1 FOR 20 TIMES)

iv) (2+I↕3 FOR I ← 1 TO N BY 3)

v) ([0:5], 1 TO 6)

vi) ([1,1:11], 3.5 TO 4.5 BY .1)

vii) (3.5 TO 4.5 BY .1)'

viii) ([-1:3,4], (1 FOR 4 TIMES), (-2,-1.75,...,
-1.25), (3 TO 6), (-10,-20,-30,-40))

The first five examples are vectors, which are considered to be column vectors in NAPSS. The lower bounds of the index of the first four vectors is 1 by default. The index of the fifth vector has a lower bound of 0 and an upper bound of 5. Vectors six and seven are both row vectors and they are identical. The eighth example is a square array with the first index ranging from -1 to 3 and the second from 1 to 4. The array is

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ -2 & -1.75 & -1.50 & -1.25 \\ 3 & 4 & 5 & 6 \\ -10 & -20 & -30 & -40 \end{bmatrix}$$

A single element, a row, a column or any arbitrary contiguous subarray may be extracted from a numerical array. Thus, if A is a two-dimensional array with the first subscript ranging from -3 to 3 and the second from 0 to 3, then A[0,*] denotes the 1st row of A, and A[-1:2,1] denotes the column vector consisting of the 3rd through 6th elements of the 2nd column of A.

Arithmetic expressions which yield array results may be subscripted in the same fashion as variables. For

example, (A*B+E)[I,J] and (A↑2)[I1:I2,*] are both valid expressions.

NAPSS also permits arrays of functions to be manipulated an element at a time: 2f'(3.5)[1,3] is the NAPSS equivalent of $2f'_{1,3}(3.5)$.

Assignment Statements

The elimination of mandatory variable declarations in NAPSS means that the association of attributes is performed when a variable is assigned a value. The attributes associated with a variable come from the expression assigned to it.

Left Arrow Assignment Statement

There are two types of assignment statements in NAPSS. In the first, the variable on the left is separated from the expression on the right by a left arrow (←). Such variables are called left arrow variables or simply variables. The left arrow indicates that the arithmetic expression on the right is evaluated and its value is assigned to the variable on the left, similar to what FORTRAN's = signifies. The attributes associated with the variable on the left are obtained from the attributes of the value of the expression on the right. This feature means that a variable may denote a real scalar at one point in a program and a complex array in another.

Example

```
A ← 2, I ← (-1)*.5
B ← ([3,3], 1,2,...,9)
A ← A*I*B
```

After line 3 has been executed, A is a square complex array.

There are two statements on line 1. The first is ended with a comma. A statement, except for a few special cases noted later, need not be terminated with any special character if it is not followed by anything on the same line. If something does follow the statement on the same line, then a comma is needed to separate statements.

The two statements A[1,*] ← (1 FOR N TIMES), A[10,*] ← 10A[1,*] create a square array with 10 rows and 10 columns. The elements in rows 2 through 9 are set to zero because no values have yet been assigned to them. When a variable is assigned a matrix or a vector and no bound information is given for the variable, as is the case here for the column bounds, the bounds are contextually defined as are the attributes from the expression on the left of the arrow. If no bounds are given, the lower bound is set to 1 for created vectors. So the first and second indices of A vary between 1 and 10

Equals Assignment Statement

In the second type of assignment statement, the equals assignment, the variable on the left is separated from the expression on the right by an equals (=). Such variables are referred to as equals variables. In this type of assignment the equals variable is set symbolically equivalent to the right hand expression, instead of being assigned its current value. Thus the variable names in the expression on the right do not have their values substituted for them when the assignment statement is executed. Values are only substituted for the variable names in the expression when a numeric value of the variable on the left is needed; eg. when it appears in an expression to the right of an ← or in an output statement.

Example

X ← 4, Y = 2X, Z ← 2X

X ← 5, W ← Z, V ← Y

The result is V=10 and W=8.

When an equal sign is used, the arithmetic expression on the right must not contain the variable appearing to the left of the equals sign, nor may any of the variables appearing in the arithmetic expression be symbolically equivalent to an arithmetic expression containing the variable name on the left of the equals sign. Examples of illegal statements are:

i) N = N + 1

ii) B = X + A, A = B + C

Both would result in an error message.

A series of assignment statements may be written by using more than one variable name, with accompanying arrow or equals sign appearing to the left of an arithmetic expression. For example X ← A = B ← 2X↑3 + 4 is equivalent to X ← 2X↑3 + 4, A = 2X↑3 + 4, B ←2X↑3 + 4 .

Function Assignment Statements

Functions may be assigned expressions by using either a left arrow or an equals sign. Functions defined to the left of an = sign are called equals functions and functions defined to the left of an ← are called left arrow functions. When the arrow is used, all non-parameter variables in the expression on the right have the current values substituted for them, while when the equals sign is used they do not. Thus the use of the ← and the = when defining a function has the same meaning as when defining a simple variable. Examples are:

```
i)  A ← 3
    F(X)[A+1] ← X↑A + COS(X), ( -2 < X < 10 )
    H(V,W) ← V↑2W, ( -10 < V < 4 AND W > 0 )
    K ← 6, A ← 7
    G(Y) ← F(Y)[4] + 5H(Y,K)
```

Line 5 defines G(Y) to be the function $Y^3 + COS(Y) + 30Y^2$ on the interval (-2,4). Since no explicit domain is defined for G, it is the intersection of the domains of F_4 and H.

```
ii)  A ← 2, B ← 8.5
     F(X,Y) = A*X↑2Y, (X↑2+Y↑2<4 OR 4<=X<5 AND →
              0<Y<1) = X↑2+Y↑3+B, (Y>2+X↑2) →
              = X↑3+(A+B)Y↑3
```

(The → denotes continuation.)

This statement is equivalent to the usual mathematical definition:

$$F(X,Y) = \begin{cases} A * X^2 Y, & X^2 + Y^2 < 4 \\ A_2{}^* X^2{}_3 Y, & 4 \leq X < 5_2 \text{AND } 0 < Y < 1 \\ X_2{}^3 + Y^3 \div B, & Y > 2 + X^2 \\ X^3 + (A + B)Y^3, & \text{ELSEWHERE} \end{cases}$$

Tabular functions are defined in NAPSS by means of the table statement. The various forms of the table statement allow definitions to be made at equally or unequally spaced points in one or more variables. Examples of this statement are:

```
i)  Z ← (7,9,12)
    TABLE( F(X,Y), (1,2,3), (4,6,7), Z )
```

defines F at 3 points: F(4,7) = 1, F(6,9) = 2, F(7,12) = 3

```
ii)  X ← (1,2), A ← 3
     TABLE( F(X,Z), X↑2 + Z + A, X BY Z ← (3,4,5) )
```

defines F at 6 points: F(1,3) = 7, F(1,4) = 8, F(1,5) = 9
F(2,3) = 10, F(2,4) = 11, F(2,5) = 12

When evaluation of a table function is requested at a non-tabulated point, interpolation is used (if possible) to obtain the value.

Equations

A NAPSS equation consists of two arithmetic expressions separated by an equals sign (=). An equation label consists of a variable name, a period, followed optionally

by an integer. A colon is used to separate the label from the equation. The equation label may be used in place of the equation. The assignment of an equation to a label is similar to an equals assignment statement. The association is done at execution time and an equation may be assigned different equations at various times. The equation label denotes the last equation assigned to it.

Example

```
EQ1.1:    2SIN(X) = A X - 2X↑2
EQ1.2:    A X↑2 + B X + C = 0

EQ1.1:    2COS(X) = A X - 2X↑2
```

SOLVE Statement

The solve statement is the most powerful statement in the NAPSS language. It gives the user a means of concisely stating the problem he wishes to solve. The user normally need not concern himself with how the problem is solved. The system selects the method or methods to be used and monitors the errors for him.

The solve statement has the form:

SOLVE EQUATIONS, FOR VARIABLES, OPTIONS;

where EQUATIONS represent the equations to be solved, VARIABLES indicate the variables to be determined, and OPTIONS represents a list of optional information.

The solve statement is one of the statements which always must be terminated with a semi-colon. This allows

the solve statement to extend over several lines without being explicitly continued. A simple example is:

SOLVE X↑2 - 4 = 0, FOR X;

will set X[1]←2 and X[2]←-2.

While the details of the problem solution may be left completely to the system, the user may exercise considerable control by providing additional information, OPTIONS, of the following types:

WITH indicates values to be assigned to variables in the equations. If absent, the current values of the variables are used.

ON indicates the range over which solutions are desired. If absent, any solution is accepted.

NUMBER indicates the maximum number of solutions desired. If absent, the system looks for all possible solutions in the desired range.

USING indicates a particular method to be used. If absent, the system selects a method or methods for solving the given problem. The polyalgorithms use intermediate results to decide which methods to use in the current situation.

TYPE indicates the type of problem or equation to be solved (eg. linear system, polynomial, boundary value). If absent, the system

determines the type.

ACCURACY indicates the number of digits of accuracy desired in the solution. If absent, then either the accuracy specified by an accuracy statement (if present), or the standard system accuracy is used.

STEP indicates the initial step size to be used (when meaningful). If absent, the initial step size is determined by the accuracy desired.

Examples

i) SOLVE TAN(X) = 2X-A, FOR X, WITH A=PI, ON
0 < X < PI;

This finds the unique solution of $\tan X - 2X + \pi = 0$ on the interval $(0,\pi)$.

ii) EQ.1: X↑2 + Y↑2 = 4, EQ.2: X = (Y-1.5)↑2
SOLVE EQ.1, EQ.2, FOR X, Y, ON 0 < X AND 0 < Y,
TYPE POLYNOMIAL SYSTEM;

This finds all solutions of the system:

$$X^2 + Y^2 = 4,\ X = (Y-1.5)^2$$

which fall in the first quadrant. If NUMBER 1 were used, only one solution would be obtained.

iii) SOLVE A X = LAMBDA X, FOR LAMBDA, X,
WITH A=([3,3],-1,0,0,3,2,0,-1,-1,1),
ACCURACY 5 DIGITS, NUMBER 3;

This will obtain all 3 eigenvalues and eigenvectors of

$$\begin{bmatrix} -1 & 0 & 0 \\ 3 & 2 & 0 \\ -1 & -1 & 1 \end{bmatrix}$$

LAMBDA will be set equal to the vector (-1,2,1) and X will be the 3 by 3 array with eigenvectors as columns:

$$X = \begin{bmatrix} k_1 & 0 & 0 \\ -k_1 & k_2 & 0 \\ 0 & k_2 & k_3 \end{bmatrix} \quad \text{where } k_i \neq 0,\ i = 1,2,3$$

Boolean Expressions

There are no boolean or logical variables in NAPSS. However, a boolean expression may be formed by connecting two arithmetic expressions with one of the relational operators =, ¬=, <, <=, >=, >.

When two symbols are used to create a single relational operator, the symbols may appear in either order. Thus >= is equivalent to =>.

The operands of the relational operators may be either arrays or scalars. If the operands are two arrays they must be of equal size and the operation is performed element by element.

Boolean expressions may be connected with one of the binary logical operators AND or OR and negated with the logical operator ¬.

Conditional Statement

The NAPSS conditional statement is similar to the conditional statement in ALGOL, and has the form:

IF B.E. THEN S_1, S_2,..., S_n ELSE T_1, T_2,..., T_m;

where B.E. is a boolean expression and S_1, S_2,..., S_n and T_1, T_2,..., T_m are NAPSS statements.

In addition to permitting implicit continuation, the semi-colon at the end of the conditional statement solves the "dangling ELSE" problem. When there is no ELSE clause the semi-colon is placed after the statement S_n.

Examples

i) IF X = 2 THEN IF X = 3 THEN Y ← 4; ELSE Y ← 5;

ii) IF X = 2 THEN IF X = 3 THEN Y ← 4 ELSE Y ← 5;;

In example i) the ELSE clause is associated with the first IF because the second IF is terminated after its THEN clause with a semi-colon. In example ii) the ELSE clause is associated with the second IF.

Iteration Statement

The iteration statement has the form:

I.S. DØ S_1, S_2,..., S_n;

where I.S. represents one of the many forms of an iteration specification and S_1, S_2,..., S_n are NAPSS statements. The extent of the iteration is indicated by the semi-colon.

The various iteration specifications are a generalization of those appearing in ALGOL:

i) FOR T←0, 1, 16, -3, 5 (T assumes values 0, 1, 16, -3, 5)

ii) FOR Q←.1 TO .9 BY .3 (Q assumes values .1, .4, .7)

FOR Q←-2 TO 2 (Q assumes values -2, -1, 0, 1, 2)

FOR Q←2 TO -2 (Q assumes values 2, 1, 0, -1, -2)

FOR Q←-3, -1,..., 6 (Q assumes values -3, -1, 1, 3, 5)

The last example is equivalent to FOR Q←-3 TO 6 BY (-1-(-3))

iii) Any combination of expressions from above which follow the ← :

FOR C←0, 1, 16, -3, 5, .1 TO .9 BY .3, -2,

TO 2, -3, -1,..., 6, 2, TO -2

iv) FOR 72.4 TIMES (loop is executed 72 times)

v) WHILE X>0 OR Y<1 (loop is executed while the boolean expression is true)

vi) UNTIL |Z-Y| = 1 (loop is executed until the boolean expression is true)

vii) Any combination of FOR with WHILE or UNTIL:

In this case the loop is executed until one of the conditions is satisfied.

FOR Y←0 TO 6X+3 OR WHILE W<.001

FOR Z3←1 TO 10, 15 TO 100 BY 5 OR UNTIL X↑2<.5¢-6

(Note: the ¢ in NAPSS is equivalent to the E in FORTRAN.)

If a loop is controlled by more than one index, each of which assumes the same values, then the iteration can be written as follows:

```
FOR J, K←1, 2,..., M DO X[K,J]←1/(K+J);
```

This is equivalent to

```
FOR J←1, 2,..., M DO
FOR K←1, 2,..., M DO
X[K,J]←1/(K+J);;
```

Iteration Variables

Since many of the iterative methods in numerical analysis test successive values of a variable for termination, iteration variables are included in NAPSS. X represents the current value of X, X↓-1 represents the previous value of X, X↓-2 the value before that and so on. The number of previous values retained for a variable does not change dynamically during execution since only negative integer constants may follow the ↓. Previous iterates may not be assigned values directly. They obtain values as X is assigned new values. If the type of X should change, for example from a scalar to an array, all previous iterates for X are set as undefined.

Example

To find a root of F(X) = 0, using Newton's method with GUESS as a starting value, we have:

```
X←GUESS
```

```
FOR 100 TIMES OR UNTIL |X - X↓-1| < .00005 DO
X←X - F(X)/F'(X);
```

The iteration terminates when two successive iterates agree to 4 decimal places, or after 100 iterations.

The appearance of the iteration variable X↓-1 in the UNTIL clause in the above example causes the boolean expression to be skipped until the loop has been evaluated once. In general, if an iteration variable, say X↓-5, appears in the boolean expression of a WHILE or UNTIL clause then the loop is executed 5 times before the boolean expression is evaluated. This allows all iterates to be properly initialized before any testing is performed.

Accuracy Statement

The accuracy statement permits the user to specify the number of digits he wants retained for all his variables, except those whose accuracy is specified in a declare statement or a solve statement.

The accuracy statement is an executable statement, so different accuracies can be used in various segments of a program. If no accuracy statement appears in a program the system default accuracy is used. The default accuracy is six digits.

Example

```
ACCURACY 8 DIGITS
```

This specifies that at least eight significant figures are retained for all variables. The polyalgorithms which carry out the numerical analysis are supposed to maintain this accuracy or to give diagnostic messages.

The polyalgorithms use either single precision or double precision arithmetic to achieve the accuracy requested for the result. Normal arithmetic expressions are also evaluated using single or double precision arithmetic. The assignment statements use the number of digits of accuracy requested, to decide if the result should be stored as a single precision or double precision value.

Type Statement

The type statement provides a means of printing the values of selected variables, functions and arithmetic expressions. If the value of a numeric variable or a named function is to be printed the system labels the output with the name of the variable or function. If the value of an arithmetic expression or an unnamed function is to be printed the system labels the value with the system generated statement number of the type statement.

The user can add his own titles and labels if desired by using strings. When a user defined label is associated with a quantity, the system labeling for that variable is omitted. The system supplied labeling is also omitted when string expressions are printed as titles.

The format used to print each value is supplied by the system. It is a function of the system accuracy in effect and the magnitude of the number to be printed.

Each item in a type statement starts on a new line. Figure 1 gives a portion of a NAPSS program and the output it generates:

```
    1.00   A ← ([-1:3], 11.2,41.2362,-13,16,15.92)
    2.00   F(X) = X↑2 + 0.1
    3.00   B←2 + (-1)↑.5
    4.00   TYPE A, B, 2 * B, A * A'
A[-1:3]:  11.2000, 41.2362,-13.0000, 16.0000, 15.9200
B =  2.00000 + 1.00000 I
ANSWER, LINE    4.00 = 4.00000 + 2.00000 I
ARRAY[-1:3,-1:3]:
ROW 1 = 125.440, 461.845,-145.600, 179.200, 178.304
ROW 2 = 461.845, 1700.42,-536.071, 659.779, 656.480
ROW 3 =-145.600,-536.071, 169.000,-208.000,-206.960
ROW 4 = 179.200, 659.779,-208.000, 256.000, 254.720
ROW 5 = 178.304, 656.480,-206.960, 254.720, 253.446
   5.00  TYPE FUNCTION(F(X),ON (0,1,.5))
F(X):
F( .000000)= .100000
F( .500000)= .350000
F( 1.00000)= 1.10000
   6.00  TYPE FUNCTION(X↑2-B, IN X, ON(0,1,.5))
FUNC(X):
FUNC( .000000)=-2.00000 - 1.00000 I
FUNC( .500000)=-1.75000 - 1.00000 I
FUNC( 1.00000)=-1.00000 - 1.00000 I
```

Figure 1. Sample Output from a NAPSS Program.

The numbers preceding each of the statements in Figure 1 are statement numbers generated by the system. These numbers are used if the program needs to be edited and to label output.

Several items may be grouped inside of pointed brackets to form a single item. This causes all the items inside the brackets to be printed on the same line. Figure 2 gives an example of this and user labeled output.

```
    1.00  A ← 2, B ← -3.5
    3.00  S1 ← "THIS IS A", S2 ← " SAMPLE TITLE"
    5.00  TYPE S1||S2, <"A = ",A,"B = ",B,"A*B↑2 = "A*B↑2>
THIS IS A SAMPLE TITLE
A = 2.00000  B = -3.50000  A*B↑2 = 24.5000
```

Figure 2. User Labeled Output

Declarations

The declare statement is optional in NAPSS, since variables can be contextually declared when they are on the left in an assignment statement. However, some or all of the attributes of a variable can be explicitly assigned in a declare statement. The advantage of this is that the declared attributes must agree with the attributes of any value assigned to the variable. If they do not, no assignment is made and an error message is printed.

The attributes which are not explicitly declared for a variable are assigned contextually during execution when the variable is assigned a value.

The attributes which can be associated with a variable are: REAL, COMPLEX, SINGLE, DOUBLE, SCALAR, ARRAY, FUNCTION, NUMERIC, STRING, LOCAL, GLOBAL, INITIAL.

Two attributes cannot be assigned contextually: LOCAL, GLOBAL.

The declaration statement in NAPSS is an executable statement. Thus the attributes explicitly assigned to a variable can be changed dynamically. When possible the value of a variable is modified to conform with the new attributes.

Example

```
DECLARE A NUMERIC SINGLE ARRAY;
        .   .   .
DECLARE A COMPLEX SCALAR;
```

A has the attributes COMPLEX SCALAR assigned explicitly to it after the second declare statement has been executed. Also A is undefined since it changed from an array to a scalar.

The attributes SINGLE and DOUBLE are available to permit a user to selectively suppress the global accuracy which is established by the accuracy statement or the system's default accuracy if no accuracy statement is present.

The attribute DOUBLE is not associated with a variable contextually unless the accuracy specified for all variables in the system requires it. Thus the appearance of a double precision variable in an arithmetic expression does not imply that the double precision result will be assigned to the variable on the left of the

assignment statement. The double precision result is only assigned when the variable on the left explicitly has been declared to be double precision, or the accuracy currently in effect requires the use of double precision.

This scheme permits selected variables to have double precision values and arithmetic expressions involving these variables to be performed in double precision arithmetic while not propagating the attribute DOUBLE to all variables.

The attribute INITIAL permits the assigning of initial values to arrays or scalars only. The initial values are assigned every time the declare statement is executed, unless a variable name is declared to be GLOBAL. When this is the case initial values are only assigned when:

i) the variable name has no values presently associated with it.

ii) the other attributes, explicitly declared in addition to GLOBAL, cause the previous values associated with the variable name to be destroyed.

Example

```
DECLARE A REAL INITIAL (5), B(3) SINGLE
  INITIAL (1,2,3);
```

A is set equal to 5 and B is set equal to the vector (1,2,3).

Procedures

External and internal procedures may be written in NAPSS. This facility is included to give the language the power of a procedural language; however, its use is optional. Thus the casual user need not be concerned with the artificial rules that procedures introduce, for he can employ the system on what is called the console level.

On console level the user's program does not contain any procedures. Statements entered at console level are normally executed as they are received.

A procedure may be defined at any point in a program, and may be referenced in the program as both a subroutine and a function.

As mentioned above, the attributes LOCAL and GLOBAL cannot be assigned to a variable contextually. They need only be used when procedures are employed.

If a variable, XNAME, is declared LOCAL anywhere in a procedure, APROC, it signifies that XNAME is a new variable distinct from variables with the same name in procedures containing APROC. All occurrences of XNAME in APROC refer to the same variable until XNAME is assigned either the attribute LOCAL or GLOBAL in a procedure which is internal to APROC. A variable may not be assigned both

the attributes LOCAL and GLOBAL in procedure APROC, excluding procedures which are themselves internal to APROC.

The declaration of a variable to be GLOBAL has the same effect as declaring it to be LOCAL except that all occurrences of the variable in other procedures where it has been declared GLOBAL refer to the same variable.

The scope of variable names which are not declared to be LOCAL or GLOBAL and are not parameters is the outermost containing procedure.

The attributes LOCAL and GLOBAL are the only two attributes which are assigned at compile time. This permits the declare statement to appear anywhere in a procedure and allows the scope to be fixed. The other attributes are assigned when the declare statement is executed.

In Figure 3 the variable names Z and K in statement L2 of procedure EXTERNAL1 and in statement L3 of procedure INTERNAL1 refer to the same variables, but the variable names Z and K declared in procedure INTERNAL2 refer to different variables.

The variable names A and D in INTERNAL1 refer to different variables than the variables named A and D in EXTERNAL1, INTERNAL2, and EXTERNAL2. But since the variables named A and D in INTERNAL2 and EXTERNAL2 have the attribute GLOBAL, they refer to the same variables.

A procedure invoked as a subroutine, INTERNAL2, may be exited by encountering the end statement of the procedure or by executing a return statement. In this case an arithmetic expression associated with the return statement is ignored.

A procedure invoked as a function, INTERNAL2, can only be exited by executing a return statement which has an arithmetic expression associated with it. If this is not the case, the function returns with its value undefined.

```
EXTERNAL1: PROCEDURE
           DECLARE (A,D) REAL, E GLOBAL;
       L2: A=Z-4 * K
       INTERNAL1: PROCEDURE (B)
             INTERNAL2: PROCEDURE (M)
                              . . .
                        DECLARE (A,E,D,K) GLOBAL, Z LOCAL;
                              . . .
                        RETURN A↑Z
                        END
                   L3: Z=G * K + INTERNAL2 (3)
                              . . .
                        DECLARE (A,E,D) LOCAL;
                        CALL INTERNAL2 (A)
                        END
                              . . .
                  END
EXTERNAL2: PROCEDURE
           DECLARE (A,E,D) GLOBAL;
                    . . .
           END
```

Figure 3. Sample Procedures

INTERPRETER STRUCTURE

Over-all Structure of the System

The internal structure of the NAPSS system consists of four major modules: the supervisor, the compiler, the editor, and the interpreter. This is further subdivided into twenty-five overlays: one is the supervisor, three compose the compiler, one is the editor, and nineteen compose the interpreter. Figure 4 gives a skeleton of the overlay structure.

The NAPSS system is written almost entirely in machine independent FORTRAN. The few machine dependent operations are restricted to "black-box" modules coded in assembly language. This is done to aid the goal of machine independence for the system.

Due to the equipment and associated software available, the current version of NAPSS does not operate in a time sharing environment. But the implementation techniques do not preclude such an extension.

The current system is running on the Control Data 6500 at Purdue University.

The supervisor controls the flow into each of the three other modules. It distinguishes between NAPSS source statements, which are processed by the compiler, and edit statements, which are processed by the editor.

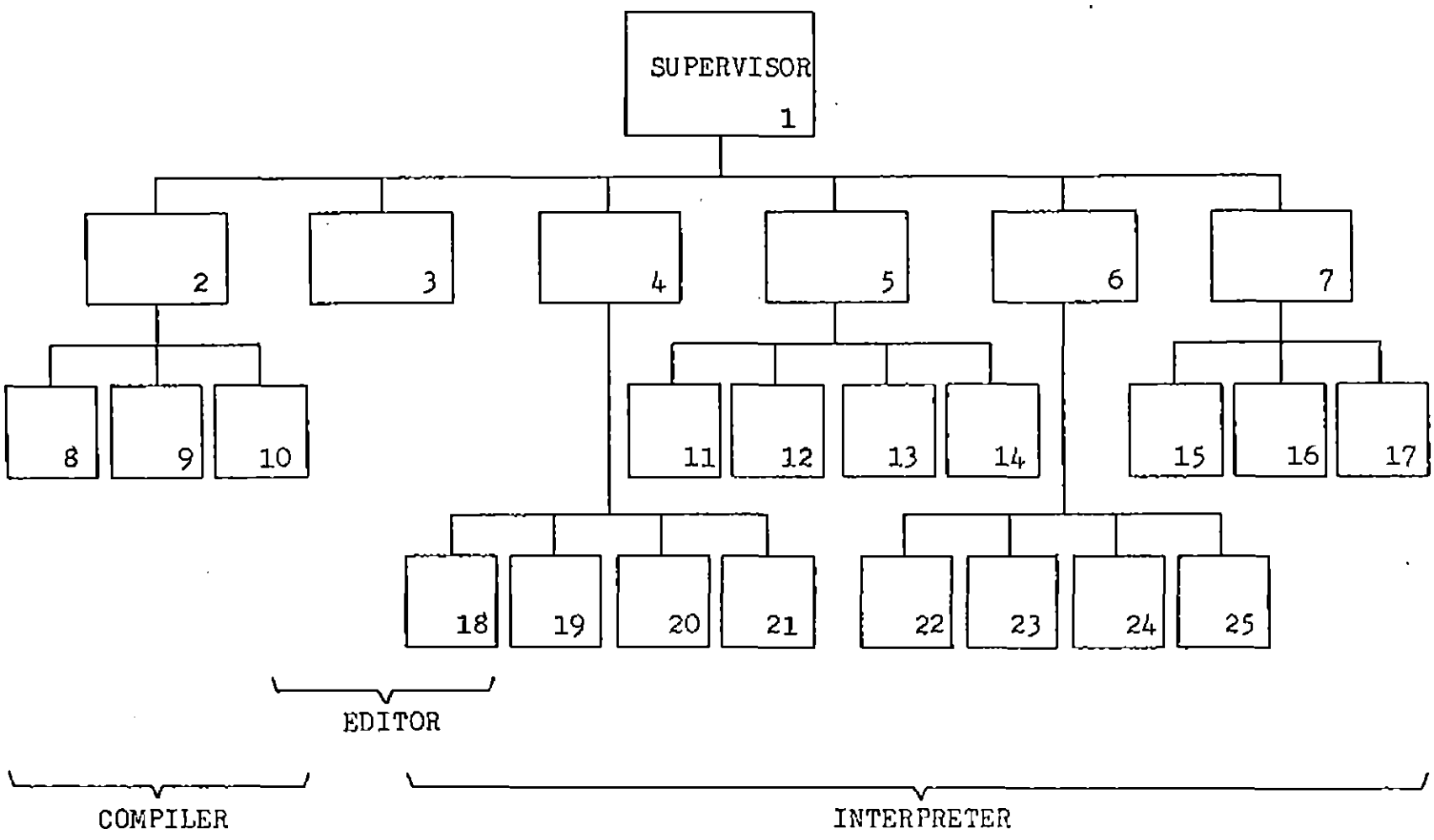


Figure 4. Overlay Structure of the NAPSS System

The supervisor is also responsible for invoking the interpreter when a NAPSS statement is to be executed.

NAPSS source statements are transformed by the compiler into an internal text which the interpreter processes. This scheme is adopted for several reasons. First, the complexity of the elements to be manipulated and the absence of declarations require execution time decoding of operands. Second, it easily allows for extensions to the system. Third, it gives the user incremental execution. Fourth, it permits extensive error diagnostics and permits error corrections without having to recompile the whole program. Fifth, statements which are repeatedly executed are only translated into internal text once.

The internal text for each statement consists of twenty-bit words. The internal and source text for each statement is stored in secondary storage. When a statement is to be executed, a copy of its internal text is passed to the interpreter. This reduces considerably the core storage required for a user's programme. Since the system is intended for use in an incrementally executing mode, no reference to secondary storage is normally required to obtain the internal text of a statement.

The internal text generated for arithmetic expressions (Appendix B) is a form of three address code. All operators, pointers, and references to temporary variables are represented by negative integers while all references

to user-created variables are represented by positive integers. This is done so that the equals assignment statements can easily detect references to user variables.

The system has two modes: suppress mode and execute mode. In the suppress mode, each statement is compiled into internal text and the internal and source text is saved on secondary storage for later execution. Suppress mode is entered by typing the statement .SUPPRESS. A block of statements which have been compiled in suppress mode may be executed at any time by typing the statement .GO.

The normal mode of execution is execute mode. Here, each statement is executed immediately after it has been compiled and a copy of its internal and source text saved in secondary storage. The system automatically enters suppress mode when the user starts a compound statement (a FOR statement) or starts a procedure. This is necessary because a compound statement cannot be executed until the whole statement is received and a procedure is only executed when invoked. The system re-enters execute mode automatically as soon as the compound statement or procedure is completed.

In the remainder of this chapter the various components of the interpreter are described.

Real and Virtual Memory

The memory of a NAPSS program is made up of a few pages of real memory which reside in core and a larger number of pages of virtual memory which reside in secondary storage and are brought in and out of real memory. Two vectors (one dealing with virtual and the other with real memory) and several pointers are used to keep track of real and virtual memory.

Each element in the virtual memory vector is subdivided into three twenty-bit bytes. The first byte contains a flag indicating whether the page contains internal text or name control blocks. The second byte is a switch, used when the page is in real memory to indicate whether or not a copy of the page also resides in secondary storage. The third byte contains the real page number the virtual page is in when it is in real memory.

The elements of the virtual memory vector which denote available pages are linked together. Initially, the element for virtual page one points to the element for virtual page two and the last element contains a zero. When a page of virtual memory is returned to the system its element is again linked to the top of the list of available virtual pages.

The real memory vector elements contain one entry per real page. This entry is the number of the virtual page occupying it (zero if it is empty). This pointer

from real memory to virtual memory is used when a new virtual page is placed in a real page. The virtual page currently in the real page must be copied out into secondary storage if it is not there already.

The amount of core assigned to real memory is dynamic. Pages are removed from the top and bottom of real memory in order to obtain contiguous blocks of storage. Pages are removed from the top of real memory for two purposes: first, to expand the name table, and second, to obtain space for the work pool. Pages are removed from the bottom of real memory to obtain space for local name control blocks during the evaluation of left arrow functions. See figure 5.

The work pool is used to hold arrays when performing array arithmetic. Requests for work pool space are always made in terms of words. However, the amount of real memory assigned to the work pool is always an integral number of pages. When a request is made for work pool space and the work pool is empty, the space supplied is zeroed. When space is requested for the work pool and the work pool is not empty, one of two situations arises. First, the space requested is less than the current size of the work pool. If the difference between the space requested and the current size of the work pool amounts to one or more pages, a corresponding number of pages is returned to real memory from the bottom of the work

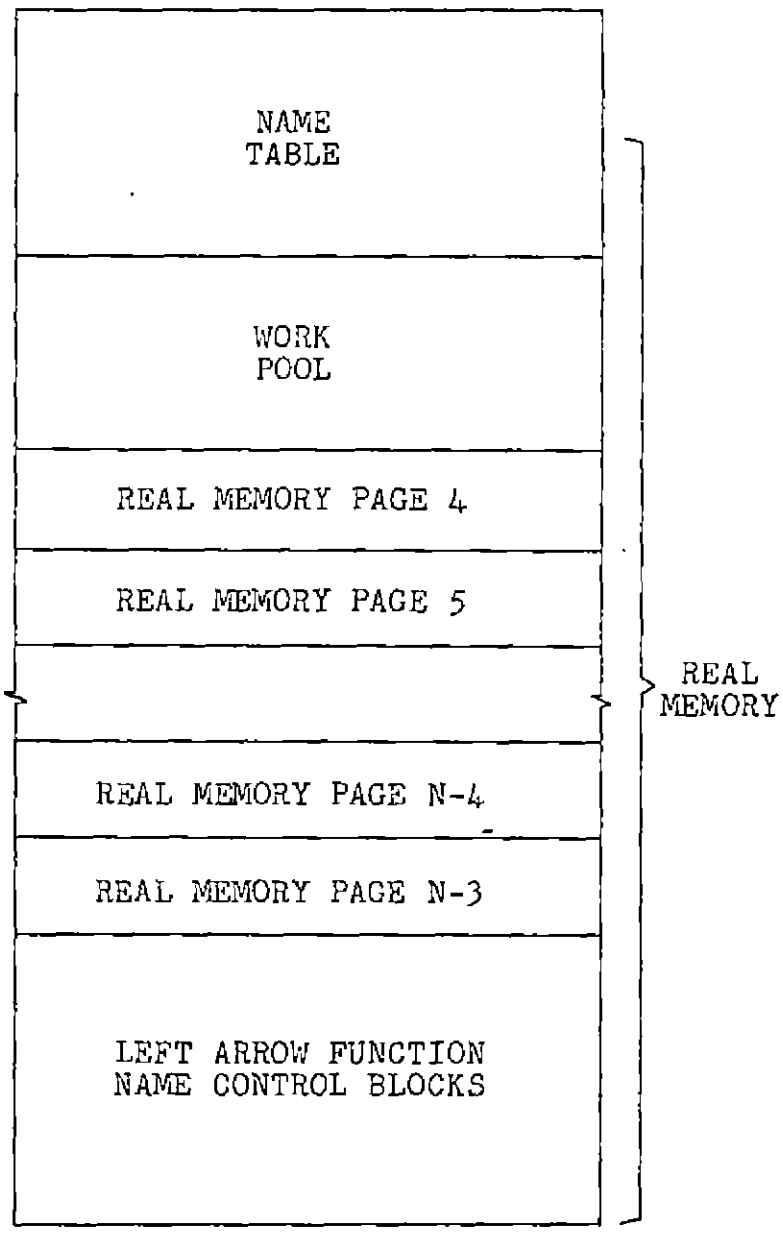


Figure 5. Real Memory Organization

pool. Second, the space requested exceeds the current size of the work pool. If additional pages are obtained from the real memory to satisfy the request, they are zeroed.

Virtual pages are assigned to real pages sequentially. Thus a virtual page is not removed from real memory until all real pages are assigned a virtual page. This sequential process may be broken whenever space is assigned to the work pool or to hold the local name control blocks for a left arrow function, since, after the space request is satisfied, the next real page to receive a virtual page may no longer belong to real memory. When this occurs the pointer to the next real page to be used is reset to the first page now in real memory.

The algorithm for bringing virtual pages into real memory is further modified when the work pool returns a page to real memory. Since the page returned is empty, a virtual page may be placed in it directly, avoiding the possibility of having to save the virtual page currently there in secondary storage. Thus the normal sequential process is interrupted until all the pages returned to real memory by the work pool are re-used.

The system does not assign all of real memory to either the work pool or to space for a left arrow function's local name control blocks. A request for real memory space is honored as long as two pages remain

in real memory after the request is satisfied. If more space is requested than can be supplied, the request is modified to correspond to the maximum amount of space available. This permits the system to continue if this is adequate.

Two pages are required in real memory to facilitate the linking of virtual pages. With two pages in real memory the above algorithm guarantees that the previous and the current virtual pages referenced remain in real memory. Thus they may be linked together if necessary, without having to save pointers and re-read a virtual page to fill in link information.

Name Table

Associated with each procedure is a name table containing entries for each variable, label, constant and procedure name appearing in that procedure. The entries are called name control blocks and are created when the name is introduced.

A name control block is made up of seven sixty-bit words, or twenty-one twenty-bit words called bytes. See Figure 6.

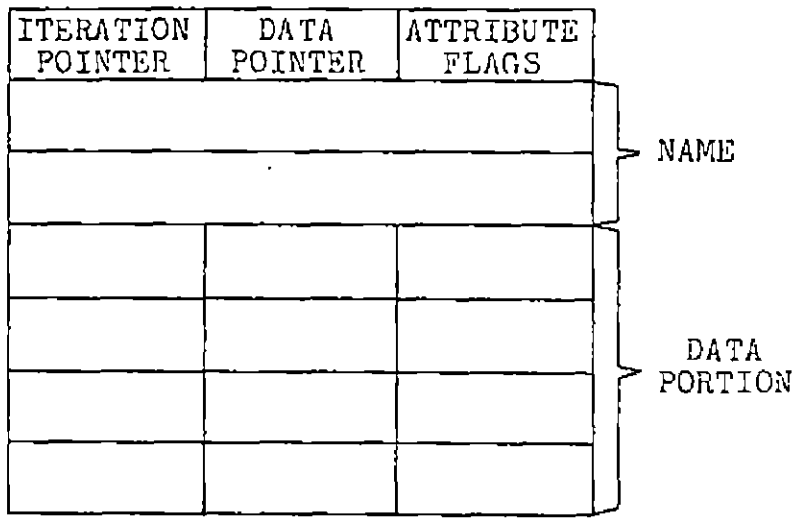
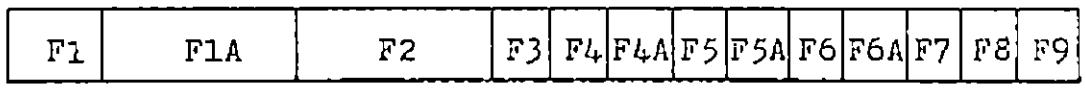


Figure 6. The Layout of a Name Control Block

Byte one contains the iteration pointer, if the variable is a member of an iteration chain (X, X↓-1,...). Byte two contains a pointer to the data for a variable if it is not a numeric scalar. Byte three contains the attribute flags. These flags encode the attribute number for the data type of the variable. See Figure 7.

During compilation only flags F1, F1A, F2, F8 and F9 are set. Flag F2 assumes only the values zero through five during compilation. A value of five specifies that the variable is used as a computational entity. Checking between the various types of computational entities is performed during execution when the variable is referenced as an operand.



F1 - Scope = 0 Unspecified
= 1 Parameter
= 2 Local
= 3 Global
F1A - Number of times declared Global
F2 - Type = 0 Formal Parameter
= 1 Constant
= 2 Statement Label
= 3 Equation Label
= 4 Procedure Name
= 5 Numeric
= 6 Symbolic Function
= 7 String
= 8 Complex Place Marker
= 14 Boolean True
= 15 Boolean False
F3 = 0 F2, Contextually Declared
= 1 F2, Explicitly Declared
F4 = 0 Scalar
= 1 Array
F4A = 0 F4, Contextually Declared
= 1 F4, Explicitly Declared
F5 = 0 Single Precision
= 1 Double Precision
F5A = 0 F5, Contextually Declared
= 1 F5, Explicitly Declared
F6 = 0 Real
= 1 Complex
F6A = 0 F6, Contextually Declared
= 1 F6, Explicitly Declared
F7 = 0 Left Arrow Defined Variable
= 1 Equals Defined Variable
F8 = 0 Variable Referenced During Execution
= 1 Variable Referenced Only By Compiler
F9 = 0 Variable Defined
= 1 Variable Undefined

Figure 7. The Arrangement and Specification of Attribute Flags in a Name Control Block Entry

Flags F1A, F3, F4A, F5A, and F6A are not used to determine the attribute number of the variable. These flags may be set only in a declare statement and are used to check attributes when an assignment is made.

Table 2 gives the various flags for each of the individual data types and the associated attribute numbers. Appendix A contains a description of how each data type is stored.

The next two words in a name control block contain the name of the variable. This limits the length of a variable name in NAPSS to twenty characters.

The next four words or twelve bytes contain either the value associated with the variable if it is a numeric scalar or information about the values if not.

For temporary variables (variables used to hold temporary results during the evaluation of an arithmetic expression) there is a fixed number of name control blocks pre-initialized in the system. These temporary name control blocks contain the same fields as a user-created name control block except for the name field.

During compilation a name control block is used as a name table entry. At this time it contains the name of the variable, some basic attributes describing how the variable appears in the program, and possibly an iteration pointer.

Table 2. Flags and Attribute Numbers of Various Data Types

DATA TYPE	FLAGS							ATTRIBUTE NUMBER
	F2	F4	F5	F6	F7	F8	F9	
SYMBOLIC SCALAR =	5	0	-	-	1	0	0	0
NUMERIC SCALAR ← SINGLE REAL	5	0	0	0	0	0	0	1
NUMERIC SCALAR ← DOUBLE REAL	5	0	1	0	0	0	0	2
NUMERIC SCALAR ← SINGLE COMPLEX	5	0	0	1	0	0	0	3
NUMERIC SCALAR ← DOUBLE COMPLEX *	5	0	1	1	0	0	0	4
NUMERIC ARRAY ← SINGLE REAL	5	1	0	0	0	0	0	5
NUMERIC ARRAY ← DOUBLE REAL	5	1	1	0	0	0	0	6
NUMERIC ARRAY ← SINGLE COMPLEX	5	1	0	1	0	0	0	7
NUMERIC ARRAY ← DOUBLE COMPLEX *	5	1	1	1	0	0	0	8
NUMERIC CONSTANT SINGLE REAL	1	-	0	-	-	0	0	1
NUMERIC CONSTANT DOUBLE REAL	1	-	1	-	-	0	0	2
IMAGINARY PLACE MARKER ($\sqrt{-1}$)	8	-	-	-	-	0	0	9
STRING SCALAR	7	0	-	-	0	0	0	10
STRING ARRAY *	7	1	-	-	0	0	0	11
BOOLEAN FALSE	14	-	-	-	-	0	0	12
BOOLEAN TRUE	15	-	-	-	-	0	0	13
STATEMENT LABEL	2	-	-	-	-	0	0	14
EQUATION LABEL	3	-	-	-	-	0	0	15
SYMBOLIC FUNCTION SCALAR ←	6	0	-	-	0	0	0	16
SYMBOLIC FUNCTION SCALAR =	6	0	-	-	1	0	0	17
SYMBOLIC FUNCTION ARRAY ←	6	1	-	-	0	0	0	18
SYMBOLIC FUNCTION ARRAY =	6	1	-	-	1	0	0	19
PROCEDURE NAME	4	-	-	-	-	0	0	20

* Not Implemented

During execution the name control block is used to hold values, pointers to values and a complete set of attributes for the variable.

This double usage of the name control block entries poses no problem if compilation and execution are performed separately. But in NAPSS the normal mode of operation is to execute each statement as soon as it is compiled. Thus, three situations are possible when a variable is entered in the name table. First, the variable may never have been used before in the program. Second, the variable may have appeared before in the program but no value has been assigned to it. Thus, it is just as it was when the compiler last saw it. Here a limited compatability check is made for the two uses of the variable in the program. For example, the use of the variable as a label and as a variable in an arithmetic expression is illegal. Third, the variable has appeared before in the program and has been assigned a value and a complete set of attributes. This enables more checking to be performed. The name table routine must not disrupt any of the attribute flags, for if any of the attribute flags are changed the attribute might no longer correspond to the value associated with the name control block.

The name table is constructed sequentially. This method requires a minimum amount of space, and permits

the name table to grow dynamically. But it requires the name table to be searched sequentially. The search goes through the name table from bottom to top. This is done because frequently the greatest percentage of references to a variable occur in the immediate vicinity of its definition.

A variable which is declared to be global in N different procedures has N+1 name control blocks associated with it. There is a name control block for the variable in the name table of each of the procedures in which it appears. Only compile time information and a pointer to the $N+1^{st}$ copy is contained in these name control blocks. The $N+1^{st}$ copy is in the global variable name table and contains a complete set of attributes for the variable and its value or pointer to its value.

The $N+1^{st}$ copy of a global variable's name control block is placed in the global name table when the first procedure is invoked in which the global variable appears, or when the variable is declared global on the console level (i.e. the portion of the program not contained in a procedure). When a global variable is added to the global name table and it already appears there, a check is made on the compatability of the attributes. An error results when they conflict. Otherwise a pointer to the $N+1^{st}$ copy is placed in the procedure's copy of the variable's name control block.

A count is kept in the global name control block of the number of procedures referencing the global variable. When a global variable is no longer referenced from any procedure or from the console routine, then its name control block is removed from the global name table and the storage associated with it is returned to the system.

A name control block is created for each iteration variable, eg. X, X↓-1, X↓-2. These name control blocks are linked together to form a chain. The iteration pointer field of the name control block of the head, X, points to the name control block for X↓-1, and the iteration pointer for the last name control block in the chain points back to the name control block of the head of the chain, X. A chain of iteration variables is constructed by placing the name control block of the head in the name table first, the name control block for X↓-1 next, and so on. This is done even if one of the iterates is referenced in the program first. Therefore, the name control blocks for an iteration chain are ordered in the name table. The name control block for the head of the chain (X) appears first and the name control block for the last iterate in the chain appears last. Thus the name control blocks for the various iterates in a chain are distinguishable by position.

A procedure is compiled when it is defined and in order to link it into the program the text generated uses

only relative pointers to name table entries, and all linking between entries in a procedure's name table is done with relative pointers. This allows procedure A, for example, to be compiled as external procedure and to be invoked either directly from the console level or from another procedure which itself is invoked from the console level. The name table for procedure A is placed in the name table after the last entry presently there and its base address is set up.

Variables which are not declared to be either local or global in an internal procedure are assumed to be known in the containing block.[1] After the procedure is compiled and a copy of its name table saved, a pass is made through the procedure's name table. This pass goes through the name table from top to bottom and places a copy of the name control block for each variable not declared to be either local or global in the name table of the containing block. If the variable has already been used in the containing block, a compatability check is made on the attributes.

During execution only one name control block is used for the value and attributes of a variable which is not declared to be local or global. This is the name control

---

[1] A block is either a procedure or the console level routine.

block entry in the outermost block. The name control block in the internal procedure is linked to this when the internal procedure is invoked. The linkage is constructed so that only one step is required to obtain the value of the variable regardless of the depth of the procedure.

Figure 8 is a portion of a NAPSS program written on console level with two internal procedures, INTERNAL1 and INTERNAL2. After the procedure statement (statement number 3) is executed, the status of the name table is depicted in Figure 9. CB is the base address for the console level name table. After the second procedure statement (statement number 6) is compiled the name table appears as in Figure 10. I1B is the base address for the name table of procedure INTERNAL1. At the end of INTERNAL2 (statement number 10) the name table appears as in Figure 11 (I2B is the base address for the name table of INTERNAL2). This end statement causes the name table of procedure INTERNAL2 to be saved along with the source code and internal text of the procedure. A check is made to insure that a copy of the name control block for each non-local and non-global variable in INTERNAL2 appears in the name table for INTERNAL1. The status of the name table after this is shown in Figure 12. The name control block for D was added to the name table of INTERNAL1 and the variables B and C were checked. The

```
 1   A ← 4
 2   B ← 5
 3   INTERNAL1: PROCEDURE
 4              DECLARE C LOCAL;
 5              C ← A + B
 6              INTERNAL2: PROCEDURE
 7                         DECLARE A LOCAL;
 8                         A ← C + B
 9                         D ← A ↑ B
10                         END
11             CALL INTERNAL2
12                A ← D ↑ C + E
13                END
14   E ← A + B
15   C ← E ↑ A
16   CALL INTERNAL1
```

Figure 8. Sample Program

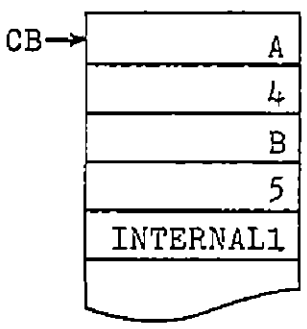


Figure 9.

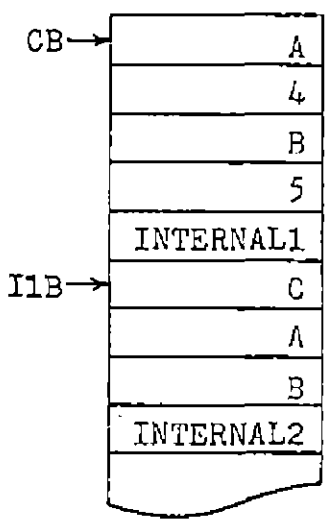


Figure 10.

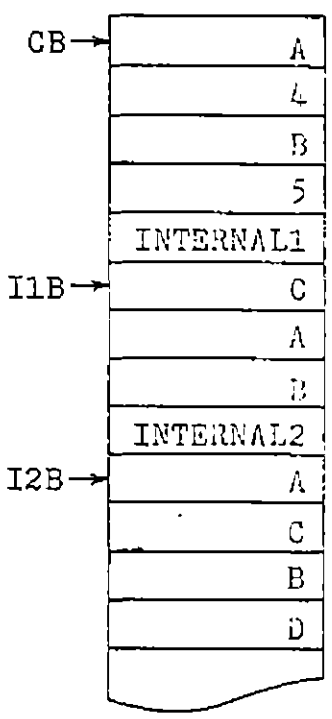


Figure 11.

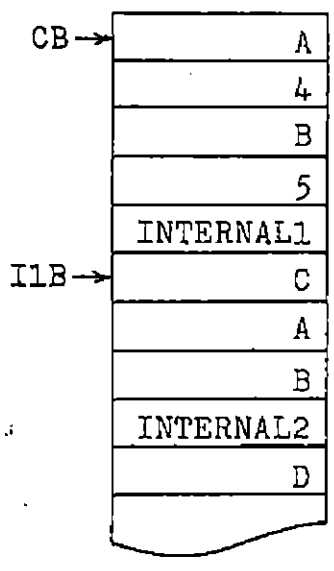


Figure 12.

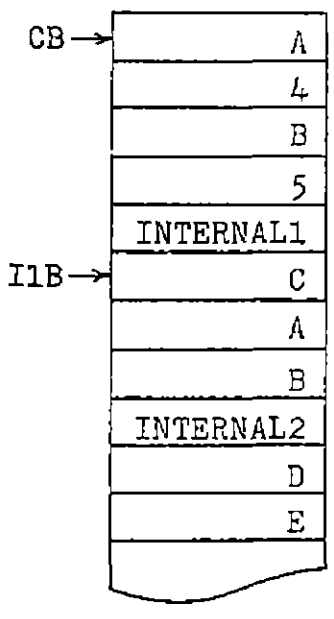


Figure 13.

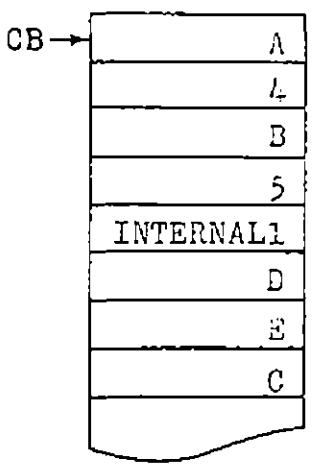


Figure 14.

Name Table Segments

use of variable A was not checked because A is declared to be local in procedure INTERNAL2. At the end of procedure INTERNAL1 the status of the name table is shown in Figure 13. Here the name table for INTERNAL1 is then saved along with the source and internal text for the procedure. Then the check made for INTERNAL2 is repeated for INTERNAL1 and the resulting name table is shown in Figure 14. This check causes D and E to be put in the name table of the console level routine.

The execution of statement number 16 invokes internal procedure INTERNAL1. Its name table is loaded immediately after the name table for the console level routine and the linkage is set up for the non-local, non-global variables in INTERNAL1, as shown in Figure 15. The execution of statement 11 invokes procedure INTERNAL2. The status of the name table at this point is given in Figure 16. The name control blocks for B and D in procedure INTERNAL2 are linked directly to the name control blocks for B and D on the console level.

There are three types of name control blocks in different memory areas: temporary, local for left arrow functions, and ordinary. See Figure 17.

A central routine is used to decode variable name control blocks during execution. This routine determines the type of the name control block and handles the linkage between global and non-local, non-global name

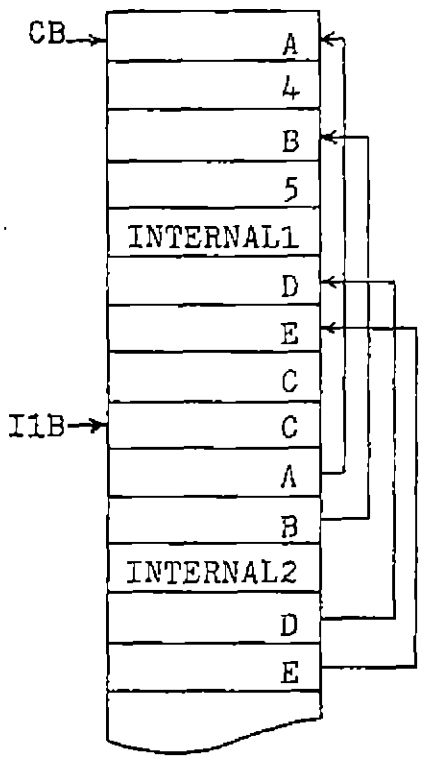


Figure 15.

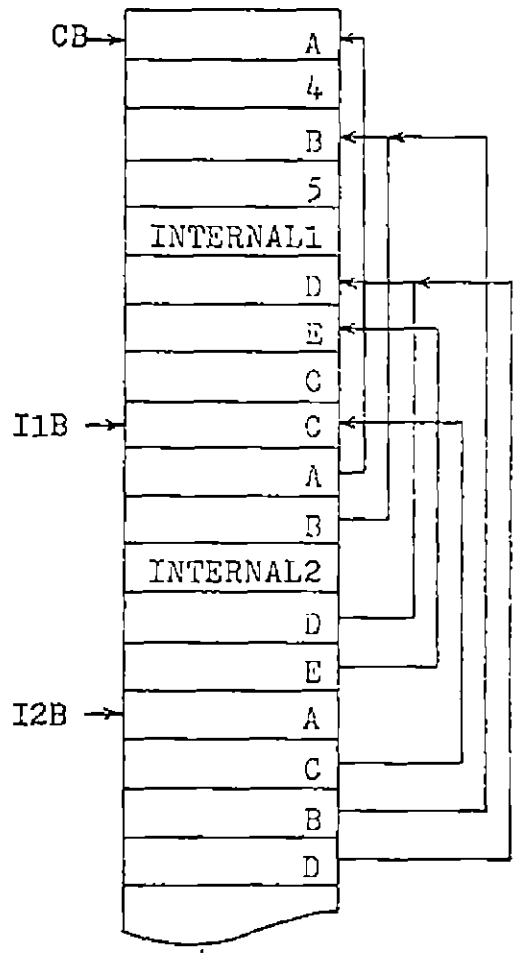


Figure 16.

Name Table Segments

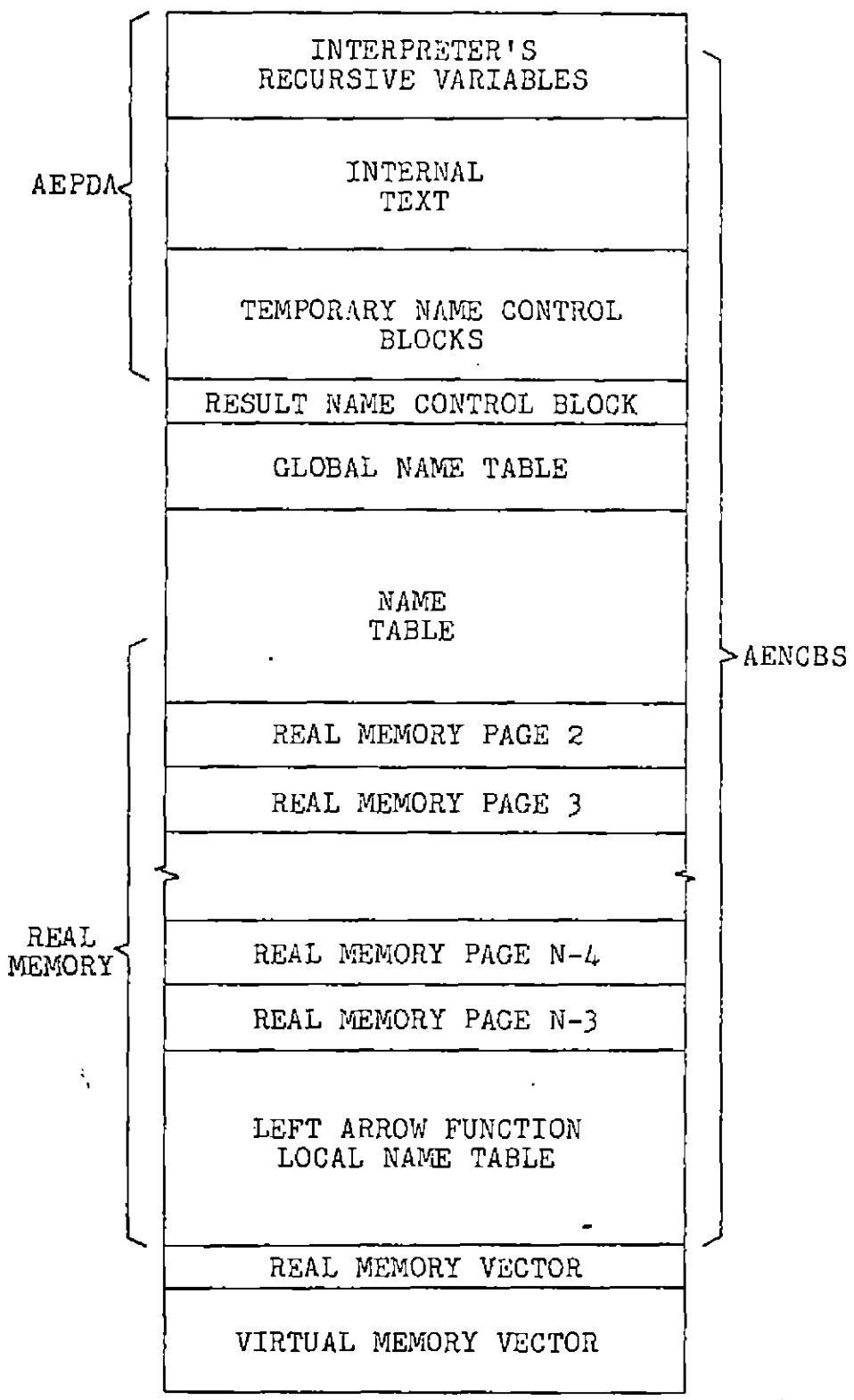


Figure 17. NAPSS Memory Organization

control blocks. Three things are returned when a name control block is decoded: the attribute number, the data pointer field and the index in the array AENCBS of the first word of the data pointer portion of the name control block. See figures 6 and 17.

Error Messages

Internally there are five severity levels for errors. Level one is a warning type error. Level two is a user-caused error such as incompatible operands. Level three is an undefined variable or function. Level four is a system error which the user can correct, and level five indicates a fatal system error.

Externally there are three error message levels available. The level may be changed dynamically by the user. On level one warning messages are ignored and only the numbers associated with other error messages are printed. On level two warning error message numbers are printed along with the messages and number of more severe errors. Level three prints the messages and numbers for all errors.

A warning message is printed by the routine which detects the error. When an error with a severity number greater than one occurs, the routine detecting the error sets a flag to the severity number of the error and returns to the routine from which it was called. This routine returns to the place from which it was called. This process is repeated until the interpreter supervisor is

reached.

Before each routine returns in this process, it must restore itself and return to the system any storage it is using to hold temporary results.

This method is used because the occurrence of an error, with an associated severity number greater than one, prevents the interpreter from continuing until the error has been corrected. This method permits the use of a common error message routine for all non-warning messages.

In addition to setting the severity flag, the routine which detects the error sets the number of the error and possibly some entries in the vector INSERTS, Figure 18. The entries from the vector INSERTS are inserted into the error message to give the user specific information about what caused the error.

The error message routine uses three vectors to construct the text for the various error messages. One vector (DICTIONARY) contains a list of all possible words and phrases required to construct any error message. The second vector (MESSAGE CODE) is broken down into fields of four octal digits. These fields contain the indices of the words to be used from DICTIONARY for the message and information on what is to be inserted in the message from INSERTS. The third vector (MESSAGE POINTER) contains the index of the start of the various messages in MESSAGE CODE. It is indexed by the error message number, Figure 18.

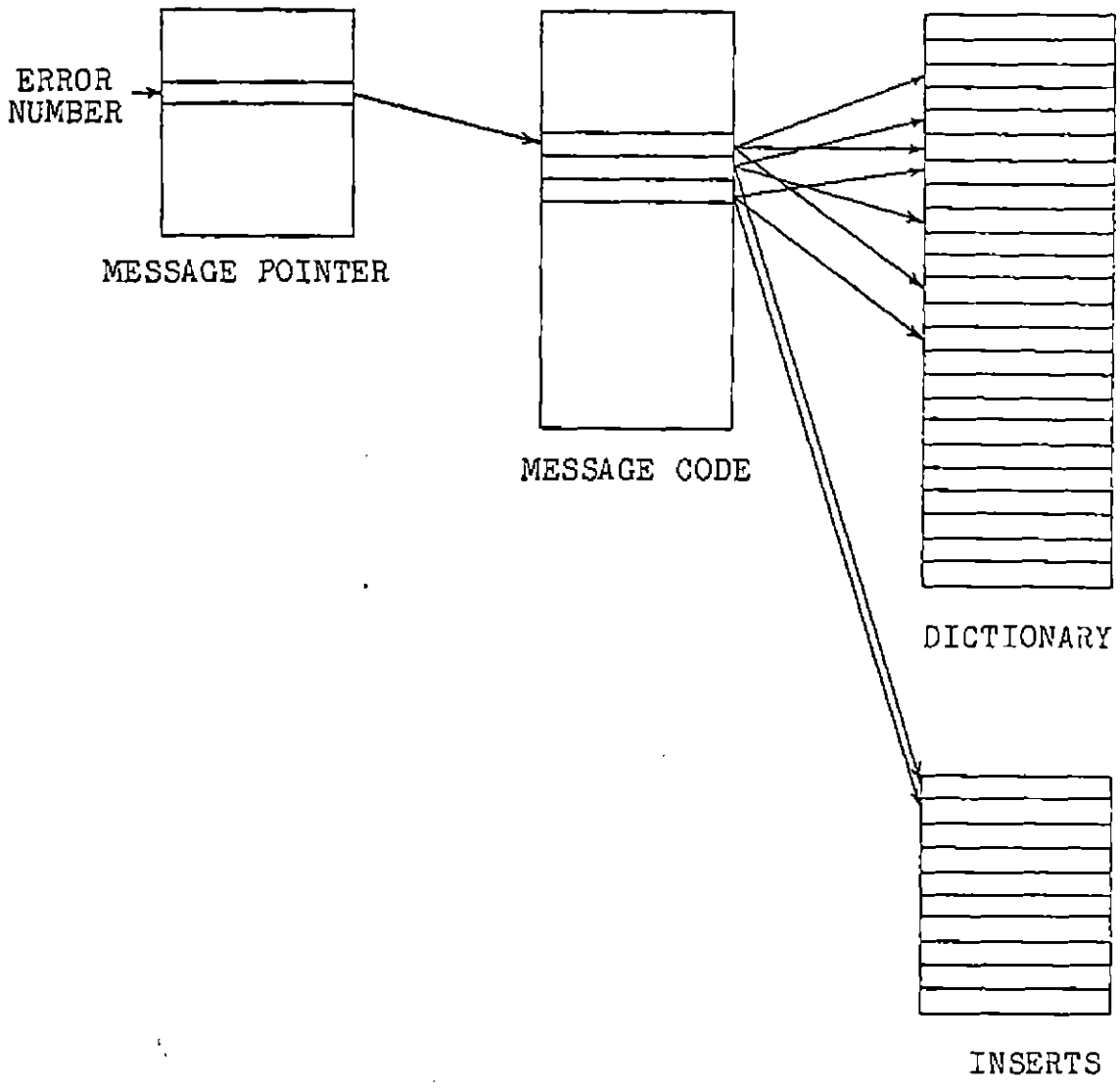


Figure 18. Vectors used for Error Message Construction

Each field in MESSAGE CODE specifies the next entry of the error message. If digit number one of a field is non-zero, then the entry is a word or phrase from DICTIONARY, and digit number one is the number of consecutive words to be obtained from DICTIONARY and digits two, three, and four form the index of the first word of the entry in DICTIONARY. If digit number one of a field is zero and the next three digits are not all zero, the next entry is an insert. Digits two, three, and four are used to encode information about the insert. If the field is all zeros, this indicates the end of the error message.

ARITHMETIC EXPRESSION EVALUATOR

Evaluation of Arithmetic Expressions with Non-Recursive Operands

The flow of control in the arithmetic expression evaluator for expressions which do not involve recursive variables, function evaluations or calls on poly-algorithms is given in Figure 19.

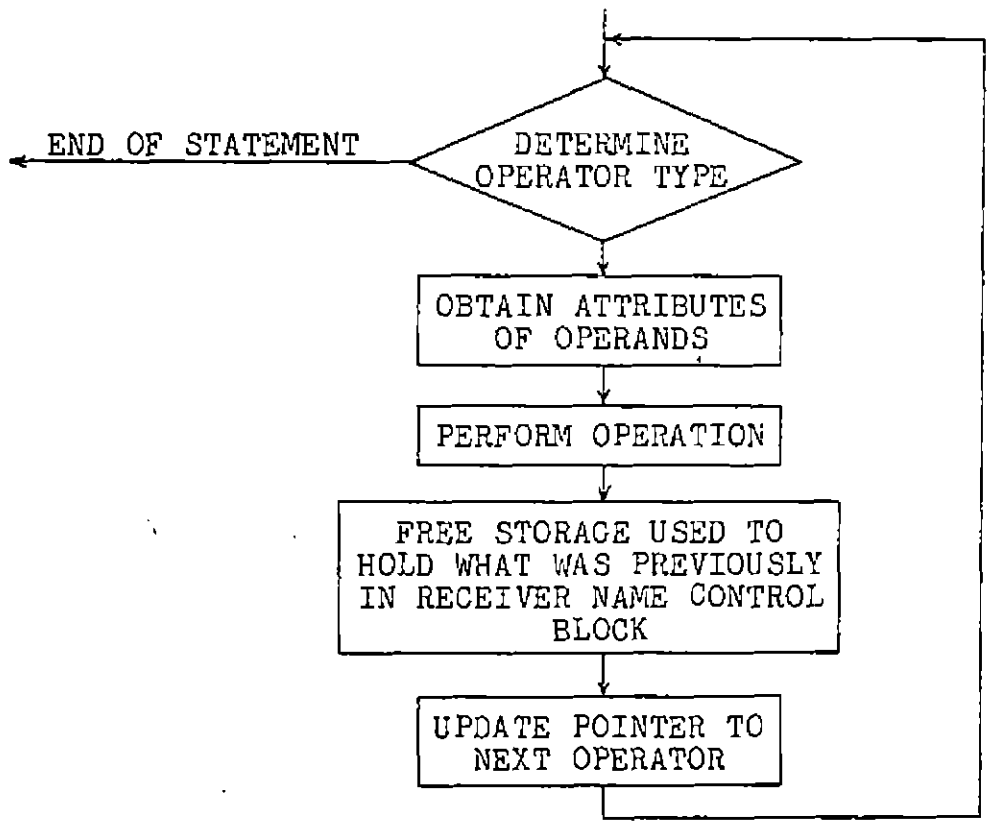


Figure 19. Flow of Control in Arithmetic Expression Evaluator

The operators are tested for in a fixed order so that the ones most frequently occurring are tested first.

The attribute or type of an operand must be determined at execution time because attributes are not associated with variables during compilation. They are associated during execution time and may dynamically change during the execution of the program.

If NAPSS had required that all attributes be either always declared or always contextually defined instead of allowing the user to declare some attributes and have the rest associated contextually, the attribute field of a name control block could have contained a simple attribute number. However, because of the mixture permitted the attribute field contains a set of flags from which an attribute number is decoded.

After the attributes of the operands have been determined the attribute of the result is obtained by a table look up, using the attributes of the operands and the operator as indices.

To eliminate the work necessary to obtain the attribute of an operand, a look ahead scheme is used where possible. If the result of an operation is an operand of the next operator, then the attribute of that operand is flagged as being known. This scheme, even though only local, is quite useful for frequently the result of the previous operation is an operand of the next operator.

There are three types of numeric scalars in NAPSS: real single precision, real double precision, and complex single precision. Integers are stored internally as real numbers. When an integer is needed, such as for a subscript, the system converts the real number to the nearest integer.

With only three types of numeric scalars the number of addition routines needed to permit all possible combinations of operands is 3^2. If a fourth data type, double precision complex, were added the number of routines needed would be 4^2 or an increase of 77 percent. For this reason, double precision complex numbers are not now provided in NAPSS.

For scalar arithmetic NAPSS does not use 3^2 routines for each of the basic binary operators but rather only 3. This is achieved by converting one of the operands to match the attribute of the other. The scalar operands are placed in a work area before the operations are performed. The conversion is performed during transfer to the work area by zeroing a word when necessary.

For array arithmetic the number of routines needed to perform the various operations cannot be reduced to the same extent as for scalar arithmetic. This is because of the time needed to convert one operand to match the other and the increase in memory required to hold the operands. The number of routines needed to perform the binary array

operations is 3^2 for multiplication and 2x3 for addition and subtraction. The number of routines needed to perform addition and subtraction is reduced more than for multiplication by taking into account the similarity between data types. The routines which perform the array arithmetic are machine language routines.

Arrays are stored permanently in a random file and are brought into memory only when needed. The empty records in this file are chained together so that when a record is requested and the file is full, the user can be asked to free a variable holding an array to allow his program to continue.

Actual array arithmetic is performed in an area called the work pool. When an array operation is to be performed enough space is assigned to the work pool to hold the operands and resulting arrays.

The result of the array operation is not immediately put out in the random file with the other arrays. Rather, the work pool remains intact with the operands and the result left in it. When the next array operation occurs the work pool is checked to see if it is empty; if not, the operands are compared with what is currently in the work pool. If the result of the previous array operation is an operand of the present array operation, then the result array need only be written out into the array file if it is an operand of a future operation.

The work pool is completely emptied at the end of each statement. Therefore, the process of optimizing the manipulation of arrays is only performed locally. The reason for this is that the work pool is used to manipulate other data types in addition to arrays.

When performing array arithmetic the system checks to see if the operands conform. The values of the index bounds of the operands do not affect the operations if the number of elements in the corresponding dimensions agree. For example, it is illegal to multiply two row vectors or to add a row vector and a column vector. The system does not attempt to determine what the user intended in these situations. Rather, it gives an error message and asks the user to clarify the meaning of the statement.

The index bounds of a result array take their values from the bounds of the operand arrays. There is one exception to this: when two arrays are added or subtracted and their index bounds are not identical, the lower bounds of the result array are set to one.

If the result of an array operation is a one element array it is not treated as an array by the system, but is stored as a scalar.

A temporary variable may be assigned several values during the evaluation of an arithmetic expression. This would pose no problem if all the results were scalars,

for scalar values are stored in the name control block for the variable. However, the name control blocks for other data types only contain pointers to where the values are stored. This causes the problem of when to free the storage used to hold temporary results. Storage can be returned to the system periodically using a garbage collection scheme, or storage can be returned immediately at the point it is no longer referenced.

Storage is freed by the NAPSS interpreter immediately after a new value is assigned to the temporary variable, thereby permitting an operation to have the same temporary variable as an operand and as a result this scheme has two main advantages: first, the type of storage to be freed is known at this point; second, the time required to free storage is uniformly consumed. The latter is of importance since the system is intended for use in an on-line incrementally executing mode.

The arithmetic expression evaluator is called from various places in the interpreter and not just to evaluate arithmetic expressions appearing to the right of assignment statements. For this reason and to facilitate recursion the result of an evaluation is associated with a fixed temporary name control block. The results of every arithmetic expression evaluation may be obtained from this temporary name control block by whatever portion of the interpreter requested the evaluation.

The name control block which receives the result of an arithmetic expression evaluation is only used to pass the value along to whatever portion of the interpreter invoked the arithmetic expression evaluator. Thus, the storage associated with its previous value is not returned to the system. If the storage associated with the result temporary name control block, Figure 17, is freed each time a new value is associated with it, storage would be returned which may now be associated with a user variable or which has already been freed by some other portion of the interpreter.

Evaluation of Arithmetic Expression with Recursive Operands

The occurrence of an equals variable in an arithmetic expression evaluator to recurse. The recursion needed to evaluate equals variables is limited to one routine, the master controller, Figure 20. This routine is responsible for determining what the next operator is, what the attributes of the operands are, and what routine is to be invoked to perform the operation.

The routines which perform the various operations expect to receive pointers to where the actual values of the operands may be obtained. This causes the master controller to evaluate the expression associated with the equals variable before calling the operator routine.

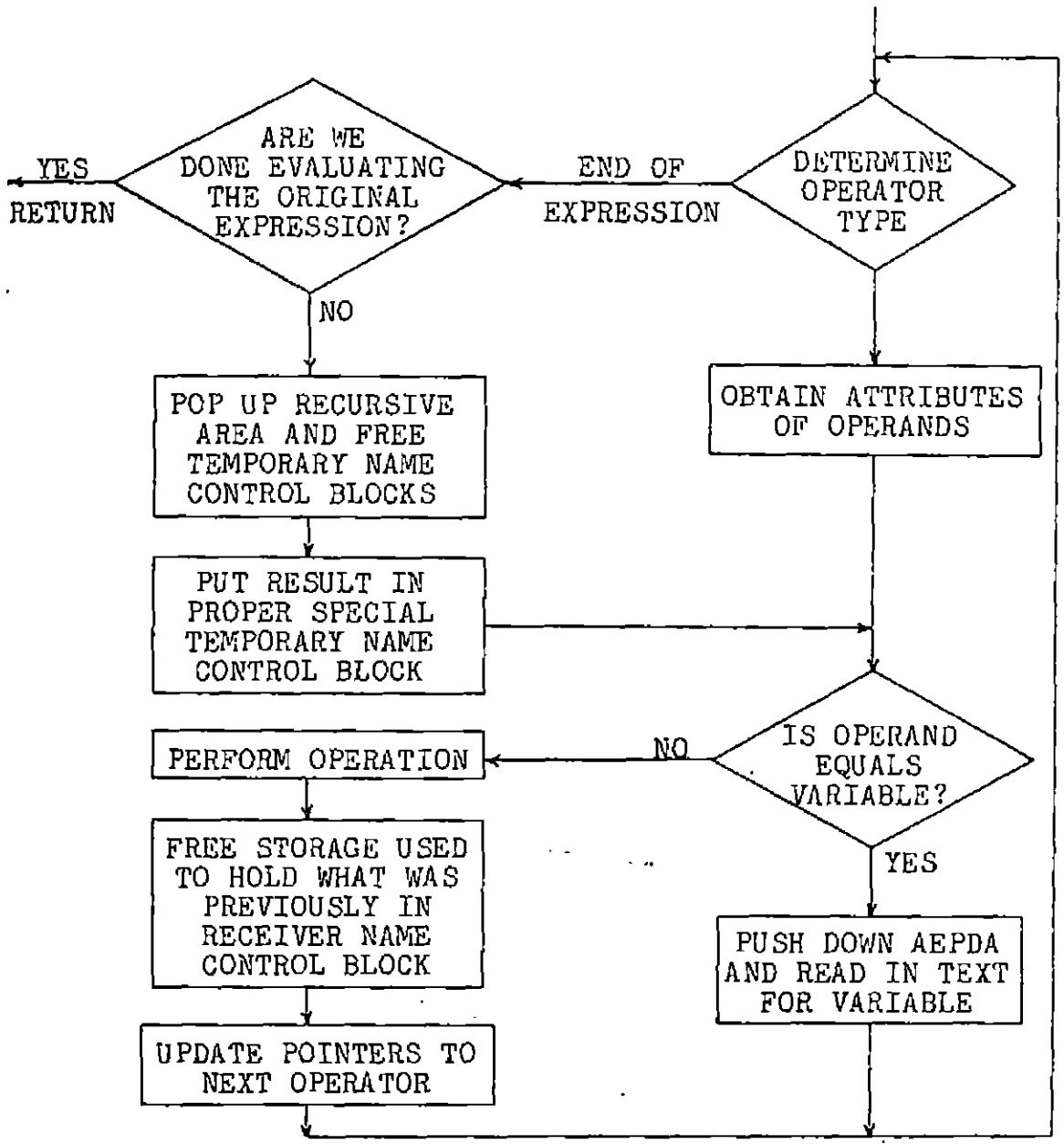


Figure 20. Flow of Control in Arithmetic Expression Evaluator when Operand is an Equals Variable

When recursion occurs the text for the current arithmetic expression is written out onto a sequential file along with a group of variables that must be saved for the interpreter and all the temporary name control blocks except for the temporary name control block used to hold the result of arithmetic expression evaluations. All of these variables are equivalenced to one contiguous area, AEPDA, so that they may be manipulated as a unit, Figure 17. A flag is set in the interpreter's recursive variable area just before the push down of storage is performed. This flag is used to return to the point in the master controller where recursion occurred after the symbolic variable's expression has been evaluated.

Because of the manner in which storage associated with temporary variables is freed, all temporary variables are set to undefined after the push down area has been written out. This allows them to be re-used during the evaluation of the new expression without the danger of freeing storage which was associated with the temporary variables at the previous level. After the new expression is read into the area used to hold text to be evaluated and the necessary pointers are adjusted, control is transferred to the main entry point of the master controller to begin execution. This new expression may also contain symbolic variables; if so, the process is repeated.

The compiler does not check for symbolic definitions which yield non-terminating definitions. This is the responsibility of the arithmetic expression evaluator during execution. The statement A = A+B and the statements A = B+C, B = A+D both give this situation. The interpreter could check for the occurrence of this when the assignment statements are made or could keep a list of what variables have caused recursion and check this before each recursion to eliminate the possibility of infinite recursion. However, neither of these methods are used in NAPSS because both require extensive checking be done for every symbolic assignment or every recursion and for the vast majority of cases this is unnecessary. Instead, a limit has been placed on the depth of recursion. If the arithmetic expression evaluator attempts to recurse past this limit, an error message is given the user indicating that the depth of recursion is greater than can be handled by the system. It is also suggested that the definition of the symbolic variable which caused the initial recursion is inconsistent.

The result of an expression associated with a symbolic variable is put in the temporary name control block that is used to receive the results of all arithmetic expression evaluations. This name control block is fixed in the compiler and the interpreter and is the only temporary name control block which is not in the push down area.

Before the push down area can be restored and execution of the original expression resumed, a pass must be made through the other temporary name control blocks to free any storage that is associated with them. If this were not done, this storage would be lost to the system since garbage collection is not used to retrieve unclaimed storage.

All temporary name control blocks need not be checked during the freeing process because the compiler assigns the temporary variables in a linear fashion and re-uses them as soon as their results are no longer needed. Thus the interpreter need only scan them until the first name control block is encountered which is still marked as undefined.

After all the temporary variables are freed, the push down area is restored and the name control block containing the result of the equals variable expression is copied into a special temporary name control block which is used only for the values of symbolic variables. The special temporary name control block is in the recursive variable area, AEPDA. This is done to permit both operands of an operator to be symbolic. The special name control block is used after evaluation in place of the symbolic variable in the evaluation of the original arithmetic expression.

To avoid needless recursions to evaluate the same symbolic variable, a local check is made to determine if any of the other operands of the current operator are the same variable. If any of them are, the special name control block is substituted in the arithmetic expression for them also.

There is a problem associated with the use of the work pool and recursion. If there are any arrays in the work pool when a symbolic variable is encountered, the work pool must be emptied. This saves the temporary result array which resides only in the work pool in the random array file. Were this not done and the symbolic expression to be evaluated involved any array arithmetic, this result array would become associated with a temporary name control block on the wrong level. Therefore, just before recursion takes place the work pool is emptied and the result array is written out into the array file and associated with the proper temporary name control block.

If an error occurs while evaluating the expression for a symbolic variable, the storage associated with the temporary variable name control blocks on the different levels must be freed. This is not necessary if the arithmetic expression evaluator is at level zero when the error occurs because in this case the normal freeing mechanism frees the storage associated with temporary variables the next time the arithmetic expression evalua-

tion is called. However, when an error is detected at a non-zero level, the storage associated with all temporary variables is freed a level at a time until level zero is reached. Information about what caused the error and at what level it occurred is saved before the recursion levels are rolled back so that an error message can be given the user.

Evaluation of Arithmetic Expressions Involving Symbolic Functions

During compilation of the text of a symbolic function, references to the first N temporary name control blocks are substituted for appearances of the N formal parameters of the function. When a function is to be evaluated the actual parameters are substituted for the formal parameters by copying the name control blocks for the actual parameters into the first N temporary name control blocks.

This cannot be done directly for two reasons: first, the function evaluation may appear at any point in an arithmetic expression and therefore some temporary values may already reside in the first N temporary name control blocks; second, one or more of the actual parameters may be arithmetic expressions which have been evaluated and had their results put in some of the temporary name control blocks.

These problems cause the arithmetic expression evaluator to recurse before the actual argument name

control blocks are copied into the temporary name control block and force the use of a temporary area to collect the parameter name control blocks. See Figure 21.

The appearance of an equals variable as an actual parameter is not handled in the same fashion as other types of parameters. Its name control block is not directly copied onto the corresponding temporary name control block. If it were, this would cause the arithmetic expression evaluator to recurse each time this parameter appears in the text for the function. Since the value of the equals variable cannot change during the evaluation of the function, this is avoided by having the arithmetic expression evaluator recurse and evaluate the equal variable before its name control block is copied into the corresponding temporary name control block. Thus, the name control block for the result of evaluating the equals variable is used in place of the name control block of the equal variable itself.

After all the name control blocks of the arguments are in the temporary area used to collect them, they are copied onto the first N temporary name control blocks.

Before evaluation commences the function is checked to see if it is a left arrow or equals function. If it is a left arrow function, then all non-parameter variables appearing in the function text had their values fixed when the function assignment was made. To fix the value of

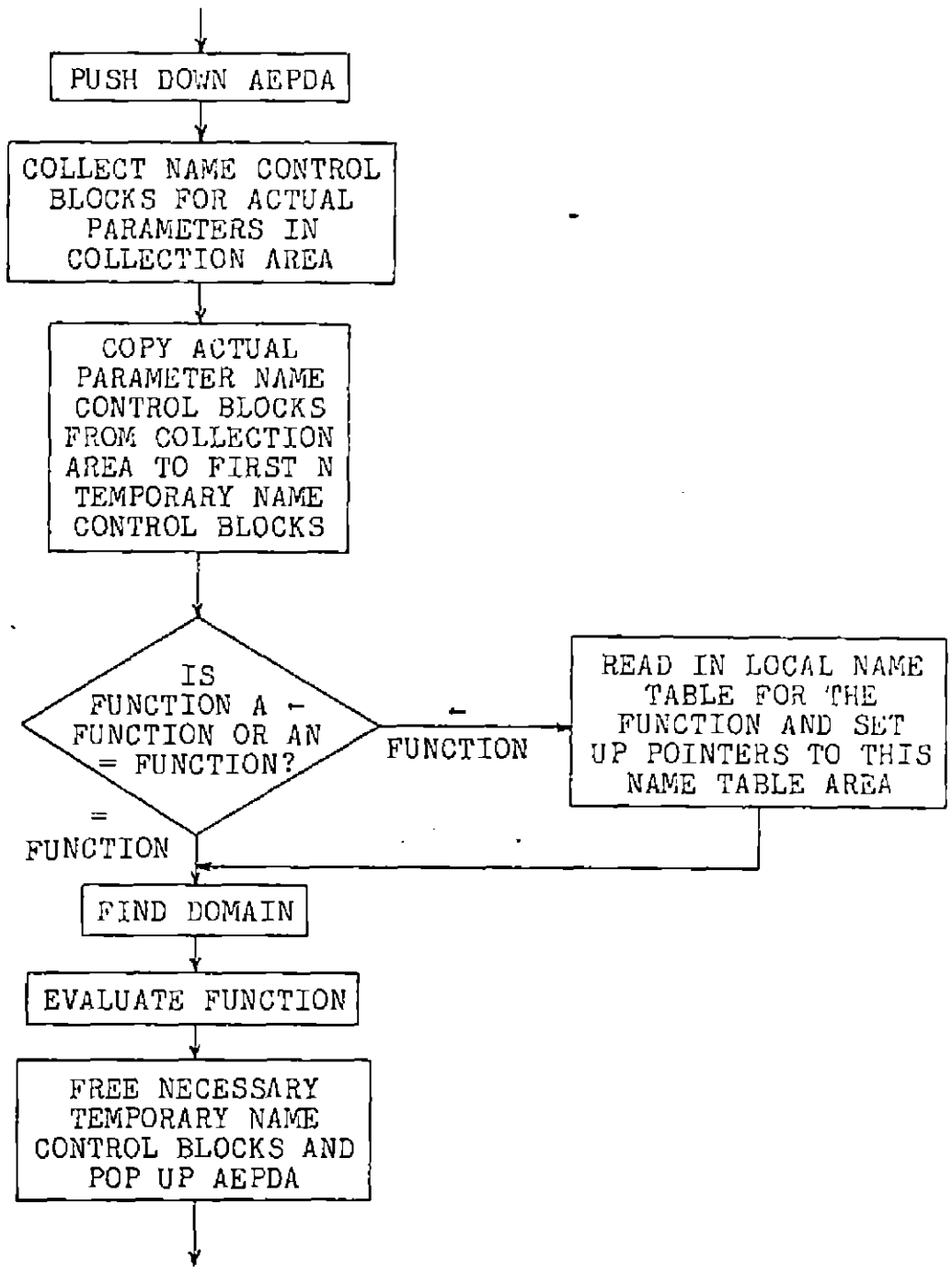


Figure 21. Symbolic Function Evaluation Flow

these variables a copy of each of their name control blocks and associated storage was created when the function assignment was performed. Thus, to evaluate a left arrow function these local name control blocks are brought into the name table area and pointers are adjusted so that these variables are referenced while the function is being evaluated.

If the function to be evaluated is an equals function, all non-parameter variables appearing in the function text are not fixed when the function assignment is made, but assume their current value when the function is evaluated. Thus, no local name control blocks are associated with equals functions.

The point at which the function is to be evaluated is checked to see if the function is defined at this point. The check is performed by evaluating the boolean expressions associated with the various definitions of the function. The boolean expressions are evaluated in the order the user has stated them. When no boolean expression appears with a definition the function is assumed to have this definition everywhere or everywhere else depending on whether or not other definitions with associated boolean expressions precede it.

After the result of a function evaluation is put in the temporary name control block which receives the result of all arithmetic expression evaluations, the arithmetic expression evaluator returns to the level at which the

function invocation occurred.

The process of returning to the level in the arithmetic expression evaluator at which the function invocation occurred is similar to what occurs when returning from the evaluation of an equals variable. The only difference is the freeing of the temporary name control blocks before the recursion area is restored. All of the temporary name control blocks may not be freed as they were after the evaluation of an equals variable, because to evaluate a function the first N temporary name control blocks were used to hold copies of the parameter name control blocks.

The copy of the actual name control block for the parameter is flagged when it is put into the corresponding temporary name control block so that when the temporary name control blocks are freed the ones used to hold parameters will not be freed. There is one temporary name control block used to hold a type of parameter which is not flagged and must have its associated storage freed. This is the temporary name control block used to hold the value of a parameter which corresponds to an equals variable. Since the equals variable is evaluated before the evaluation of the function, the only name control block pointing to the value of the equals variable is the temporary name control block used as parameter.

If an error occurs during the evaluation of a function, the arithmetic expression evaluator saves information as to what caused the error and at which level it occurred and returns to level zero as it does when an error occurs during the evaluation of an equals variable.

Evaluation of Arithmetic Expressions with Polyalgorithm Calls

A polyalgorithm is formed by grouping several numerical procedures and a supervisor into a single procedure for solving a specific problem. The polyalgorithm combines the various methods along with the strategy for their selection and use into a single method which is relatively efficient and very reliable.

The appearance of either an integral or a derivative in an arithmetic expression causes the arithmetic expression evaluator to invoke a polyalgorithm to perform the operation. Although the polyalgorithm contains its own supervisor, it requires the arithmetic expression evaluator to evaluate the function involved. Therefore, the process of evaluating an integral or derivative of a function is recursive. It is also considerably more complicated than evaluation of an equals variable or a function. In the latter two cases only the master controller of the arithmetic expression evaluator itself is involved; here the arithmetic expression evaluator and a polyalgorithm are involved. In addition, since the

polyalgorithm may require that the value of the function involved be computed repeatedly, the normal process of function evaluation which is itself recursive cannot be used in this case for practical reasons.

When a derivative or integral appears in an arithmetic expression being evaluated, all the arguments required by the polyalgorithm, such as number of derivatives, integral bounds, or point of differentiation, are evaluated in the arithmetic expression evaluator before the polyalgorithm is invoked. The values of these parameters are passed to the polyalgorithm initially so that the arithmetic expression need only be re-entered from the polyalgorithm when necessary.

Before the polyalgorithm is called the arithmetic expression evaluator recurses as it does when evaluating a function. The text of the function involved in the operation is placed in the appropriate place in the interpreter for evaluation. All parameters necessary for evaluation are also set up except for filling in the temporary name control block which corresponds to the variable of differentiation or integration. Thus, when the polyalgorithm needs to evaluate the function all that remains to be done is supply the value of this point.

When the polyalgorithm is called from the arithmetic expression evaluator and a value of the function involved is needed the arithmetic expression evaluator

must be returned to, or must be called from, the polyalgorithm. If the polyalgorithm calls the arithmetic expression evaluator, the address where the arithmetic expression evaluator was initially called from would be destroyed. If the polyalgorithm returns to the arithmetic expression evaluator, this would create problems in the organization of the polyalgorithm. For if the point at which the function must be evaluated is several routines removed from the original call on the polyalgorithm, all of these calls would have to be retraced for each evaluation of the function, or the polyalgorithm would have to be re-organized.

To avoid both of these problems direct transfers are used to transfer control between the arithmetic expression evaluator and the polyalgorithm after the polyalgorithm is initially entered. This method of transferring between routines is accomplished by the use of assigned go to statements in each of the routines.

When the polyalgorithm completes its work it returns to the arithmetic expression evaluator normally. The arithmetic expression evaluator then restores itself to the level at which the integral or derivative occurred. The process of freeing storage associated with temporary name control blocks and the popping up of the recursive area is similar to what is done after the evaluation of a function.

If an error occurs which causes the polyalgorithm to terminate evaluation, it returns to the arithmetic expression evaluator as if the evaluation was successful but with an error flag set. The arithmetic expression evaluator returns to level zero as is done when an error occurs during an equals variable or a function. The actual message is issued by the routine which initially called the arithmetic expression evaluator.

ASSIGNMENT STATEMENTS

General Problems

The variable appearing on the left in an assignment statement may have none, some, or all of its attributes explicitly declared. If no attributes are explicitly declared for a variable, it may appear on the left in any assignment statement and it assumes all of its attributes contextually from the type of assignment statement and the attributes of the expression on the right. If some or all of its attributes are explicitly declared, then the type of assignment statement in which it appears and the attribute of the expression on the right must agree with the variable's declared attributes. If they do not agree, the assignment is not performed, an error message is issued, and the variable's value remains unchanged. When a new value is assigned to a variable the storage occupied by the previous value is normally returned to the system.

The previous value may be pushed down to the next name control block in an iteration chain when a new value is assigned to the head of the chain. An iteration chain is pushed down only when the structure of the new value assigned to the head is compatible with that of the previous value. Thus, if the previous value assigned to

the head of an iteration chain is a real scalar and the new value is a complex scalar, the chain is pushed down. However, if the previous value is a real array and the new value is a real scalar, the iteration chain is freed. Previous values are saved only for numeric scalars and arrays.

The pushing down of an iteration chain is accomplished by copying each name control block onto the one following it, except for the iteration pointer field. If the last name control block in the chain is defined, the storage associated with its previous value is returned to the system.

When a new value is assigned to the head of an iteration chain and the previous value is not to be pushed down, each name control block in the chain is set as being undefined and the storage associated with them is returned to the system. When this occurs, the iteration chain is referred to as being freed.

The work pool must be emptied by each assignment statement that calls the arithmetic expression evaluator. If the result of the expression on the right is an array, this stores it out in the array disk file, since it currently resides only in the work pool.

A description of the storage associated with the various variable data types is given in Appendix A.

Left Arrow Assignment Statement: A←

The left arrow assignment statement assigns the value of the expression on the left to the variable on the right. The value assigned may be a numeric scalar, a numeric array, or a scalar string.

When the result of the expression is numeric, it cannot be complex if the variable on the left is declared explicitly to be double precision. However, if the result is complex and the variable on the left is declared to be real, single or double precision, conversion is made when the result is scalar and its imaginary part is zero. No test is made to determine if the imaginary part is zero when the result is a complex array. Other conversions for both arrays and scalars are made automatically.

When the result is an array and the variable on the left is the head of an iteration chain, the fact that the previous value is also an array does not guarantee that the iteration chain is pushed down. When arrays are involved, an iteration chain is pushed down only when the number of indices and the sizes of the new and the previous array values match. The actual bounds of each index are not checked; instead, the number of elements in each dimension are compared. If either the number of indices or the number of elements in each dimension are not identical, the iteration chain is freed.

The variable on the left assumes the bounds of the result array when it does not have any bounds explicitly declared. When bounds are explicitly declared for the variable the number of dimensions must match on both sides and the number of elements assigned in each dimension must not exceed the number declared. If both of the above conditions are met, the lower bounds of the result array are set to the declared lower bounds of the variable and the upper bounds of the result are adjusted accordingly.

Array Element Left Arrow Assignment Statement: A[I,J]←

This assignment statement assigns the value of the arithmetic expression on the right to the element or elements of the subscripted variable on the left. The expression on the right can yield either a numeric scalar or a numeric vector.

The subscripts may be a single arithmetic expression, two arithmetic expressions, or an arithmetic expression and a *. Subscript expressions must yield numeric scalars. If the scalar is not real, a warning message is issued and only the real portion of the number is used. Real subscript values are converted to an integer by rounding.

When all of the subscripts of the variable on the left are arithmetic expressions, the expression on the right must yield a scalar. And, when one of the subscripts is a *, the expression on the right must yield either a row or column vector. Thus, for example, a row vector may

be assigned to the column of a matrix.

There are two sets of bounds associated with each array: the actual bounds and the optional declared bounds. The actual bounds are the current bounds of an array and they must always remain inside the declared bounds when such are given.

The number of cases that are handled by this assignment is large, due to the fact that the user may declare none, some, or all of the attributes for the variable on the left. If the variable on the left has not previously been assigned a value, four possible cases exist.

Case one: the variable on the left is declared to be an array and the number of subscripts is fixed. The bounds for the subscripts may also be declared.

The actual bounds associated with a * subscript are the actual bounds of the vector result when no bounds are declared for the index of the variable. But if the * index has its bounds declared, then the lower bound of the result vector is adjusted to match the declared lower bound of the variable and the actual upper bound is modified accordingly. If after the adjustment is made the actual upper bound of the vector exceeds the declared upper bound of the variable, no assignment is made and an error results.

Case two: the variable on the left is declared to be an array, but no declaration has been made concerning the

number of subscripts or their bounds. Then the variable on the left may appear with either one or two subscripts, and it is contextually defined to have the corresponding number of dimensions. The actual bounds of the array are established from the subscript expressions or from the vector result for a * index.

Case three: the variable on the left is not declared explicitly to be an array, but can have an array as its value. Then the variable is contextually declared to be an array and the assignment proceeds as in case two.

Case four: the variable on the left is declared to be a scalar. In this case, no assignment is made and an error results.

If the variable on the left has previously been assigned a value, there are six possible cases.

Case one: the variable on the left presently is an array, the number of subscripts match, and the values of the subscripts lie within the present actual bounds, or in the case of a * subscript the number of elements in the vector result equals the number of elements in the * index. Thus only a replacement of previous values is necessary.

Conversion is performed when possible to insure that the mode and precision of the current and new values agree. If no conversion can be performed, as is the case when the variable on the left is declared to be real and the result

of the expression on the right is complex, an error is issued and no assignment is made. If the variable on the left is the head of an iteration chain, the chain is pushed down in this case.

Case two: the variable on the left currently is an array but the dimensions do not match on the left and right. If the dimension is explicitly declared for the variable on the left, no assignment is made and an error results. If the dimension is not declared, the previous array value is destroyed and a new array is created.

Case three: the variables on the left and right have the same number of dimensions, but the one on the left must be expanded. This occurs when either the value of a subscript falls outside the actual bounds of the current array value (but inside the declared bounds), or the number of elements in a vector to be assigned exceeds the number of elements currently in the * dimension.

An array is expanded by adding zero columns and rows to the previous value. Then the assignment is made to the expanded array.

When the expansion results from a vector being assigned to a row or column, the actual upper bound is adjusted first. If the declared upper stops the expansion before enough space is obtained, then the actual lower bound is adjusted to obtain the remaining space needed.

When an array is expanded and the variable on the left is the head of an iteration chain, the chain is freed.

Case four: the variable on the left currently is an array and the number of elements to be assigned to a * index is less than the number of elements currently in that dimension. Because it is not known which elements are to be replaced in this case, no assignment is made and an error results.

Case five: the variable on the left currently does not have an array as its value, but it may. The assignment of the new value is made as in case three, when the variable had no previous value.

Case six: the variable on the left is declared to be scalar. Because attributes are assigned at execution time, this error cannot be detected by the compiler but is detected when the assignment is attempted and an error results.

Equals Assignment Statement: A =

This assignment statement assigns the text, not the value, of the expression on the right of the = to the variable on the left.

Before the assignment is performed, the internal text for the expression on the right is scanned to check for scope conflicts between the variables on the right and left. Also during this scan the relative pointers to user-created variables are replaced by absolute pointers.

This process is simplified by the fact that the pointers to user variables are represented in the internal text by positive integers, while all other quantities in the internal text are represented by negative integers.

If the variable on the left is global all variables appearing in the expression on the right must also be global. If the variable on the left is non-local all the variables in the expression must be known in the outer-most procedure in which the variable on the left is known.

The pointers generated by the compiler to user variables are relative to the beginning of the name table for the procedure being compiled. Thus for the variable on the left to be assigned a value in an internal procedure and have its value computed in another procedure or the console level, the relative pointers to user variables must be replaced by absolute pointers. The relative pointers to variable name control blocks are replaced by absolute pointers to the name control blocks which contain the actual values for the variables. The absolute pointers are into the array AENCBS. See figure 17. This replacement insures that wherever the value of the variable on the left is computed the correct name control block entries are used.

For global variables the absolute pointer is to the name control block for the variable in the global table, and for non-local variables the absolute pointer is to

the name control block for the variable in the outermost name table in which the variable is known; see figure 17.

Equals Function Assignment Statement: $F(X_1, X_2, ---, X_N) =$

This assignment statement defines the variable on the left to be a symbolic function (an equals function). The variables appearing in the text of the function do not have their current value fixed when the assignment is made; rather they assume their current value when the function is evaluated.

A symbolic function may be defined to have from one to four formal parameters. The maximum of four formal parameters for a function is a system parameter and may be changed at any time.

Before the assignment is made, the scope of the variables appearing in the text is checked, and all relative pointers are replaced with absolute pointers, as is done in the equals assignment statement: $\Lambda =$.

The formal parameters are not treated in the same manner as other variables appearing in the text of the function. References to the N formal parameters are replaced during compilation with references to the first N temporary name control blocks. Therefore, they do not appear as normal variable references.

Left Arrow Function Assignment Statement: $F(X_1,X_2,\ldots,X_n)$←

This assignment statement also defines the variable on the left to be a symbolic function (a left arrow function). The difference between this assignment and the previous one is that here all variables on the right have their values fixed when the assignment is made. Thus there are no scope conflicts to be checked, but all variables appearing in the function must be defined.

Values are fixed by creating a local name control block for each variable on the right and associating a copy of the current value of the variable with this name control block. All references to these variables are modified to point to these local name control blocks.

A function definition is broken up by domains and a local name table is created for the variables occurring in each domain.

For each left arrow variable appearing in a function definition a copy of the variables name control block is created and placed in the local name table for the function. If the value of the variable is an array, its value is not copied. The name control block points to the array and the reference count associated with the array is increased by one.

When an equals variable appears in the text of the function, the expression associated with the variable is evaluated and a name control block is created in the

local name table with the same name and the value of the evaluation is associated with it.

The occurrence of a left arrow function in a function's definition causes a copy of the function's name control block to be placed in the local name table for the function being defined. Instead of making a copy of the definition of the left arrow, the reference count associated with the function definition is increased by one.

If none of the arguments of a left arrow function appearing on the right involve any of the formal parameters of the function being defined, then the function may be evaluated. This is done and the resulting value is associated with a special name control block which is placed in the local name table of the function being defined. This results in a time saving whenever the function being defined is evaluated.

When an equals function appears in the definition of a left arrow function it, too, is evaluated if none of its arguments involve any of the formal parameters of the function being defined. When this is not the case, the equals function is converted to a left arrow function. This causes the routine which performs the left arrow function assignment to recurse, and results in two copies of the function definition to be saved. One definition, the original, is associated with the equals function name control block and the other is associated with a name

control block in the local name table of the function being defined.

The mechanism used to save temporary variables when this routine recurses is the same as used by the arithmetic expression evaluator.

If a variable appears in the definition of a left arrow function which is undefined, non-numeric, or itself not a function, then the assignment is not made and an error results. When this occurs, because garbage collection is not used all storage associated with local name control blocks and the definition of the function up to the point of the error must be returned to the system.

Array of Functions Assignment Statements:
$F(X_1, X_2, \ldots, X_N)[I,J]\leftarrow$

These assignment statements define an element in an array of symbolic functions. We use all the rules involved when assigning an element in a numeric array, or defining a scalar left arrow or equals function. In addition, we require that only arithmetic expressions appear as subscripts. The elements of the array must either all be left arrow functions or all equals functions.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Culler, G. J. (1968), "Mathematical Laboratories: A New Power for the Physical Sciences," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 355-384.

2. Engleman, C., "MATLAB - A Program for On-Line Machine Assistance in Symbolic Computation," Proceedings Fall Joint Computer Conference 1965, pp. 423-

3. Falkoff, A. D. and Iverson, K. E. (1968), "The APL 360 Terminal System," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 22-37.

4. Hearon, A. C. (1968), "REDUCE: A User-oriented Interactive System for Algebraic Simplification," Interactive Systems for Experimental Applied Mathematics,(Klerer, M. and Reinfelds, J., eds.), pp. 79-90.

5. Hill, P. B. and Stowe, A. N. (1968), "Implementation of a Reckoner Facility on the Lincoln Laboratory IBM 360/67," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 385-389.

6. Kaplow, R., Brackett, J., and Strong, S. (1966), "Man-Machine Communications in On-Line Mathematical Analysis," Proceedings - Fall Joint Computer Conference, pp. 465-477.

7. Klerer, M. and May, J. (1964), "An Experiment in a User-Oriented Computer System," Communications ACM, 7, No. 5, pp. 290-294.

8. Klerer, M. and May, J. (1965), "A User Oriented Programming Language," Computer Journal, 8, No. 2, pp. 103-109.

9. Klerer, M. and May, J. (1967), "Automatic Dimensioning," Communications ACM, 10, No. 3, pp. 165-166.

10. Klerer, M., Grossman, F., and Amann, C. H. (1968), "Design Philosophy for an Interactive Keyboard Terminal," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 183-191.

11. Lock, K. (1968), "An Object Code for Interactive Applied Mathematical Programming," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 222-224.

12. Matthews, H. F. (1968), "VENUS: A Small Interactive Non-procedural Language," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 97-101.

13. Reinfelds, J. (1968), "An Implementation of Automatic Array Arithmetic by a Generalized Push-Down Stack," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 411-422.

14. Roos, D., "An Integrated Computer System for Engineering Problem Solving," Proceedings Fall Joint Computer Conference 1964.

15. Roos, D. (1967), ICES System Design, MIT Press.

16. Rice, J. R. and Rosen, S., "NAPSS: A Numerical Analysis Problem Solving System," Proceedings - ACM National Meeting 1966, pp. 51-56.

17. Rice, J. R. (1968), "On the Construction of Polyalgorithms for Automatic Numerical Analysis," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 301-313.

18. Rice, J. R. (1969), "A Polyalgorithm for the Automatic Solution of Non-linear Equations," Purdue University Technical Report, CSD TR 32.

19. Roman, R. V. and Symes, L. R. (1968), "Implementation Considerations in a Numerical Analysis Problem Solving System," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 400-410.

20. Ruyle, A., Brackett, J. W., and Kaplow, R., "The Status of Systems for On-Line Mathematical Assistance," Proceedings - ACM National Meeting 1967, pp. 151-168.

21. Schlesinger, S. and Sashkin, L. (1967), "POSE: A Language for Posing Problems to the Computer," Communications ACM, 10, No. 5.

22. Schlesinger, S. I., Sashkin, L., Reed, K. C. (1968), "Two Analyst-Oriented Computer Languages: EASL, POSE," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 91-96.

23. Seitz, R. N., Wood, L. H., and Ely, C. A. (1968), "AMTRAN: Automatic Mathematical Translation," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 44-66.

24. Shaw, J. C., "JOSS: A Designer's View of an Experimental On-Line Computing System," Proceedings Fall Joint Computer Conference 1964, pp. 455-464.

25. Stowe, A. N., Weisen, R. A., Yntema, D. B., and Forgie, J. W., "The Lincoln Reckoner: An Operation-Oriented On-Line Facility with Distributed Control," Proceedings - Fall Joint Computer Conference 1966, pp. 433-444.

26. Symes, L. R. and Roman, R. V. (1967), "NAPSS Primer," Purdue University Technical Report, CSD TR 11.

27. Symes, L. R. and Roman, R. V. (1967), "Syntactic and Semantic Description of the Numerical Analysis Programming Language (NAPSS)," Purdue University Technical Report, CSD TR 11.

28. Symes, L. R. and Roman, R. V. (1968), "Structure of a Language for a Numerical Analysis Problem Solving System," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 67-78.

29. Wiesen, R. A., Yntema, D. B., Forgie, J. W., and Stowe, A. N. (1968), "Coherent Programming in the Lincoln Reckoner," Interactive Systems for Experimental Applied Mathematics, (Klerer, M. and Reinfelds, J., eds.), pp. 167-177.

30. Wood, L. H., Reinfelds, J., Seitz, R. N., and Clem, P. L. (1966), "The AMTRAN System," Datamation, 12, No. 10.

APPENDICES

APPENDIX A

DATA STRUCTURES

Equals Variable

The attribute number for an equals variable is zero. The data pointer field of the variable's name control block contains the page number of the first virtual page used to store the text for the expression. The data portion of the name control block is unused.

The text is packed in each virtual page, three twenty-bit integers per word. The first word of each virtual page is used for linking. The link contains the virtual page number of the next page used to hold the text of the expression or zero if the page is the last.

Figure A1 displays the data structure for an equals variable. Two pages of virtual memory are used to hold the text for the expression.

Left Arrow Variables

Real Single Precision Scalar

The attribute number 1 specifies a variable whose value is a real single precision scalar. The value is stored in the first word of the data portion of the name control block. The remaining three words of the data portion are unused, see Figure A2.

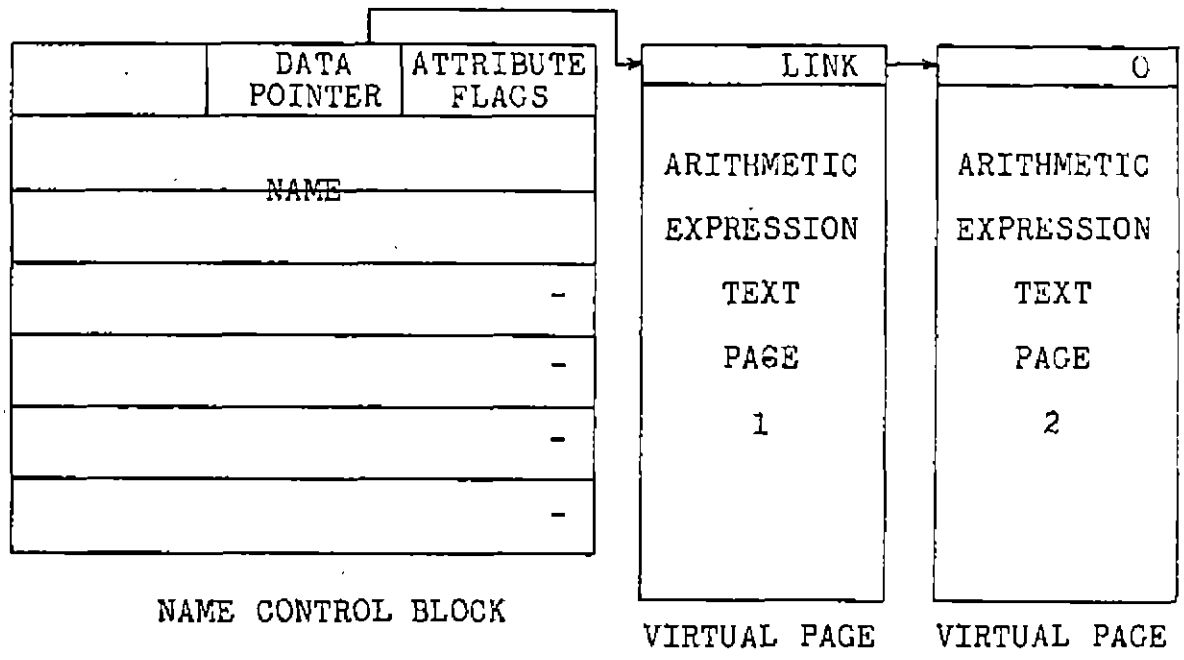


Figure A1. Equals Variable Data Structure

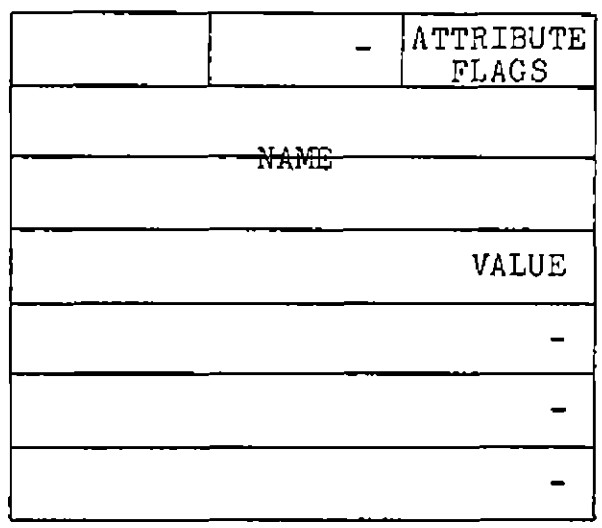


Figure A2. Real Single Precision Scalar Data Structure

Real Double Precision Scalar

The attribute number 2 specifies a variable whose value is a real double precision scalar. The value is stored in the first and second words of the data portion of the variable's name control block. Words three and four of the data portion are unused. See Figure A3.

Complex Single Precision Scalar

The attribute number 3 specifies a variable whose value is a complex single precision scalar. The value is stored in the first and third words of the data portion of the variable's name control block. The real part is in word one and the imaginary part in word three. Words two and four of the data portion are unused. See Figure A4.

Complex Double Precision Scalar

The attribute number 4 specifies a variable whose value is a complex double precision scalar. The value is stored in the four word data portion of the variable's name control block. The real part is in words one and two and the imaginary part in words three and four. See Figure A5.

Numeric Arrays

The attribute numbers 5,6,7 and 8 denote a real single precision array, a real double precision array, a complex single precision array, and a complex double

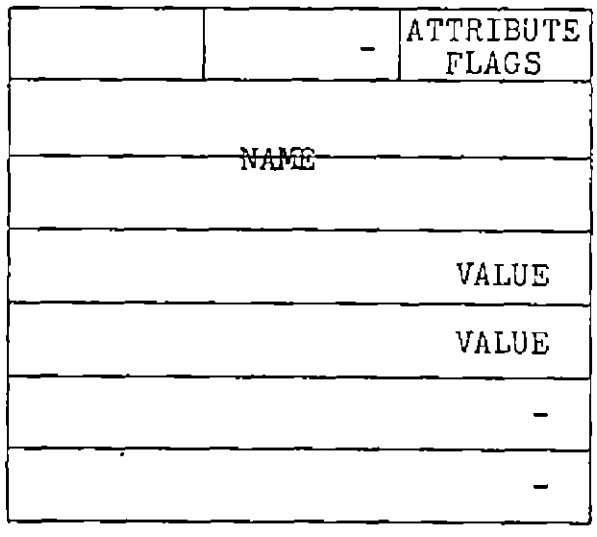


Figure A3. Real Double Precision Scalar Data Structure

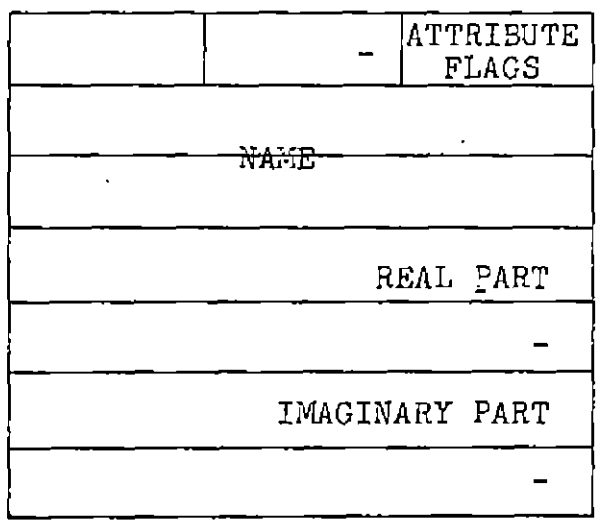


Figure A4. Complex Single Precision Scalar Data Structure

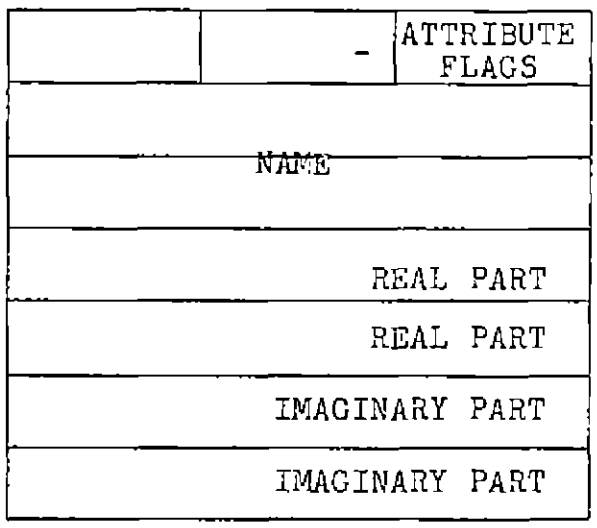


Figure A5. Complex Double Precision Scalar Data Structure

precision array, respectively.

If the data pointer field of the name control block for the array variable is non zero, a copy of the array exists in secondary storage in the array file. The data pointer is then the number of the record used to store the array and an index in the vector AEPAR.

The vector AEPAR contains additional information about the array. Each entry in AEPAR is broken into three bytes, each of twenty bits. These bytes are numbered from left to right, byte three, byte two, and byte one.

Byte three contains the reference count for the array. Byte two contains the number of dimensions in the array. And byte one contains the number of words in the array. The number of words in an array is equal to the number of elements in the array times the number of words in each element.

There is one word per element if the array is real single precision, two words per element if the array is either real double precision or complex single precision, and four words per element if the array is complex double precision. The elements are stored consecutively by rows.

If the data pointer field of the array's name control block is zero, the only copy of the array exists in the work pool, and the array is the result of the last array operation performed.

The vectors AEAWP1 and AEAWP2 contain information about the arrays in the work pool. AENAWP is the number of arrays in the work pool. Each entry in AEAWP1 contains the index in AENCBS of the first word of the data portion of the name control block of the array variable. Each entry in AEAWP2 is subdivided into three bytes. Byte three contains the index in the work pool of the first word of the array; byte two contains the number of dimensions in the array, and byte one contains the number of words in the array.

When the data pointer field of an array variable's name control block is zero, the information about where the array is in the work pool and the number of words in the array is contained in AEAWP2 (AENAWP).

The bound information for an array's indices is contained in the name control block. The three bytes of the first word of the data portion of the name control block contain the declared lower bound for index one (DLB1), the actual lower bound for index one (ALB1), and the declared upper bound for index one (DUB1). Word two contains the actual upper bound for index one (AUB1), the declared lower bound for index two (DLB2), and the actual lower bound for index two (ALB2). Word three contains the declared upper bound for index two (DUB2), the actual upper bound for index two (AUB2), and the number of dimensions for the array. Word four of the data portion is unused.

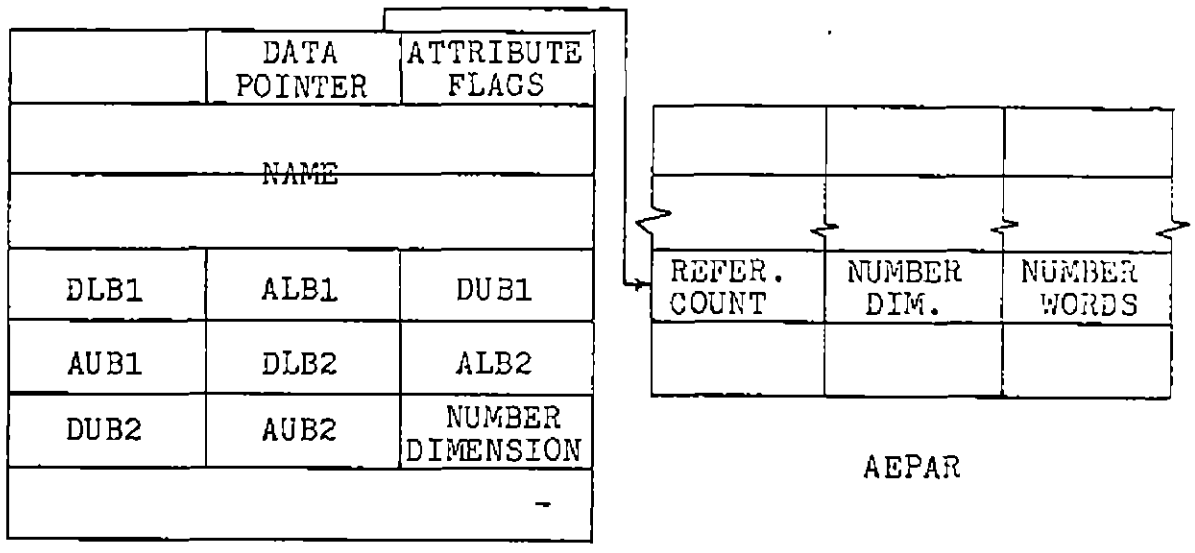


Figure A6. Data Structure for an Array in Array File

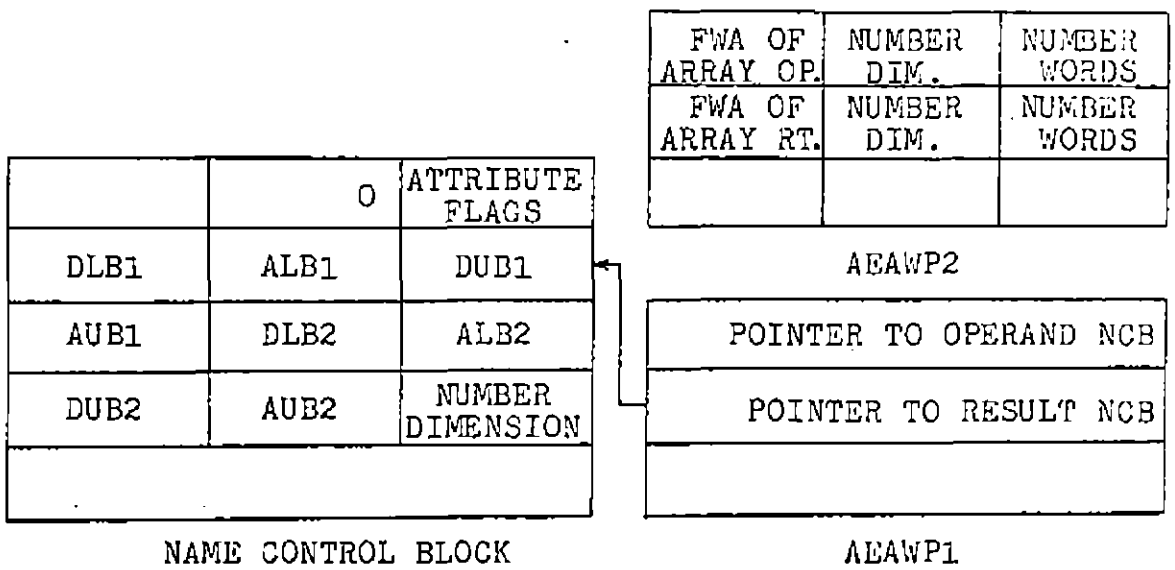


Figure A7. Data Structure for an Array in Work Pool, Only

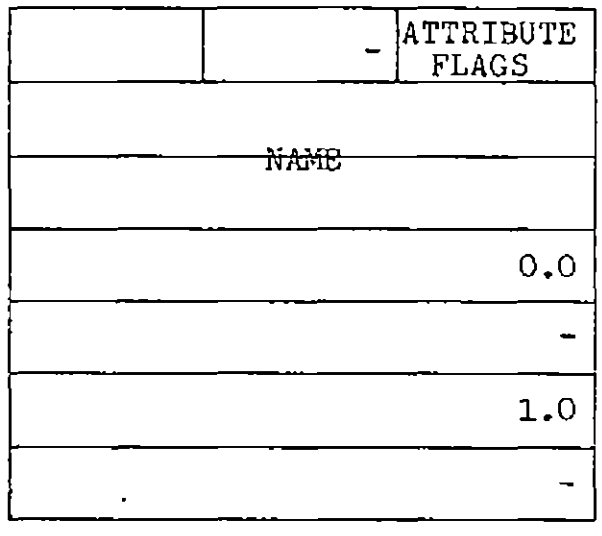
Declared bound information is optional for an array. The user may declare the number of dimensions, plus (optionally) some of the bounds for the indices. If a bound is declared for an index, the corresponding field contains the value of the declared bound. If a dimension is declared but no bound is declared for the index, the declared bound fields for the dimension contain 1777776_8. If a variable is declared to have only one dimension, the declared bound fields for the second index contain 3777777_8. If no dimension information is declared for an array variable, all the declared bound fields contain 1777777_8.

Figure A6 displays the name control block for an array variable when a copy of the array resides in the array file.

Figure A7 displays the temporary name control block of an array result which resides only in the work pool. In Figure A7 only two arrays exist in the work pool: an operand array and the result array. Thus AENAWP = 2.

Imaginary Place Marker

The attribute number of a variable declared to have the value of $\sqrt{-1}$ is 9. The first word of the data portion of the variable's name control block contains zero and the third word contains one. The second and fourth words of the data portion are unused. Figure A8 describes this data structure.



NAME CONTROL BLOCK

Figure A8. Data Structure for Imaginary Place Marker

Scalar String

The attribute number 10 specifies a variable whose value is a scalar string. The data pointer field of the variable's name control block contains the string number. The data portion of the name control block is unused.

The string number is the index of an entry in the string relocation table, AERLTB. Each entry in AERLTB contains additional information about a string. An entry is subdivided into three bytes. Byte three contains the index of the start of the actual string description in the string picture table. Byte two contains the reference count for the string. The reference count designates the

number of times the string is referenced from the string picture table plus one for the original reference from the string variable's name control block. Byte three contains the index in AENCBS of the first word of the data portion of the name control block for the string variable.

The string picture table contains a description of each string. Several entries are used to characterize a string. Each entry denotes either a literal string, a reference to a string variable, or the end of a string picture.

An entry in the string picture table, AESTRP, is subdivided into three bytes.

If byte one is not zero or 1313, then the entry describes a literal. Byte one is the number of characters in the literal, byte three is the virtual page number of the page on which the literal is stored, and byte two is the displacement on that page to where the literal begins.

Each word in a virtual memory page used to hold string literals is subdivided into three bytes. A literal is divided into segments of three characters. Each segment is stored in a byte. If a string literal will not fit in the number of bytes remaining in the current string page, the literal is broken. As many segments of the literal as possible are placed in the current string page and the remainder are placed in a new string page. When

this occurs two entries are put in the string picture table. The maximum length of a string literal is 576 characters.

If byte one is 1313, then the entry denotes the null string. It has no length and does not require any storage, so byte two and three are zero.

If byte one is zero and byte three is not 501, the entry denotes a reference to a string variable. Byte three contains the index of the entry for the string variable in the string relocation table.

If byte one is zero and byte three is 501, the entry denotes the end of a string picture.

Space in the string relocation table and the string picture table is returned to the system when the string they describe is no longer referenced. The use of pointers in the string picture table to the string relocation table saves space, because only one copy of a given string picture needs to appear in the string picture table.

Figure A9 describes the data structure for the strings created by the assignment statements B ← "YZ", A ← "ABCDEF"||B||"".

String Array

The attribute number 11 denotes a variable whose value is an array of strings. The data portion of the variable's name control block contains the same bound

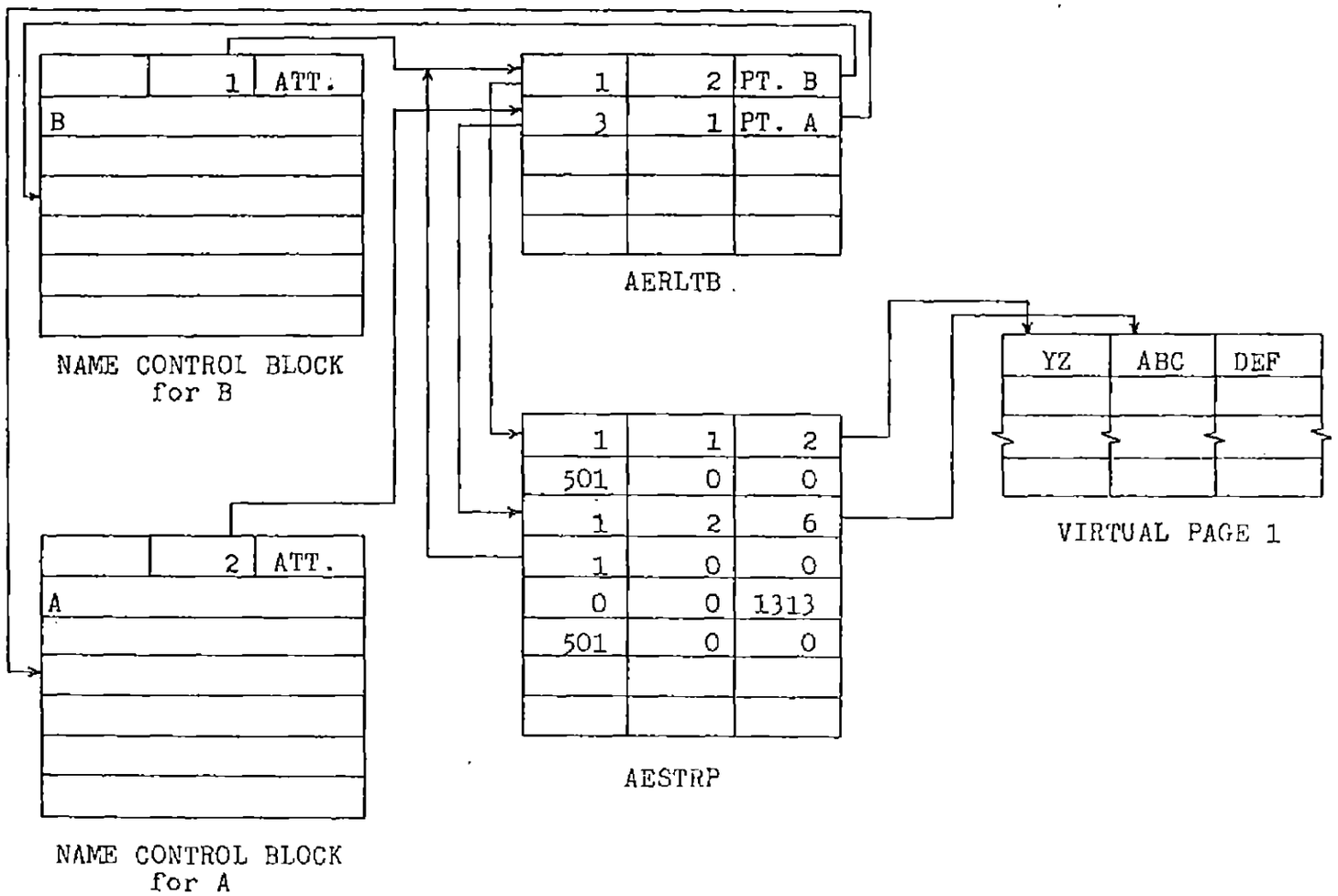


Figure A9. Data Structure for Strings

information as is contained in the name control block of a variable which denotes a numeric array. The data pointer field of the name control block contains the array number.

The array is treated as a single precision real array. The elements of the array contain the indices of the entries in the string relocation table for the string descriptions. If an element is undefined, its value is zero.

Boolean Values

The attribute numbers 12 and 13 denote logical variables whose values are true and false respectively. A user-created variable may not be assigned a logical value; however, temporary name control blocks are assigned boolean values when a relational or boolean expression is evaluated. The data portion and the data pointer field of a temporary name control block assigned a boolean value are not used.

Scalar Symbolic Left Arrow Function

The attribute number 16 denotes a variable whose value is a scalar symbolic left arrow function. The data pointer field of the variable's name control block contains the page number of the first virtual page used to store the arithmetic expression text for the first domain. The number of arguments of the function is contained in

byte three of the fourth word of the data portion of the name control block. The remainder of the data portion is unused.

The first four words of the first page used to store the arithmetic expression text for each domain contains a set of pointers. The first word is used to link together the pages required to store the text for the arithmetic expression of the domain. It contains the virtual page number of the next virtual page used. A zero link denotes the last page. Byte three of the second word contains the number of words of internal text in the boolean expression for the domain (WORDS B.E.). Byte two of the second word contains the reference count for the function definition. This byte is only used in the first domain of the function. Byte one of the second word contains the virtual page number of the first virtual page used to hold the boolean expression text for the domain (V.P.B.E.). This byte is zero if there is no boolean expression. Byte three of the third word contains the number of virtual pages that are required to hold the local name table for the domain (N.P.L.N.T). Byte two of the third word is unused, and byte one contains the page number of the first virtual page used to hold the local name table (V.P.L.N.T.). Byte three of the fourth word contains the number of words of internal text in the arithmetic expression for the

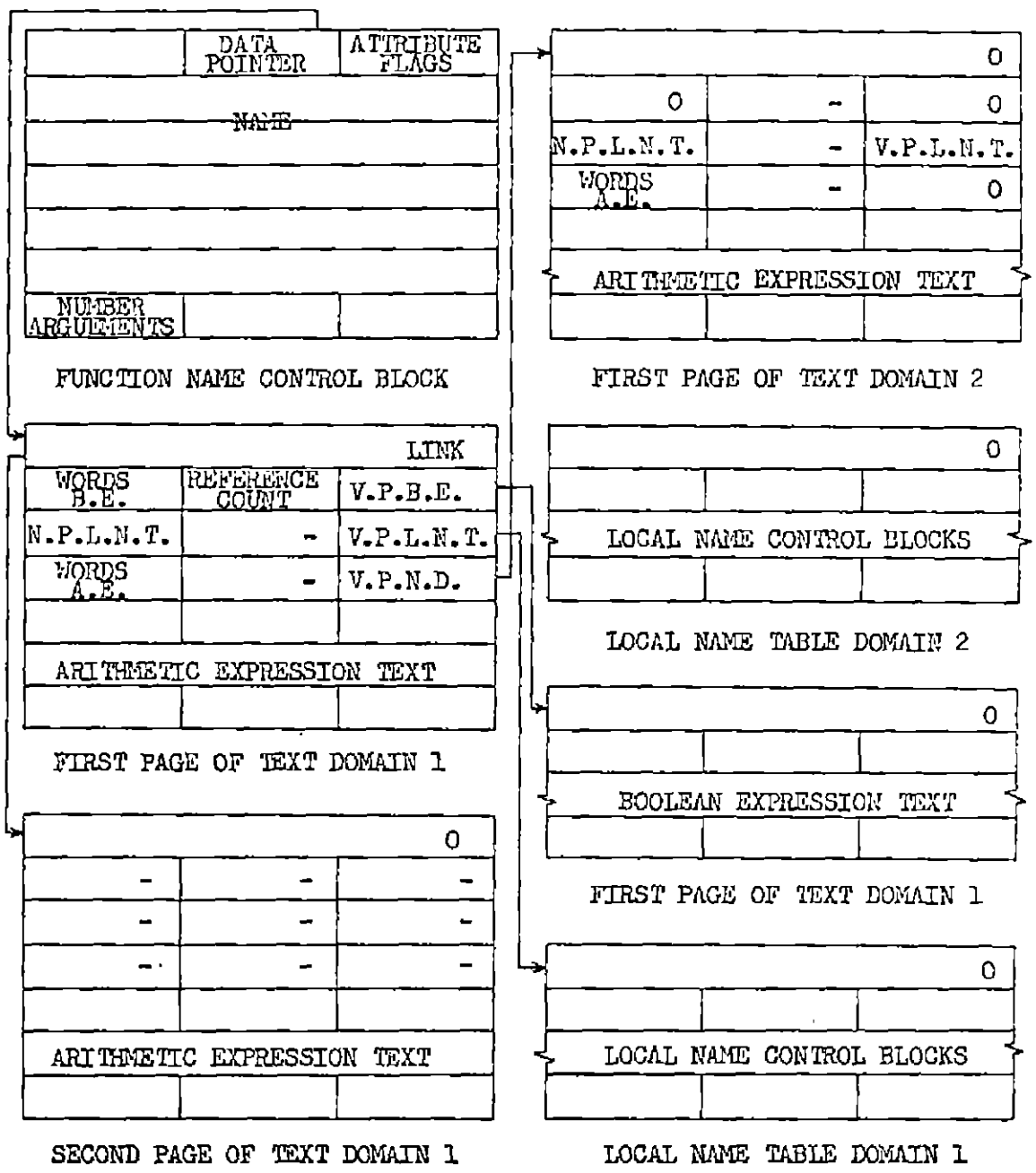


Figure A10. Data Structure of Left Arrow Function

domain (WORDS A.E.). Byte two of the fourth word is unused and byte one contains the virtual page number of the first page of arithmetic expression text for the next domain (V.P.N.D.). If this byte is zero, there is not another domain defined for the function.

The virtual pages used to store the text for a boolean expression or the local name table are linked together by the first word of each page. A zero link specifies the last page.

Figure A10 describes the data structure for a left arrow scalar function with two domains. Two virtual pages are required to hold the arithmetic expression text of the first domain, one page is required for the local name table of each domain, the boolean expression text for the first domain and the arithmetic expression text for the second domain. There is no boolean expression associated with the second domain.

Scalar Symbolic Equals Function

The attribute number 17 denotes a variable whose value is a scalar symbolic equals function. The name control block for the variable contains the same information as the name control block of a scalar symbolic left arrow function.

The first four words of the first virtual page used to store the arithmetic expression text for each domain contains a set of pointers. Word one contains a link to

additional pages used to store the text of the arithmetic expression for a domain. Word two contains the number of words of text in the boolean expression and the virtual page number of the first virtual page used to store the boolean expression text for the domain (V.P.B.E.). This word is zero if there is no boolean expression. Word three is unused since there is no local name table. Word four contains the number of words of arithmetic text and the virtual page number of the first page used to store the arithmetic expression text for the next domain (V.P.N.D.). Byte 3 is zero if there is not another domain defined.

Figure A11 displays the name control block of a scalar symbolic equals function and a portion of the first virtual page used to store the arithmetic expression text for the first domain.

Array Symbolic Left Arrow Function

The attribute number 18 denotes a variable whose value is an array of symbolic left arrow functions. The name control block of the variable contains the same bound information in the first three words of the data portion as a numeric array. The array number is in the data pointer field of the name control block and byte three of the fourth word of the data portion contains the number of arguments in each of the functions.

The array is treated as if it is an array of real

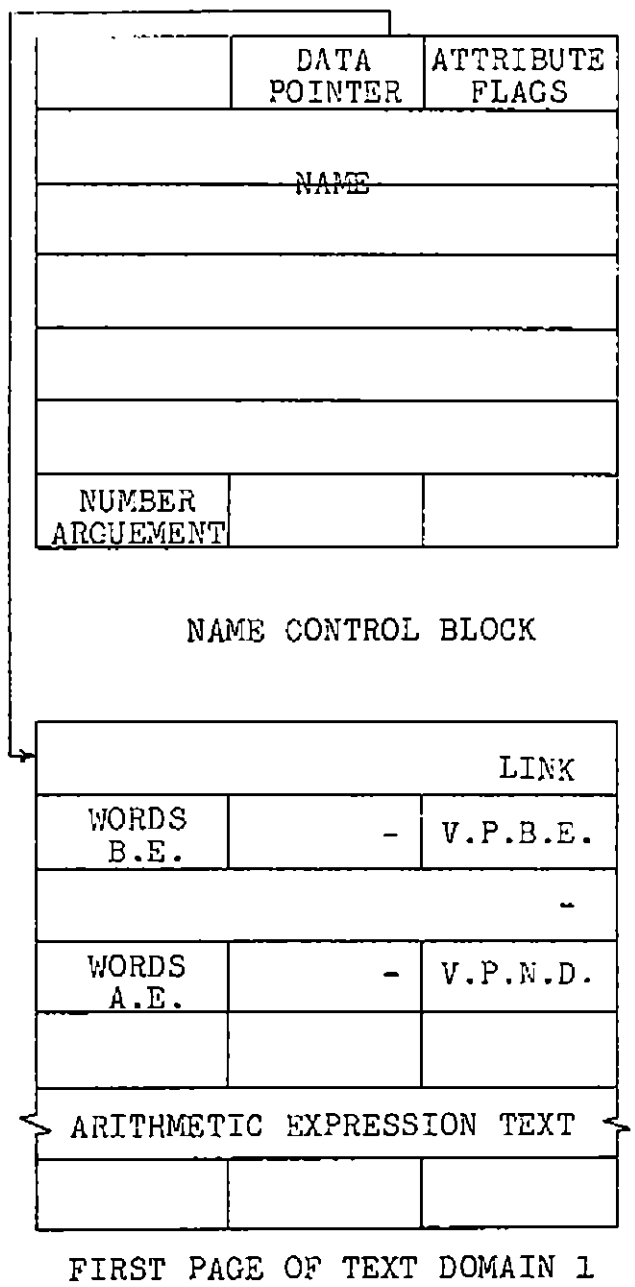


Figure A11. Data Structure of Equals Function

single precision numbers. Each element contains the virtual page number of the first virtual page used to store the arithmetic expression text for the first domain of the element's definition. If an element is not defined, its value is zero.

The text for the definition of each element is linked together in the same manner as a scalar symbolic left arrow function. See Figure A10.

Figure A12 displays the structure of the name control block for an array of symbolic left arrow functions.

	ARRAY NUMBER	ATTRIBUTE FLAGS
NAME		
DLB1	ALB1	DUB1
AUB1	DLB2	ALB2
DUB2	AUB2	NUMBER DIMENSION
NUMBER ARGUEMENT		

NAME CONTROL BLOCK

Figure A12. Name Control Block of an Array of Symbolic Functions

Array Symbolic Equals Function

The attribute number 19 denotes a variable whose value is an array of symbolic equals functions. The name control block of the variable contains the same information as the name control block for an array of symbolic left arrow functions. See Figure A12.

The array is treated as if it is an array of real single precision numbers. Each element contains the virtual page number of the first virtual page used to store the arithmetic expression text for the first domain of the element's definition. If an element is not defined, its value is zero.

The text for the definition of each element is linked together in the same manner as a scalar symbolic equals function. See Figure A11.

APPENDIX B

OPERATION CODES

The source code and the three address internal text generated by the compiler for each operator is given in Table B1.

The negative integer preceding each segment of internal text is the operator number. The text following the operator number consists of pointers to the name control blocks for the operands and the result. R and T_i denote references to temporary variables.

In the text for some of the operators (eg. F'(A), $A[C_1,\ldots,C_N]$) negative integers appear in addition to the pointers to the operand and the result name control blocks. These negative integers are used to specify the number of derivatives, the number of arguments, or the number of subscripts involved.

Table B1. Operation Codes

SOURCE CODE	INTERNAL TEXT
A ← B	-1 B A
- A	-2 A R
A - B	-3 A B R
A + B	-4 A B R
A // B	-5 A B R
A / B	-6 A B R
A * B	-7 A B R
A ↑ B	-8 A B R
$F\underbrace{'\ldots'}_{N}(A)$	-9 F -N A R
$A[C_1,\ldots,C_N]$	-10 A -N C_1 C_1...C_N C_N R
$A[C_{11}:C_{12},\ldots,C_{N1}:C_{N2}]$	-10 A -N C_{11} C_{12}...C_{N1} C_{N2} R
$A[C_{11}:*,\ldots,*:C_{N2}]$	-10 A -N C_{11} 0...0 C_{N2} R
$A'[C_1,C_2]$	-10 A -2 C_2 C_2 C_1 C_1 R
A || B	-11 A B R
$F(X_1,\ldots,X_N)[C_1,\ldots,C_M]$	-12 F -N X_1...X_N-M C_1 C_1... C_M C_M R
$F(X_1,\ldots,X_N)'[C_1,C_2]$	-12 F -N X_1...X_N-2 C_2 C_2 C_1 C_1 R
$F\underbrace{'\ldots'}_{L}(X_1,\ldots,X_N)[C_1,\ldots,C_M]$	-13 F -L -N X_1...X_N-M C_1 C_1... C_M C_M R
$F\underbrace{'\ldots'}_{L}(X_1,\ldots,X_N)'[C_1,C_2]$	-13 F -L -N X_1...X_N-2 C_2 C_2 C_1 C_1 R
| A |	-14 A R
$F(X_1, X_2,\ldots, X_N)$	-15 F -N X_1 X_2...X_N R
(A, B)	-16 A B R

Table B1 (cont'd.)

SOURCE CODE	INTERNAL TEXT
([C_{11}:C_{12},...,C_{N1}:C_{N2}], A)	-17 -N C_{11} C_{12}...C_{N1} C_{N2} A R
(A TO B BY C)	-18 A B C R
(A, B,..., C)	-3 B A T_i -18 A T_i C R
(A FOR B TIMES)	-20 A B R
A'	-27 A R
A = B	-41 A B R
A < B	-42 A B R
A > B	-43 A B R
A ¬= B	-44 A B R
A <> B	-44 A B R
A =¬ B	-44 A B R
A >= B	-45 A B R
A => B	-45 A B R
A <= B	-46 A B R
A =< B	-46 A B R
¬ A	-51 A R
A AND B	-52 A B R
A OR B	-53 A B R