1969

Evaluation of NAPSS Expression Involving Polyalgorithms, Functions, Recursion, and Untyped Variables

Lawrence R. Symes

Report Number:
69-033

EVALUATION OF NAPSS EXPRESSIONS INVOLVING POLYALGORITHMS, FUNCTIONS, RECURSION, AND UNTYPED VARIABLES

Lawrence R. Symes

February 1969
CSD TR 33

Evaluation of NAPSS Expressions Involving Polyalgorithms, Functions, Recursion, and Untyped Variables

ABSTRACT

This paper describes how arithmetic expressions are evaluated in NAPSS. A brief discussion is included covering the types of expressions permitted and where the distinctive operands arise. First, the flow through the arithmetic expression evaluator is given for arithmetic expressions which do not involve recursion, function evaluation or polyalgorithm calls. The handling of each of these three operations is then described separately.

Evaluation of NAPSS Expressions Involving Polyalgorithms, Functions, Recursion, and Untyped Variables

INTRODUCTION

The Numerical Analysis Problem Solving System (NAPSS) project has been undertaken at Purdue University to design an interactive system for solving numerical problems [1]. The system is designed to accept input in a language [2] which is closely akin to normal mathematical notation. It permits the user to manipulate directly quantities other than scalars: e.g., array, functions and array of functions. Polyalgorithms [3] are included to implement the basic mathematical procedures. This significantly reduces the amount of analysis required to solve a wide variety of problems using the system.

This paper is primarily concerned with the problem of how arithmetic expressions, involving the various operands and operators permitted in the language, are evaluated.

TYPES OF EXPRESSIONS

Rather than present a detailed description of the NAPSS language [4], we describe a sampling of the allowable constructions.

The arithmetic expression in NAPSS permits the direct manipulation of numeric scalars, vectors and arrays, symbolic and tabular functions, and variables which denote symbolic expressions. The user need not worry about the type or mode of the operands; rather, all that need concern him is whether or not the arithmetic expression is mathematically correct.

The input language, while linear, attempts to resemble normal mathematical notation. For this purpose several special characters have been included. For example: $\int$ for integration, || for absolute value, and ' for differential and transposition. But to permit the use of standard

terminals and aid the goal of machine independence for the system, the number of characters in the NAPSS language is limited to 63. This requires that some of the operators appearing in mathematics be functions in NAPSS. For example: $\partial f(x,y)/\partial y \Big|_{\substack{x=2\\y=3}}$ is written as DER(F(X,Y)/(y) |x←2,y←3) in NAPSS.

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.

Examples:

2A, A2←C and A B←C

mean

2*A, A2←C and A*B←C respectively.

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 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 resulting array is

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

From a numerical array a single element, a row, a column or any arbitrary contiguous subarray may be extracted. 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 0th row of A, and A[-1:2,1] denotes the column vector consisting of the 3rd through 6th elements of the 1st row 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 equivalence of $2f'_{1,3}(3.5)$.

Several examples of arithmetic expressions and assignment statements appear below.

i) A ←(B+C)|D-E|

ii) T = S+V↑2

iii) F(X) ← A X↑2 + B X + C

iv) G(X) = A X↑2 + B X + C

v) X ←X-F(X)/F'(X)

vi) W ← ∫L(X,Y), (X←0 TO 1)

vii) DOTPRODUCT(X) = X'X

viii) H(X) ←X↑2,(X=>0)← -(X↑2),(X<0)

ix) T(X) ←X-1,(X>1) ←X+1,(X<-1)←0

The left arrow operator (←) indicates that the arithmetic expression on the right is to be evaluated and its value is to be assigned to the variable on the left, similar to what FORTRAN's = signifies. The equals sign (=) has the more usual mathematical meaning. Statement two establishes that a future occurrence of T is equivalent to the expression S + V↑2. Values are only substituted for the variables in the expression to the left of the = when a value is needed for the variable on the left. Thus if the values of S or V should change between the definition of T and the use of T this is reflected in the value of T. Variables defined to the left of an = are referred to as equals variables and variables defined to the left of an ← are called left arrow variables, or simply variables.

The difference between statements three and four is similar to the difference between statements one and two. In the definition of F the variables A, B, and C have their current values substituted for them while in the definition of G they do not. Values are only substituted for A, B and C when a value of the function G is needed. 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.

In statement six the integral of L(X,Y) is computed for X on the interval (0,1). Since Y is not a variable of integration its current value is used.

Statement seven defines DOTPRODUCT to be a function which computes the dot product of a vector and illustrates that arguments to function may be array. Functions also may yield arrays on evaluation.

Statements eight and nine illustrate that functions may be defined to have different values on different domains. If there is no domain specified with the last definition of the function, this definition is used when the point of evaluation does not lie any of the explicitly stated domains.

BASIC CONSTRUCTION OF THE INTERPRETER

NAPSS source text is transformed by a compiler into an internal code consisting of twenty bit integers. This scheme has the advantage of removing some of the burden from the interpreter, and for statements which are repeatedly executed the decoding is performed only once.

The internal text generated for arithmetic expressions is a form of three address code. All operators, temporary variables and pointers are negative integers and all user variables are positive integers. This is done so that user variables may easily be detected by a simple scan of the text.

During compilation a name control block is created for each user variable. At this time the name control block is used as a name table entry. It contains the name of the variable and some basic attributes as to how it appeared in the program.

The name control block is used during execution to hold values, pointers to values, and definitive attribute information for the variable. When a name control block denotes a numeric scalar the actual value of the variable is stored in the name control block itself. If the name control block denotes something other than a numeric scalar, it contains a pointer to where the values are stored and information such as bounds, number of dimensions, and number of arguments.

There is a fixed set of name control blocks used for storing temporary results during the evaluation of arithmetic expressions. They contain the same fields as a user variable name control block except for a name field. These are referred to as temporary name control blocks.

The memory that a NAPSS program has 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.

As the number of user variables increases the name table size is dynamically increased by obtaining a page at a time from real memory. When a page is removed from real memory for the name table this page is not returned to real memory until the system is re-initialized.

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

Due to 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.

NORMAL 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 polyalgorithms is given in Figure 1.

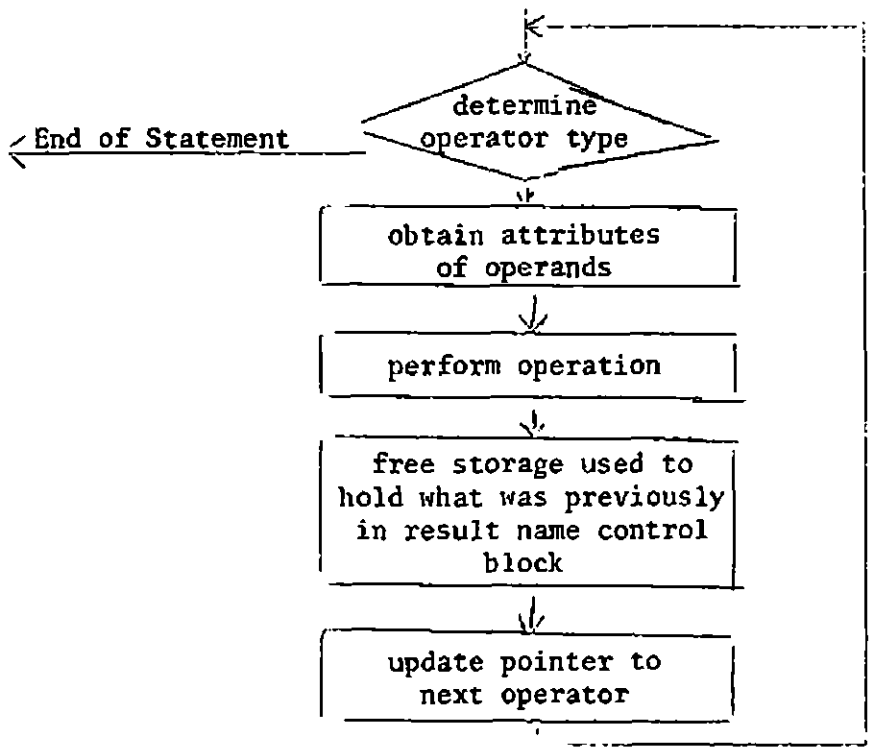

Figure 1. Flow of Control in Arithmetic Expression Evaluator

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.

At the same time that the attribute of a variable is determined, necessary pointers are obtained from the name control block so that the variable may be used as an operand.

When 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 samecextent 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.

Arrays are stored permamently 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. The work pool is a dynamic contiguous area of memory. It obtains its space from real memory. When an array operation is to be performed enough space is assigned to the work pool to hold the operands and resulting arrays. The required space is obtained by removing pages from real memory. When a page of real memory is assigned to the work pool it is removed from real memory until it is explicitly returned.

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 are conformal. 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. This 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 expression 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 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 ARITHMETIC EXPRESSION WITH RECURSIVE OPERANDS

The occurrence of an equals variable in an arithmetic expression causes the arithmetic expression evaluator to recurse. The recursion

needed to evaluate equals variables is limited to one routine, the master controller. 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.

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 so that they may be manipulated as a unit. 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 reused 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 reuses 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. 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 area 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 difference 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 variable the next time the arithmetic expression evaluation 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 by the portion of the interpreter which initially called the arithmetic expression evaluator.

There are three error messages 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 numbers of other errors. Level three prints the messages and numbers for all errors.

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 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 can not 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.

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, the arithmetic expression evaluator recurses and the actual parameter name control blocks 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 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 restroed. 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 later 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 reentered 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 reorganized.

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 of derivative occurred. The process of freeing storage associated with temporary name control blocks and the popping up 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 occur during an equals variable or a function. The actual message is issued by the routine which initially called the arithmetic expression evaluator.

ACKNOWLEDGEMENT

The work was supported in part by NSF Contract GP-05850.

REFERENCES

1. Rice, J. R., and Rosen, S., NAPSS - A Numerical Analysis Problem Solving System, Proc. ACM Natl, Conf. 21st, Los Angeles, 1966, ACM Publ. P-66, p. 51 (1966).
2. Symes, L. R., and Roman, R. V., Structure of a Language for a Numerical Analysis Problem Solving System, "Interactive Systems for Experimental Applied Mathematics" (M. Klerer and J. Reinfelds eds.). Academic Press, New York, 1968, p. 67.
3. Rice, J. R., On The Construction of Polyalgorithms for Automatic Numerical Analysis, "Interactive Systems for Experimental Applied Mathematics" (M. Klerer and J. Reinfelds eds.) Academic Press, New York, 1968, p. 301.
4. Symes, L. R., and Roman, R.V., Syntactic and Semantic Description of the Numerical Analysis Programming Language (NAPSS). Purdue Univ. Technical Rept., CSD TR 11, (1967).
5. Roman, R. V., and Symes, L. R., Implementation Considerations in a Numerical Analysis Problem Solving System, "Interactive Systems for Experimental Applied Mathematics" (M. Klerer and J. Reinfelds eds.) Academic Press, New York, 1968, p. 400.