ANALYTIC DIFFERENTIATION
OF A
FORTRAN IV FUNCTION SUBPROGRAM

by

PETER MADDEROM

B.A.Sc., University of British Columbia, 1966

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computer Science

We accept this thesis as conforming to the

required standard

THE UNIVERSITY OF BRITISH COLUMBIA

January, 1968

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date January 8, 1968

Abstract

A summary of the work done in the field of analytic differentiation by computer is presented. It is shown that there exists a need for an analytic differentiation routine whose output can be processed by a regular algebraic compiler. An algorithm is presented which will transform a FORTRAN FUNCTION subprogram into a FORTRAN SUBROUTINE subprogram that evaluates both the original function and its first derivative with respect to one independent variable. Implementation of this algorithm and possible extensions to it are discussed.

CONTENTS

ILLUSTRATIONS

Acknowledgment

I would like to take this opportunity to thank the National Research Council for its financial support of the project, Dr. J. M. Kennedy, Dr. C. Fischer, and Dr. J. R. H. Dempster for supplying ideas and corrections. A word of thanks also goes to Terrie, my wife, who typed the final manuscript without a murmur of complaint.

Introduction

In many applications in numerical analysis the first derivative of an algebraic expression is required. One example is finding the roots of an equation by the Newton-Raphson method. For an equation of the form: $f(x) = 0$, the iteration formula is $x_{n+1} = x_n - f(x_n)/f'(x_n)$. The method can be used for both algebraic and transcendental equations, and it also works when coefficients or roots are complex. Another example is the problem of determining the parameters a_i in the least-squares fit of the function $f = f(a_1, a_2, \ldots, a_n, x)$ to some function $g = g(x)$ (where g can be data). The matrix of the normal equations for the linearized problem contains all the partial derivatives $\frac{\partial f}{\partial a_i}$.

The earliest work on analytic differentiation by computer was done in 1953 by Kahrimanian [5] and independently by Nolan [9]. The limited input facilities and storage capacities of these first generation computers made analysis of the mathematical expression by the program prior to differentiation impossible. Consequently both programs had extremely primitive input/output.

Kahrimanian's program was written for the UNIVAC I. The input to the program was a 3-address pseudocode in which each code essentially represented one operator. The user had to fully parenthesize his expression and then code it in this pseudocode from the inner-most brackets outward. The internal

representation of the expression consisted of a table of pseudocodes. This tabular form had the elementary functions listed first and combinations of these functions in later lines. The program operated on this tabular form and generated a similar form which represented the derivative. That is, the result is a sequence of derivatives in the 3-address pseudocode where the nth order derivative of a composite function is expressed in terms of lower order derivatives. The output had to be recombined by the user to get the final analytic expression.

Nolan's program was written for the Whirlwind I at MIT. His program also required the input expression to be completely parenthesized and then encoded in a 3-address code by the user. In Nolan's case, however, the first line in the input table corresponded to the operator specified by the outermost parenthesis. The program differentiated the expression by introducing the differential operator into the first line of the table and then advancing it down through the lines by recursive applications of the elementary rules of differentiation. During this process, the tabular representation was transformed so that when the differential operator had been eliminated (by taking the derivative of a constant or of the independent variable), the resultant tabular form represented the desired derivative. The result had to be hand-translated to conventional mathematical notation.

J. R. Slagle [12] uses Nolan's differentiation

algorithm in his analytic integration program SAINT.

The Dartmouth Mathematics Project [1] also uses Nolan's algorithm. The input to their differentiation routine (they also developed routines for integration, solving first-order differential equations, and simplification) was full parenthetical. The output was also in the full parenthetical form. The elementary functions of logarithm, exponential, sine, tangent, cosine, arc sine, arc tangent, powers and roots are accepted.

A routine for the IBM 704 written by Hellerman [3] used machine instructions to represent the expression to be differentiated. The innermost terms were differentiated first and the procedure was continued outward until the whole expression was differentiated. The functions allowed were power, exponentiation, logarithm, sine, and cosine.

The work done at North Carolina in 1962 by Hanson, Caviness, and Joseph [2] allows the mathematical expression to be input in a compiler-like language. The language is defined as follows:

Operands/operation	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	P
Left parenthesis	(

Variable of differentiation	Any program - designated alphabetic character except P
Constants	The remaining unassigned alphabetic characters and the single decimal digits 0 to 9

The allowed transcendental functions are:

EXP	ARCCSC
LØG	SINH
SIN	CØSH
CØS	TANH
TAN	CØTH
CØT	SECH
SEC	CSCH
CSC	ARSINH
ARCSIN	ARCØSH
ARCCØS	ARTANH
ARCTAN	ARCØTH
ARCCØT	ARSECH
ARCSEC	ARCSCH

The set of expressions that can be differentiated is defined as:

1. If A is a variable or a constant, then A is an expression.

2. If A is an expression, then (A), +A, and -A are expressions.

3. If A and B are expressions, then A + B, A - B, A * B, APB, and A/B are expressions.

4. If A is an expression and FUN represents one of the allowable transcendental functions defined above, then FUN (A) is an expression.

5. The above are the only allowable expressions.

The representation of the expression in the machine is a table of triples which is equivalent to the Polish-prefix form of the expression. This table is called the M-matrix. As an example of this representation consider the expression: A + B * X. The M-matrix is:

line 1	B	*	X
line 2	A	+	line 1

The following precedence hierarchy is used to create the M-matrix:

function reference

exponentiation

unary minus

multiplication = division

subtraction = addition

left parenthesis = right parenthesis

The derivative of the expression in the M-matrix is placed in the D-matrix (another table of triples) using the following rules:

1. The derivative of a constant is zero.
2. The derivative of a variable is one.

If A and B are expressions and A' and B' their respective derivatives, then:

3. (A+B)' = A' + B'
4. (A*B)' = A'*B+A*B'
5. (+A)' = +A'
6. (-A)' = -A'

```
 7.  (A/B)' = (A'*B - A*B')/BP2
 8.  (APB)' = APB*(B*A'/A+B'*LØG. (A) )
 9.  (LOG. (A) )' = A'/A
10.  (EXP. (A) )' = EXP. (A)*A'
11.  (SIN. (A) )' = CØS. (A)*A'
12.  (CØS. (A) )' = -SIN. (A)*A'
13.  (TAN. (A) )' = SEC. (A)P2*A'
14.  (COT. (A) )' = -CSC. (A)P2*A'
15.  (SEC. (A) )' = TAN. (A)*SEC. (A)*A'
16.  (CSC. (A) )' = -CSC. (A)*CØT. (A)*A'
17.  (ARCSIN. (A) )' = A'/(1-AP2)P(1/2)
18.  (ARCCØS. (A) )' = -A'/(1-AP2)P(1/2)
19.  (ARCTAN. (A) )' = A'/(1+AP2)
20.  (ARCCOT. (A) )' = -A'/(1+AP2)
21.  (ARCSEC. (A) )' = A'/(A*(AP2-1)P(1/2) )
22.  (ARCCSC. (A) )' = -A'/(A*(AP2-1)P(1/2) )
23.  (SINH. (A) )' = CØSH. (A) * A'
24.  (CØSH. (A) )' = SINH. (A) * A'
25.  (TANH. (A) )' = SECH. (A)P2 * A'
26.  (CØTH. (A) )' = -CSCH. (A)P2 * A'
27.  (SECH. (A) )' = -SECH. (A)*TANH. (A)*A'
28.  (CSCH. (A) )' = -CSCH. (A)*CØTH.(A)*A'
29.  (ARSINH. (A) )' = A'/(1 + AP2)P(1/2)
30.  (ARCØSH. (A) )' = A'/(AP2 - 1)P(1/2)
31.  (ARTANH. (A) )' = A'/( 1 - AP2)
32.  (ARCOTH. ( A) )' = A'/(1 - AP2)
```

33. (ARSECH. (A))' = -A'/(A*(1 - AP2)P(1/2)

34. (ARCSCH. (A))' = -A'/(A*(AP2 + 1)P(1/2))

In the implementation of this algorithm, the D-matrix is an extension of the M-matrix. The rules of differentiation are applied to the M-matrix starting on the first row and proceeding down the matrix. Another table is generated which lists the row in the D-matrix corresponding to the derivative of the lines in the M-matrix. The entries generated for the D-matrix are checked for redundancies (such as addition of 0, multiplication by 1, etc.). These redundancies are removed as they are generated.

For output, the triples in the D-matrix are converted to a parenthesized string. This string has redundant brackets removed by application of the operator precedence relations listed above.

FØRMAC (FØRmula MAnipulator Compiler) [10] is an experimental programming system to the IBM 7090/94 for performing algebraic manipulations on complicated expressions. FØRTRAN IV is a subset of the FØRMAC language. The concept of the FØRMAC variable is introduced into the FØRTRAN IV language together with a number of new operators. The most important of these operators is the differentiator. A description of the FØRMAC differentiation algorithm and simplification capabilities is given in Appendix I. A FØRMAC variable represents an arithmetic expression. This expression is input in regular FØRTRAN IV notation but is represented

internally as well-formed formulas in delimiter Polish. This language is defined in Appendix I.

The FØRMAC commands which generate FØRMAC variables are:

LET: constructs specified expressions; it is the algebraic equivalent of the normal assignment statement in FØRTRAN.

SUBST: replaces a variable with an expression

EXPAND: removes all parentheses by applying the multinomial and distributive laws.

CØEFF: obtains the coefficient of a variable or of the variable raised to a power.

PART: separates expressions into terms, factors, or exponents.

The following commands yield FØRTRAN variables (the first two yield real variables and the last two yield logical variables):

EVAL: evaluates an expression for specific numerical values.

CENSUS: counts words, terms, or factors.

MATCH: compares two expressions for equivalence or identity.

FIND: determines dependence relations and/or whether a particular variable appears in an expression.

FØRMAC was implemented by adding to the FØRTRAN library a set of object time subroutines for the FØRMAC commands. A preprocessor translates FØRMAC programs (which include regular

FØRTRAN statements) into legitimate FØRTRAN IV programs. These are then processed by the regular FØRTRAN IV compiler.

Schorr [11] (whose differentiation algorithm appears in Appendix II) uses syntax-directed analysis to differentiate algebraic expressions. He writes Backus Normal Form specifications of the algebraic expressions and associates a rule of differentiation for each specification. This syntax table is used by a syntax-directed compiler developed by E. T. Irons [4]. The method is quite versatile since it is easily modified to handle more complicated situations (for example, a series of interdependent expressions). Schorr points out extensions to the syntax table which, if implemented, would make the system capable of handling functions defined in an ALGØL-like procedure. The extensions do not, however, remove redundancies such as multiplication by unity.

For obtaining the numerical value of derivatives without actually developing the analytical expressions, the work of Wengert [15] deserves mention. A set of FØRTRAN subroutines was created which handles the evaluation of elementary functions such as sums, products, sines, cosines, etc. and their derivatives. For example, instead of writing Y = X*Z, he writes CALL PRØD(Y,X,Z). This subroutine contains the statements:

```
Y = X * Z
DY = X * DZ + DX * Z
```

where DY, DZ and DX are associated variables which contain the

values of the derivatives of Y, Z, and X respectively. The calling order is similar to that stipulated for the Kahrimanian program.

Lesk [6] implements Wengert's procedure by modifying the complex arithmetic facilities available on the 7090 FØRTRAN II and IV. Instead of requiring an additional variable for each variable, the imaginary part of a complex variable is used to represent the derivative. This method is easy to implement on a system which has the complex arithmetic feature. It does not require the calling sequence set-up necessary for Wengert's method.

All of the above methods of differentiation (except Wengert's and Lesk's) suffer from one major drawback: the resultant derivative cannot be processed by a standard algebraic compiler. FØRMAC circumvents the difficulty by the EVAL(uate) command. However, Neidleman [8] points out that using this command in FØRMAC increases the time for evaluation by a factor of 30 when compared to a standard FØRTRAN routine. Another disadvantage of this command is that all arithmetic is done in the single precision, real mode. In many cases double precision arithmetic is necessary in calculating a derivative.

Wengert's method overcomes both difficulties. All his subroutines are written in FØRTRAN. Thus there is no restriction on the mode of the arithmetic and the derivative evaluation program is compiled on a standard FØRTRAN compiler. Lesk's modification makes the user's task easier at the expense

of versatility. The one draw-back is simplification. The method does not allow any. The efficiency of the program depends entirely on the programmer's foresight in declaring which variables will be handled in the special way. A careless programmer can cause many additions of 0 and multiplications by 1.

The program that is needed for work in numerical analysis is one which will take as input a natural statement of the expression to be differentiated and which will output a derivative that can be directly processed by an algebraic compiler for computer use.

FØRTRAN IV [7] was chosen as the input and output language because it is widely used and because it is a powerful language when used to represent algebraic and transcendental expressions. FUNCTIØN subprograms were used as input since they represent a logical subset of FØRTRAN IV for representing algebraic functions. A FØRTRAN IV SUBRØUTINE subprogram which evaluates both the original function and its derivative is produced for output.

Differentiation Algorith for Arithmetic Assignment Statements

The internal representation of an arithmetic assignment statement is basically the same as that used by Hanson et al. It is a table of triples in which each triple corresponds to an operator and two operands. All legal FØRTRAN IV arithmetic assignment statements are accepted as input and are converted to the table of triples using the following precedence hierarchy:

function reference

exponentiation

unary minus

division

multiplication

addition

subtraction

The FØRTRAN IV functions accepted are:

EXP	TANH	CSQRT
SQRT	ERF	CEXP
SIN	GAMMA	CLØG
CØS	ALGAMA	CSIN
TAN	ABS	CCØS
CØTAN	FLØAT	CABS
ATAN	DEXP	SNGL
ARSIN	DSQRT	REAL
ARCØS	DLØG	AIMAG

ALØG	DLØG10	DBLE
ALØG10	DSIN	CØNJG
SINH	DCØS	DABS
CØSH	DATAN	

During the initial scan of the FUNCTIØN subprogram a list of dependent variables is assembled. All other variables except the independent variable are classified as constants.

The differentiation algorithm is defined by the following rules:

1. D(constant) → 0.0
2. D(independent variable) → 1.0
3. D(dependent variable) → associated derivative variable
4. D(A = B) → D(A) = D(B) Note: A is always a dependent variable in this case
5. D(A ± D(B) → D(A) ± D(B)
6. D(A *B) → (D(A) * B) + (A* D(B))
7. D(A/B) → (D(A) - ((A/B) * D(B)))/B
8. D(A**B) → (A**(B-1.0))*((ALØG(A)*(A*D(B)))+(B*D(A)))
9. D(EXP(A)) → D(A) * EXP(A)
10. D(SQRT(A)) → D(A) * (0.50/SQRT(A))
11. D(SIN(A)) → D(A) * CØS(A)
12. D(CØS(A)) → D(A) * (-SIN(A))
13. D(TAN(A)) → D(A)/(CØS(A) * CØS(A))
14. D(COTAN (A)) → D(A)/(-(SIN(A) * SIN(A)))

15. D(ATAN(A))⟶D(A)/(1.0 + (A * A))

16. D(ARSIN(A))⟶D(A)/SQRT((1.0 - (A * A)))

17. D(ARCØS(A))⟶D(A)/(-SQRT((1.0 - (A * A))))

18. D(ALØG(A))⟶D(A)/A

19. D(ALØG10(A))⟶D(A)*(0.43429448/A)

20. D(SINH(A))⟶D(A) * CØSH(A)

21. D(CØSH(A))⟶D(A) * SINH(A)

22. D(TANH(A))⟶D(A)/(CØSH(A)*CØSH(A))

23. D(ERF(A))⟶D(A)*(1.1283792 * EXP((-(A*A))))

24. D(GAMMA(A))⟶D(A) * (DIGAMA(A)*GAMMA(A))

25. D(ALGAMA(A))⟶D(A) * DIGAMA(A)

26. D(ABS(A))⟶0.0

27. D(FLØAT(A))⟶0.0

28. D(DEXP(A))⟶D(A) * DEXP(A)

29. D(DSQRT(A))⟶D(A) * (0.50/DSQRT(A))

30. D(DLØG(A))⟶D(A)/A

31. D(DLØG10(A))⟶D(A)*(0.4342944819032518/A)

32. D(DSIN(A))⟶D(A) * DCØS(A)

33. D(DCØS(A))⟶D(A) * (-DSIN(A))

34. D(DATAN(A))⟶D(A)/(1.0+(A * A))

35. D(CSQRT(A))⟶D(A) * (0.50/CSQRT(A))

36. D(CEXP(A))⟶D(A) * CEXP(A)

37. D(CLØG(A))⟶D(A)/A

38. D(CSIN(A))⟶D(A) * CCØS(A)

39. D(CCØS(A))⟶D(A) * (-CSIN(A))

40. D(CABS(A))⟶0.0

41. D(SNGL(A)) ⟶ SNGL(D(A))

42. D(REAL(A)) ⟶ REAL(D(A))

43. D(AIMAG(A)) ⟶ AIMAG(D(A))

44. D(DBLE(A)) ⟶ DBLE(D(A))

45. D(CØNJG(A)) ⟶ CØNJG(D(A))

46. D(DABS(A)) ⟶ 0.0

After differentiation the result is simplified using the following transformations:

1. A ± 0.0 ⟶ A
2. 0.0 - A ⟶ -A
3. A * 1.0 ⟶ A
4. A * 0.0 ⟶ 0.0
5. A - A ⟶ 0.0
6. A/A ⟶ 1.0
7. 0.0/A ⟶ 0.0
8. 0.0^{A} ⟶ 0.0 for all A
9. $A^{0.0}$ ⟶ 1.0 except for A = 0.0
10. $A^{1.0}$ ⟶ A

The result is then converted back to a form acceptable to a FØRTRAN IV compiler. This output form is full parenthetical

Algorithm for Differentiating a FØRTRAN FUNCTIØN Subprogram

The algorithm will apply to subprograms which obey the following restrictions:

1. The algebraic function defined by the subprogram must everywhere have a derivative.

2. All temporary (or intermediate) storage space (except the FUNCTIØN name) must be defined in the subprogram.

3. Storage set by DATA statements must not be modified by the subprogram.

4. If the type of a variable is declared, the declaration statement must precede the first reference to the variable in an executable statement.

5. The independent variable must be a simple variable.

6. Calls to SUBRØUTINES or I/Ø must not depend on the independent variable.

7. FUNCTIØN references must have only one argument.

The necessity of having restriction 1 is obvious: it serves no purpose differentiating a function whose derivative does not exist. Restriction 2 ensures that the subprogram calculates one and only one function. Restriction 3 reflects common programming practice. By insisting on this restriction, the argument list of DATA statements does not have to be scanned during the compilation of the subprogram. A one-pass compile technique is possible because of restriction 4. Prior to execution of a subprogram, it is not, in general, possible to

determine the values of subscripts. This fact makes differentiation with respect to a subscripted variable a more difficult problem. Removal of restriction 5 will be discussed in a later section. Restriction 6 is necessary because the dependence relationship of the variables in the argument list on the independent variable cannot be established. The last restriction defines the independent variable in a user supplied FUNCTIØN (either statement or subprogram).

The algorith involves two passes over the statements in the subprogram. The first pass compiles the statements into a form acceptable to the differentiation pass. The second pass uses the differentiation algorith described in the previous section to differentiate flagged (by the first pass) arithmetic assignment statements. Its output is a FØRTRAN IV SUBRØUTINE subprogram which evaluates both the original FUNCTIØN and its derivative.

The Compilation Pass

The following statements are recognized and left unaltered:

ASSIGN	PAUSE
BACKSPACE	PRINT
CALL	PUNCH
CONTINUE	READ
DATA	RETURN
ENDFILE	REWIND

EXTERNAL	STOP
FORMAT	TYPE
GO TO	WRITE

All type declaration statements are handled in a similar manner. The statements are:

COMMON

COMPLEX

DIMENSION

DOUBLE PRECISION

INTEGER

LOGICAL

REAL

Every variable declared in one of these statements is inserted in a variable vector and type and/or dimension information is inserted in an information vector.

The following FUNCTION statements are accepted:

COMPLEX FUNCTION

DOUBLE PRECISION FUNCTION

FUNCTION

REAL FUNCTION

During the compilation pass, these statements are transformed according to the following rules:

FUNCTION A (list) → SUBROUTINE NAME (A, DIFNME, list)

REAL FUNCTION A (list) → SUBROUTINE NAME (A, DIFNME, list)
REAL A

where NAME is the subroutine name specified by the user,

DIFNME is the name of the derivative of A and is also specified by the user, and 'list' is the argument list of the FUNCTIØN A.

Because we are using a one-pass compiler, statement numbers already used in the subprogram cannot be anticipated during the compilation pass. In order to process DØ statements and some cases of logical IF statements, a sequence of unused statement numbers is required. This sequence, i_1, i_2, ..., i_n of statement numbers must be supplied by the user.

The DØ statement:

```
DØ   n   I = m1, m2, m3
...
n   legal statement
```

is changed to:

```
DØ  ij    I = m1, m2, m3
n   legal statement
ij  CØNTINUE
```

where i_j is one of the user supplied statement numbers and n is the original statement number of the statement ending the range of the DØ statement.

Arithmetic IF statements are left unaltered. Since logical IF statements can contain an arithmetic assignment statement, they are scanned for this possibility. If this is not the case, the IF statement is left unaltered. If it is the case, the following replacement rule is used:

```
IF(LØG) A = B ---> IF(.NØT.LØG)  GØ TØ ij
                   A = B
                ij CØNTINUE
```

where LØG is a logical expression, i_j is a user supplied statement number, and A = B is an arithmetic assignment statement. The next statement to be processed by the compiler is

the statement: A = B.

The arithmetic assignment statements are the important part of the subprogram. If the variable on the left-side of the equal sign is either an integer or a logical variable, the statement is by-passed. If not, the variable is entered in the variable stack and flagged (if it is already present it is still flagged). The statement is converted to the line representation (matrix of triples) and the line number corresponding to the equal sign together with the card number in the output subroutine is stored in a list. This list contains the locations of all the output card images and also indicates which statements have to be differentiated.

Statement functions are treated in a similar manner. The function name, its type, and its argument are stored in a table. If the type is integer or logical, the statement is then ignored. If this is not the case, the statement is converted to the line representation and the line number corresponding to the equal sign is stored in the above mentioned table.

Every pair of variables equivalenced in an EQUIVALENCE statement is stored in an other stack.

The END statement signals the completion of the compile pass.

The Differentiation Pass

The first step of the differentiation pass is to set up the list of associated variables corresponding to the derivatives

of the variables defined in the subprogram. Since only variables which appear on the left side of arithmetic assignment statements or in type declaration statements are present in the variable list, it is not possible to determine whether a variable name has been used in the subprogram. Therefore, a sequence of unused variables names must be specified by the user. The variable list is scanned and for every flagged variable in the list a derivative variable is inserted in the derivative stack. If a variable is not flagged it is a constant and its derivative entry is set to 0.0. Type declaration statements are generated for derivative variables when the corresponding dependent variables had their type declared. The derivative entry for the independent variable is set to 1.0.

The stack of equivalenced variables is checked next. If it is empty, the derivative variable stack has been completely set up. If some variables have been equivalenced, the following six cases must be considered: (The notation (A,B) means A equivalent to B, D(A) = QQ1 means the derivative variable associated with A is QQ1).

1. If (A,B), D(A) = QQ1, and D(B) = QQ2 then the derivative table must be modified to make D(B) = QQ1.

2. If (A,B), D(A) = 0.0, and D(B) = QQ1 then the derivative table must be modified to make D(A) = QQ1.

3. If (A(1), B(2)), D(A) = QQ1, and D(B) = QQ2 then (QQ1(1), QQ2(2)).

4. If (A(1), B(2)), D(A) = QQ1, and D(B) = 0.0 then modify the derivative table to make D(B) = QQ2 and make (QQ1(1), QQ2(2)).

5. If (A,B), D(A) = 1.0, and D(B) = 0.0 then the derivative table must be modified to make D(B) = 1.0.

6. If, for any pair of equivalenced variables, both derivatives are zero, the equivalence can be ignored.

Case 5 requires some elaboration. If the independent variable were equivalenced with an array restriction 5 would be effectively violated. If the derivative of the equivalent variable were not zero then the independent variable would depend on some other variable (clearly a contradiction).

In order to handle statement functions correctly, restriction 7 must be extended. The argument in the statement function must be the only dependent variable in its definition. For example F(A) = A + X where X is the independent variable is not allowed. The reason becomes obvious when the method of dealing with these statement functions is described.

If the stack containing the names of the statement functions is empty, this section is omitted. For each statement function, the following procedure is repeated. The argument of the function is inserted in the variable stack. (If it is already present, no action occurs.) The contents of the associated derivative variable are stored and reset to 1.0. The algorithm for differentiating an arithmetic assignment

statement is applied to the line representation of the statement function. The result is a statement function with the same argument which evaluates the derivative of the original with respect to its argument. This function is inserted in the output SUBRØUTINE directly before the original statement function. The name the statement function together with the name of its derivative function are inserted in the differentiation program. This program uses these names in formulas similar to that used for the derivative of a sine. For example, if the two names are F and DF then the differentiation rule generated is D(F(A))⟶D(A)*DF(A). The final step is to reset the derivative variable to its original value. If the generated statement is longer than 10 card images, the statement is split into a number of shorter statements and the derivative of the original statement function becomes a separate FUNCTIØN subprogram.

Every arithmetic assignment statement flagged in the original subprogram is now differentiated. If the resultant derivative requires more than 9 continuation cards, the statement is split up into a suitable number of interrelated arithmetic assignment statements. This set of statements is placed immediately before the differentiated arithmetic assignment statement in the output SUBRØUTINE subprogram. The statement number of the original statement is transferred to the first statement of the set.

The resultant SUBRØUTINE subprogram evaluates both

the original function and its derivative. If the user supplied statement number sequence and variable sequence are legal and if the FUNCTIØN subprogram is a legal FØRTRAN IV subprogram obeying the restrictions mentioned above, then the SUBRØUTINE subprogram generated by this algorithm will be accepted by a FORTRAN IV compiler (except for the possibility of mixed mode in a few special cases).

Verification of the Subprogram Differentiation Algorithm

It is clear that the algorithm does not change the logical flow of the original subprogram. That is, the original arithmetic assignment statements are still executed in the same order as they were prior to the differentiation process. Also, since differentiation introduces only new variables on the left side of arithmetic assignment statements, all the original variables remain defined in exactly the same way as they were defined prior to differentiation. Therefore, the differentiation process does not affect the original function evaluation.

Let us consider a FUNCTIØN subprogram which has no I/O statements and no statement functions and whose statements are rearranged in the following order:

```
 0        FUNCTION  F(X, list)
 1
 .
 .        type declaration statements
 .
 m
m+1
 .
 .        executable statements
 n
n+1       END
```

Any subprogram which obeys the original restrictions can be put into this form without changing the value of the function. (Statement functions can be substituted into places where they are referenced, read statements can be replaced by arithmetic assignment statements, and write statements do not affect the function evaluation and may be dropped).

The executed arithmetic assignment statements and the order in which they are executed depends on the input parameters (parameters in the argument list and in COMMON statements) but on nothing else. Also, since we assume that the derivative of F exists, the function evaluated at $x - e$, x, and $x + e$ (where x is the independent variable and $0 < e << 1$) will all be evaluated by the same series of assignment statements or else by a different set which give the same result for both the function and its derivative (else there would be a discontinuity at x).

Now consider the subprogram for a given set of input values. All transfers can now be eliminated by introducing duplicate sections of arithmetic assignment states. DØ loops can be eliminated by repeating the statements inside their range the number of times the DØ is to be executed and by placing an arithmetic assignment statement giving the correct value of the DØ index at the start of each repeated section. CONTINUE statements can be deleted along with logical assignment statements. The resultant subprogram will be of the following form:

```
0       FUNCTION F(X, list)
1
.
.       type declaration statements
.
m       v1 = f1(x)
m+1     v2 = f2(x, v,)
.
.       . . .
.
l       vs = fs(x, v1, v2, ..., vs-1)
l+1     RETURN
l+2     END
```

Variable v_s can be considered to be F since if this were not the case line l would not contribute to the evaluate of F and could be dropped.

Only non-integer v_i are dependent variables and x is the independent variable. Differentiating the dependent variables, we get:

$$q_1 = f_1'(x)$$
$$v_1 = f_1(x)$$
$$q_2 = f_2'(x, v_1, q_1)$$
$$v_2 = f_2(x, v_1)$$
$$\vdots$$
$$q_s = f_s'(x, v_1, q_1, \ldots, v_{s-1}, q_{s-1})$$
$$v_s = f_s(x, v_1, v_2, \ldots, v_{s-1})$$

The variable q_s is obviously the derivative of F with respect to the independent variable x. Since this process can be repeated for any set of input parameters, the algorithm is successful as long as derivative exists.

Implementation

In order to obtain a simple differentiation program, the following additional restrictions were placed on the input FUNCTION subprogram:

1. DØ loops must end on CØNTINUE statements.

2. A logical IF statement must not contain an arithmetic assignment statement.

3. User supplied functions must not be included (this includes both statement and subprogram FUNCTIONS).

4. EQUIVALENCE statements are not allowed.

In the MAP (Macro Assembly Program) language program written for the IBM 7040/44 to implement the algorithm, the various FORTRAN IV statements are processed in the following way:

These statements are recognized but not processed:

ASSIGN	IF
BACKSPACE	PAUSE
CØMMØN	PRINT
CØNTINUE	PUNCH
DATA	READ
DØ	RETURN
ENDFILE	REWIND
EXTERNAL	STØP
FØRMAT	TYPE
GØ TØ	WRITE

The following FORTRAN statements result in a warning message if processed:

BLØCK DATA

CALL

EQUIVALENCE

INTEGER FUNCTIØN

LØGICAL FUNCTIØN

SUBRØUTINE

In order to inform the program about the independent variable and user supplied information for output format, a TITLE card has been devised. The format of this optional card is:

TITLE [DIFNME/DY] [,DIMENS/1] [,INDPEN/X]
[,PUNCH/NØ] [,SUBNME/DIFFER] [,TMPNME/QQ]

The square brackets indicate optional entries. If the card is not included at the front of a FUNCTION deck, standard options are used. The significance of the symbols (with standard options in parenthesis) is:

DIFNME	=	name of the derivative of the FUNCTION (DY)
DIMENS	=	dimension of the independent variable (must be 1 for the present implementation)
INDPEN	=	name of the independent variable (X)
PUNCH	=	name of the generated subroutine (DIFFER)
TMPNME	=	first two characters of temporary variables used (QQ)

All other legal statements are processed in the manner described in the algorithm for differentiating the subprogram.

The only difference is in the transformation of the differentiated arithmetic assignment statements back to a form acceptable to a FORTRAN IV compiler. The line representation is converted to Polish prefix and this is converted to full parenthetical. If the full parenthetical result has more than 250 words (variables and operators), it is possible that more than 9 continuation cards will be required for the statement. In this case, the program attempts to divide the results into a number of separate statements (at most 8). This is done by examining the last 16 lines of the line representation of the derivative and choosing those lines which have two previous lines as operands. Starting at the lowest numbered of these lines, it is made into a new statement by generating a line with an equal sign for operator, a temporary variable for the first operand and the original line as the other operand. All references to this line are then replaced by the new variable. The next line with two line references is then handled in a similar manner. The process is continued until the original derivative line is reached. If any of the statements generated by this process is still too large, an error message is printed and processing goes on to the next statement.

On the above implementation, the following points should be noted:

1. Although DØUBLE PRECISIØN and CØMPLEX FUNCTIØNS are processed, the result is not always free of mixed mode.

2. The function CMPLX(X,Y) and complex constants cannot be processed. This greatly reduces the value of the CØMPLEX FUNCTIØN.

3. The result of differentiating the exponential (**) is: D(A**B)⟶(A**(B-1.0))*((ALØG(A)*(A*D(B)))+(B*D(A)))
If B is an integer, (B - 1.0) and B*D(A) are both mixed mode expressions. If A is a double precision variable, ALØG must be replaced by DLØG. If A is a complex variable, ALØG must be replaced by CLØG.

4. In FØRTRAN IV, a complex or double precision supplied function (for example: CCØS, DLØG) must have its type declared in the program. The present system does not generate type declarations for supplied functions that it generates as derivatives.

Examples of Differentiated FUNCTIØNS

The first example (Figures 1 and 2) illustrates the following points:

1. logical variables are not differentiated.

2. integer variables are also not differentiated.

3. differentiation of the exponential (**) results in mixed mode in this case.

The second example (Figures 3 and 4) is a FUNCTION which approximates the logarithm of x for $1 \leq x \leq 2$. It illustrates the generation of type statements for variables. The sequence of temporary variables generated is also illustrated

```
      TITLE TMPNME/AH,SUBNME/JOHN,INDPEN/XX,DIFNME/AJUNE,PUNCH/YES
      REAL FUNCTION LOG(XX)
      REAL K(6)
      DATA K/0.894E-7,1.0000091,2.0005859,3.0311933,1.0787748,8.8952784/
      X=XX-1.0
      IF(X.LT.0.0.OR.X.GT.1.0) GO TO 2
      DEN=K(6)
      DO 1 I=1,5
      J=6-I
      DEN=K(J)+X/DEN
1     CONTINUE
      LOG=DEN
      RETURN
2     WRITE(6,3)
3     FORMAT(6H ERROR)
      RETURN
      END
```

Figure 1 Example 1 input

```
      OSUBROUTINEJOHN(LOG,AJUNE,XX)                                     DIF00000
      OREALAJUNE                                                        DIF00001
       REAL              LOG                                            DIF00002
       REAL K(6)                                                        DIF00003
       DATA K/0.894E-7,1.0000091,2.0005859,3.0311933,1.0787748,8.8952784/DIF00004
       AH2=1.0                                                          DIF00005
       X=XX-1.0                                                         DIF00006
       IF(X.LT.0.0.OR.X.GT.1.0) GO TO 2                                 DIF00007
       AH=0.0                                                           DIF00008
       DEN=K(6)                                                         DIF00009
       DO 1 I=1,5                                                       DIF00010
       J=6-I                                                            DIF00011
       AH=(AH2-((X/DEN)*AH))/DEN                                        DIF00012
       DEN=K(J)+X/DEN                                                   DIF00013
1      CONTINUE                                                         DIF00014
       AJUNE=AH                                                         DIF00015
       LOG=DEN                                                          DIF00016
       RETURN                                                           DIF00017
2      WRITE(6,3)                                                       DIF00018
3      FORMAT(6H ERROR)                                                 DIF00019
       RETURN                                                           DIF00020
       END                                                              DIF00021
```

Figure 2 Example 1 output

```
      TITLE DIFNME/DF,TMPNME/DV,SUBNME/DERIV ,PUNCH/YES
      FUNCTION F(X,A,B,C,D)
      LOGICAL A,GOGO
      INTEGER B,C
      DIMENSION D(C)
      GOGO=A.OR.X.GT.10.0
      IF(GOGO) GO TO 2
      F=D(C)
      N=C-1
      DO 1 I=1,N
      J=C-I
      F=D(J)+X*F
1     CONTINUE
      RETURN
2     F=SIN(X)+D(B)*X**(B-1)
      RETURN
      END
```

Figure 3 Example 2 input

```
      OSUBROUTINEDERIV(F,DF,X,A,B,C,D)                                  DIF00000
       LOGICAL A,GOGO                                                   DIF00001
       INTEGER B,C                                                      DIF00002
       DIMENSION D(C)                                                   DIF00003
       GOGO=A.OR.X.GT.10.0                                              DIF00004
       IF(GOGO) GO TO 2                                                 DIF00005
       DF=0.0                                                           DIF00006
       F=D(C)                                                           DIF00007
       N=C-1                                                            DIF00008
       DO 1 I=1,N                                                       DIF00009
       J=C-I                                                            DIF00010
       DF=F+(X*DF)                                                      DIF00011
       F=D(J)+X*F                                                       DIF00012
 1     CONTINUE                                                         DIF00013
       RETURN                                                           DIF00014
 2     DF=COS(X)+(D(B)*((X**((B-1)-1.0))*(B-1)))                        DIF00015
       F=SIN(X)+D(B)*X**(B-1)                                           DIF00016
       RETURN                                                           DIF00017
       END                                                              DIF00018
```

Figure 4 Example 2 output

(AH, AH2).

Examples three and four (Figures 5 to 8) are included to illustrate the dependence of the efficiency of the generated SUBROUTINE on the form of the original FUNCTION. The first case shows a perfectly legitimate use of nested calls to FORTRAN functions. The resultant subprogram is extremely inefficient. It has repeated calls to FØRTRAN functions which have already been evaluated. The result does illustrate the method used to break an arithmetic assignment statement into a series of interrelated statements. The second case uses a set up similar to that used by Wengert. The result is impressive. In this case there are only four redundant calls to FORTRAN functions. It can be seen that some care must be exercised in using the present system if the result is to be used in places where speed is an important factor.

Possible Extensions

A mode checking routine would be a useful addition to the program. This routine would check the mode of constants generated by the differentiation routine and would ensure that mixed mode could not occur. It would also generate type declaration statements for FORTRAN functions whose type is implicitly given by their name.

The set of restrictions given at the start of the implementation section can be eliminated by programming the

```
 TITLE PUNCH/YES
 FUNCTION Y(X)
 Y=SIN(SINH(TAN(TANH(ERF(COTAN(ALOG(X)))))))/
1COS(COSH(ATAN(EXP(SQRT(X+1.0)))))
 RETURN
 END
```

Figure 5 Example 3 input

```
0SUBROUTINEDIFFER(Y,DY,X)                                              DIF00000
 QQ2=1.1283792*EXP((-(COTAN(ALOG(X))*COTAN(ALOG(X)))))                 DIF00001
0QQ3=SIN(ALOG(X))*SIN(ALOG(X))                                         DIF00002
0QQ4=(1.0/X)/(-QQ3)                                                    DIF00003
0QQ5=(QQ4*QQ2)/(COSH(ERF(COTAN(ALOG(X))))*COSH(ERF(COTAN(ALOG(X))))DIF00004
1)                                                                     DIF00005
0QQ6=(QQ5/(COS(TANH(ERF(COTAN(ALOG(X)))))*COS(TANH(ERF(COTAN(ALOG(XDIF00006
1)))))))*COSH(TAN(TANH(ERF(COTAN(ALOG(X))))))                          DIF00007
0QQ7=(QQ6*COS(SINH(TAN(TANH(ERF(COTAN(ALOG(X)))))))-((SIN(SINH(TANDIF00008
1(TANH(ERF(COTAN(ALOG(X)))))))/COS(COSH(ATAN(EXP(SQRT((X+1.0)))))))DIF00009
2*(((((0.50/SQRT((X+1.0)))*EXP(SQRT((X+1.0))))/(1.0+(EXP(SQRT((X+1.DIF00010
30)))*EXP(SQRT((X+1.0))))))*SINH(ATAN(EXP(SQRT((X+1.0))))))*(-SIN( DIF00011
4COSH(ATAN(EXP(SQRT((X+1.0)))))))))                                    DIF00012
0DY=QQ7/COS(COSH(ATAN(EXP(SQRT((X+1.0))))))                            DIF00013
 Y=SIN(SINH(TAN(TANH(ERF(COTAN(ALOG(X))))))))/                         DIF00014
1COS(COSH(ATAN(EXP(SQRT(X+1.0)))))                                     DIF00015
 RETURN                                                                DIF00016
 END                                                                   DIF00017
```

Figure 6 Example 2 output

```
TITLE PUNCH/YES
FUNCTION Y(X)
A=ALOG(X)
A=COTAN(A)
A=ERF(A)
A=TANH(A)
A=TAN(A)
A=SINH(A)
A=SIN(A)
B=SQRT(X+1.0)
B=EXP(B)
B=ATAN(B)
B=COSH(B)
B=COS(B)
Y=A/B
RETURN
END
```

Figure 7 Example 4 input

```
0SUBROUTINEDIFFER(Y,DY,X)                                    DIF00000
 QQ=1.0/X                                                    DIF00001
 A=ALOG(X)                                                   DIF00002
 QQ=QQ/(-(SIN(A)*SIN(A)))                                    DIF00003
 A=COTAN(A)                                                  DIF00004
 QQ=QQ*(1.1283792*EXP((-(A*A))))                             DIF00005
 A=ERF(A)                                                    DIF00006
 QQ=QQ/(COSH(A)*COSH(A))                                     DIF00007
 A=TANH(A)                                                   DIF00008
 QQ=QQ/(COS(A)*COS(A))                                       DIF00009
 A=TAN(A)                                                    DIF00010
 QQ=QQ*COSH(A)                                               DIF00011
 A=SINH(A)                                                   DIF00012
 QQ=QQ*COS(A)                                                DIF00013
 A=SIN(A)                                                    DIF00014
 QQ1=0.50/SQRT((X+1.0))                                      DIF00015
 B=SQRT(X+1.0)                                               DIF00016
 QQ1=QQ1*EXP(B)                                              DIF00017
 B=EXP(B)                                                    DIF00018
 QQ1=QQ1/(1.0+(B*B))                                         DIF00019
 B=ATAN(B)                                                   DIF00020
 QQ1=QQ1*SINH(B)                                             DIF00021
 B=COSH(B)                                                   DIF00022
 QQ1=QQ1*(-SIN(B))                                           DIF00023
 B=COS(B)                                                    DIF00024
 DY=(QQ-((A/B)*QQ1))/B                                       DIF00025
 Y=A/B                                                       DIF00026
 RETURN                                                      DIF00027
 END                                                         DIF00028
```

Figure 8 Example 4 output

rest of the algorithm given in the previous chapter. It does mean that the TITLE card must have a statement number option. This option will specify the start of an unused sequence of statement numbers used to modify DO ranges and logical IF statements.

The scan routine could be rewritten to accept complex constants and the functions CMPLX, ATAN2, and DATAN2 (which have two arguments but which can be differentiated). This would also mean rewriting the output routine. The rewritten output routine could remove redundant parentheses and thus produce a more legible output.

There are two ways in which the code generated can be made more efficient. The first must be done in the compile pass. A dependency table must be generated which shows the variables that depend in some way on the independent variable. These variables will then be assigned associated derivative variables and all other variables will be treated as constants. This will eliminate using derivative variables whose value is identically zero. The second method of increasing the efficiency is the examining of the SUBROUTINE generated after the differentiation pass for common FORTRAN function references. These references would then be replaced by temporary variables and extra assignment statements with the references would be inserted. The coding required for this extension would be very complex.

The last and by far the most interesting extension

is the lifting of the restriction on the independent variable. Can the independent variable be a dimensioned variable? The first point to be made is that, in general, the value of subscripts cannot be determined until execution time. Since the differentiation program is a processor which is run before the subprogram is compiled, the coding generated must make a decision at execution time. This decision depends on the numerical values of the subscripts. The solution is to differentiate the arithmetic assignment statement as before except that the derivative of the independent variable becomes a function whose value is either 0.0 or 1.0. The derivative statement is then put in a DO loop and the arguments of the special function are the original subscript of the independent variable and the DO loop index. All other dependent variables must have derivatives which have one more dimension than their variable. An example is helpful:

```
        TITLE DIMENS/4
        FUNCTIØN F(X, N)
        DIMENSIØN X(N)
        IF(N•LT•2) GØ TØ 1
        F = 5.0 *X(N-1)*X(N)
        RETURN
1       F = 5.0 * X(1)
        RETURN
        END
```

Using the extended algorithm:

```
        SUBRØUTINE DIFFER(F, DY, X, N)
        DIMENSIØN DY(4)
        DIMENSIØN X(N)
        IF(N•LT•2) GØ TØ 1
        DØ 200 I = 1, 4
        DY(I) = 5.0*(ZØRØ(I, N-1)*X(N)+X(N-1)*ZØRØ(I, N))
  200   CØNTINUE
        F = 5.0*X(N-1)*X(N)
        RETURN
   1    DØ 201 I = 1, 4
        DY(I) = 5.0*ZØRØ(I, 1)
  201   CØNTINUE
        F = 5.0*X(1)
        RETURN
        END
```

The FUNCTIØN ZØRØ is:

```
        FUNCTIØN ZØRØ (I, J)
        ZØRØ = 0.0
        IF(I•EQ•J)ZØRØ = 1.0
        RETURN
        END
```

It should be pointed out that this method introduces many redundant operations.

Conclusions

The algorithm presented is suitable for differentiating FORTRAN IV FUNCTION subprograms that have derivatives and that obey a few minor restrictions which make compilation easier. An effective technique for differentiating with respect to a subscripted variable has been illustrated.

It is illuminating to compare this method with those of Wengert, Schorr, and the FORMAC system. The system's mode handling capabilities are as versatile as that of Wengert. It allows more natural expression of the problem (arithmetic assignment statements as opposed to a series of calls of subroutines). Because redundant operations are eliminated in differentiated arithmetic assignment statements, the resultant program is more efficient than that of Wengert.

Schorr's proposed extended syntax-directed differentiator can be applied to FORTRAN since a Backus Normal Form description of FORTRAN is possible. It is not possible, however, to use the syntax-directed compiler to eliminate all the redundant expressions. The result would again be less efficient.

The FORMAC system is designed for a general formula manipulation problem. It is not possible to use FORMAC if the required derivative is not in the real mode or if the expression must be executed on a FORTRAN IV compiler. FORMAC is superior for cases where the result can be simplified by collecting like terms.

Bibliography

1. Dartmouth Mathematics Project, "Symbolic work on high speed computers." Project Report No. 4, Dartmouth College, Hanover, N.H., 1959.
2. Hanson, J.W., Caviness, J.G., and Joseph, C., "Analytic differentiation by computer." CACM 5, 6 (June 1962) p. 349.
3. Hellerman, L., "A computer analytic method for solving differential equations." 1959 Proc. Eastern Joint Comput. Conf., Boston, Mass. (Dec. 1959) p. 238.
4. Irons, E. T., "The Structure and Use of the Syntax Directed Compiler" Annual Review in Automatic Programming, Vol. 3 (1963) p. 207.
5. Kahrimanian, H. G., "Analytic differentiation by a digital computer." M.A. Thesis, Temple U., Philadelphia, Pa., May 1953.
6. Lesk, Arthur M., "Dynamic Computation of Derivatives," CACM10, 9 (Sept. 1967) p. 571.
7. McCracken, D. D., A Guide to FORTRAN IV Programming, New York, John Wiley & Sons, Inc., 1965.
8. Neidleman, L. D., "An application of FORMAC," CACM 10, 3 (March 1967) p. 167.
9. Nolan, J., "Analytical differentiation on a digital computer." M.A. Thesis, Math. Dept., MIT, Cambridge, Mass., May 1953.

10. Sammet, J. E. and Bond, E. R., "Introduction to FORMAC," IEEE Transactions on Electronic Computers, Vol. EC-13 No. 4 (Aug. 1964) p. 386.

11. Schorr, H., "Analytic Differentiation using a Syntax-Directed Compiler," Computer Journal 7, 4 (Jan. 1965) p. 290.

12. Slagle, J. R., "A heuristic program that solves symbolic integration problems in freshman calculus." JACM 10, 4 (Oct. 1963) p. 507.

13. Tobey, R. G., "Experience with FORMAC algorithm design" CACM 9, 8 (Aug. 1966) p. 589.

14. Tobey, R. G. Bobrow, R. J., and Zilles, S., "Automatic Simplification in FORMAC," Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27 part 1 (Nov. 1965) p. 37.

15. Wengert, R. E., "A Simple Automatic Derivative Evaluation Program," CACM 7, 8 (Aug. 1964) p. 463.

16. Wilkins, R. D., "Investigation of a new analytical method for numerical derivative evaluation," CACM 7, 8 (Aug. 1964) p. 465.

Appendix I

The FORMAC Differentiation Algorithm

The external representation of strings accepted by the algorithm is FORTRAN IV expressions with the following accepted functions:

FMCLØG	natural logarithm	
FMCSIN	sine	
FMCCØS	cosine	
FMCEXP	exponential	
FMCATN	arc tangent	
FMCHTN	hyperbolic tangent	
FMCDIF	differentiation	
FMCØMB	combinatorial	integer functions
FMCFAC	factorial	
FMCDFA	double factorial	

The internal representation of mathematical expressions in FORMAC is a string in delimiter Polish. This mathematical system is defined as follows:

Let A be the set of primitive elements in this system (variables and constants in FORTRAN IV). An elementary function will be defined as a function that can be represented by an expression in the set B of expressions generated from the primitive elements by the following rules:

Basic operators:

Unary operators: -, FMCLØG, FMCSIN, FMCCØS, FMCEXP, FMCATN, FMCHTN, FMCFAC, FMCDFA

Binary operators: ↑, FMCØMB

Variary operators: +, *

Delimiter:]

Note: FORTRAN 1. binary A-B replaced by +A(-B)

2. binary / replaced by binary $\uparrow$(-1)

3. ** equivalent to $\uparrow$

Rules for generating expressions in B:

1. If a ϵ A, a ϵ B.
2. Let b ϵ B, then -b, FMCEXPb, ..., FMCDFAb ϵ B.
3. Let c, d ϵ B, then $\uparrow$c d and FMCØMBcd ϵ B.
4. Let c_1, c_2, ..., c_s ϵ B, then $+c_1c_2\cdots c_s]$ and $*c_1c_2\cdots c_s]$ ϵ B.
5. These are the only expressions in B.

Let D be the differential operator, it is unary.

Let C be generated by the following rule:

6. If b ϵ B, Db ϵ C.

The rules for forming Db are as follows: The expression is first scanned from right to left and the subexpressions which depend on the variable of differentiation are flagged. (In the rules this is indicated by a dot over the subexpression). Then the following rules are applied in a left to right scan:

1. Given Db, b ϵ B. If the lead (left-most) element of b is independent of the variable of differentiation, then Db $\rightarrow$ 0. Otherwise apply the transformations from the remainder of the table.
2. $D\dot{b} \rightarrow -D\dot{b}$

3. $D\ \dot{\text{FMCEXP}}\ \dot{b} \longrightarrow *\ \text{FMCEXP}\ b\ D\dot{b}]$

4. $D\ \dot{\text{FMCLØG}}\ \dot{b} \longrightarrow *\ \uparrow b - 1\ D\ \dot{b}]$

5. $D\ \dot{\text{FMCSIN}}\ \dot{b} \longrightarrow *\ \text{FMCCØS}\ b\ D\ \dot{b}]$

6. $D\ \dot{\text{FMCCØS}}\ \dot{b} \longrightarrow -\ *\ \text{FMCSIN}\ b\ D\ \dot{b}]$

7. $D\ \dot{\text{FMCATN}}\ \dot{b} \longrightarrow *\ \uparrow + 1 \uparrow b\ 2]\ -1\ D\ \dot{b}]$

8. $D\ \dot{\text{FMCHTN}}\ \dot{b} \longrightarrow *\ 4 \uparrow +\ \text{FMCEXP}\ b\ \text{FMCEXP} - b]\ -2\ D\ \dot{b}]$

9. $D\ \dot{\uparrow}\ \dot{a}\ \dot{b} \longrightarrow +\ *\uparrow a + b - 1]\ D\ \dot{a}\ b]\ *\ \text{FMCLØG}\ a\ D\ \dot{b} \uparrow a\ b]]$

10. $D\ \dot{\uparrow}\ \dot{a}\ b \longrightarrow *\ \uparrow a + b - 1]\ D\ \dot{a}\ b]$

11. $D\ \dot{\uparrow}\ a\ \dot{b} \longrightarrow *\ \text{FMCLØG}\ a\ D\ \dot{b} \uparrow a\ b]$

12. $D\ \dot{+}\ a_1 \ldots a_s] \longrightarrow +D\dot{a}_{i_1} \ldots D\ \dot{a}_{i_p}]$ where only those a_{i_j} occur which are dependent upon the variable of differentiation.

13. $D\ \dot{*}\ a_1 \ldots a_s] \longrightarrow +\ \ldots\ *\ a_1 \ldots D\ \dot{a}_i \ldots a_s] \ldots]$ where products are generated only for a_i which are dependent upon the variable of differentiation.

14. If $a \in A$, $D\ \dot{a} \xrightarrow{x} 1$ for $a = x$

$D\ \dot{a} \xrightarrow{x} \frac{da}{dx}$ for $a \neq x$ where $\frac{da}{dx}$ is the symbolic name of the derivative and "a" has been declared in a DEPEND statement.

The resultant expression is simplified using the FORMAC AUTSIM (AUTomatic SIMplification) routine. The following transformations are performed:

1. $\uparrow 0\ A \longrightarrow 0 \qquad A \neq 0$

2. $\uparrow 1\ A \longrightarrow 1$

3. $\uparrow A\ 0 \longrightarrow 1$

4. $\uparrow A\ 1 \longrightarrow A$

5. $\uparrow -A\ N \longrightarrow \begin{cases} -\uparrow A\ N & \text{if } N \text{ is an odd integer} \\ \uparrow A\ N & \text{if } N \text{ is an even integer} \end{cases}$

6. $--A \longrightarrow A$

7. FMCEXP FMCLØG $A \longrightarrow A$

8. FMCLØG FMCEXP $A \longrightarrow A$

9. $- * 3\ A - B\ C - D] \longrightarrow * - 3\ A\ B\ C\ D]$

10. $+ A_1 \ldots A_n] \longrightarrow + A_1 \ldots \underset{i \neq i_k}{A_i} \ldots A_n]$ where $A_k = 0$

$* A_1 \ldots A_n] \longrightarrow * A_1 \ldots \underset{i \neq k}{A_i} \ldots A_n]$ where $A = 1$

11. $*A_1 \ldots A_n] \longrightarrow 0$ if $A_k = 0$ for any k

12. $+ \ldots A_j \ldots] \longrightarrow + \ldots B_k \ldots]$

$(*)\quad j = 1,p \qquad (*)\quad k = 1,q$

where $A_i = + \ldots C_m \ldots]$

$m = 1,s$

$q = p + s - 1$

$B_1 = A_1,\ B_{i-1} = A_{i-1}$

$B_i = C_1,\ B_{i+s-1} + C_s$

$B_{i+s} = A_{i+1},\ B_q = A_p$

13. $\uparrow\uparrow ABC \longrightarrow \uparrow[A*BC]$

14. $-+ABC] \longrightarrow +-A-B-C]$

15. FMCLØG $* B_1 \ldots B_s] \longrightarrow +$ FMCLØG $B_1 \ldots$ FMCLØG $B_s]$

16. FMCLØG $\uparrow AB \longrightarrow *\ B$ FMCLØG $A]$

17. $\uparrow * A_1 \ldots A_s]\ C \longrightarrow * \uparrow A_1C \uparrow A_2C \ldots \uparrow A_sC]$

The last five transformations are used to collect and cancel terms in products and sums. In sums, numeric coefficients of like symbolic expressions are combined. In products, exponents of like symbolic expressions are combined

Simplification is possible because the expressions

are in "p-canonical" form. This form specifies that rule 12 is applied whenever possible and that all symbols except +, *, ↑ , and - are sorted in a set sorting order. The result of this sorting is that terms that can be combined always appear together. The associative and commutative laws are assumed to hold for any internal FORMAC expression.

Appendix II

Schorr's Differentiation Algorithm

The input to this program is an ALGOL-like language defined in Bachus normal form by the following:

```
<constant> :: = a|b|...|v|w|z
<independent variable> :: = x
<dependent variable> :: = y
<digit> :: = 0|1|...|9
<number> :: = <digit>|  <number>  <digit>|
              <number> . <number>| . <number>
<expression> :: = <constant> | <independent variable>|
              <dependent variable> | <number>|
              <expression> ↑ <expression>|
             - <expression>|
             + <expression>|
              <expression> * <expression>|
              <expression> / <expression>|
              <expression> + <expression>|
              <expression> - <expression>|
             ( <expression> )|
             sin ( <expression> ) |
             cos ( <expression> )
<statement> :: = <expression> = <expression>
<program> :: = <statement>
```

The language described in the above table is processed by a syntax-directed compiler [Irons] using the following

syntax table.

$\langle$statement$\rangle$ = :: $\langle$program$\rangle$ $\{p_1\}$

$\langle$expression$\rangle$ = $\langle$expression$\rangle$ = :: $\langle$statement$\rangle$ $\{p_{3.2} = p_{1.2}\}$

$\underline{\sin}$($\langle$expression$\rangle$)=:: $\langle$expression$\rangle$ $\{\underline{\sin}(p_2) \mid (p_{2.2}) * \underline{\cos}(p_2)\}$

$\underline{\cos}$($\langle$expression$\rangle$)=:: $\langle$expression$\rangle$ $\{\underline{\cos}(p_2) \mid -(p_{2.2}) * \sin(p_2)\}$

$\langle$expression$\rangle$ $\uparrow$ $\langle$expression$\rangle$ = :: $\langle$expression$\rangle$

$\{p_3 \uparrow p_1 \mid (p_3 \uparrow p_1) * ((p_1 * p_{3.2})/p_3 + p_{1.2} * \underline{\ln}\ (p_3))\}$

- $\langle$expression$\rangle$ = :: $\langle$expression$\rangle$ $\{-p_1 \mid -p_{1.2}\}$

$\langle$expression$\rangle$ / $\langle$expression$\rangle$ = :: $\langle$expression$\rangle$

$\{p_3/p_1 \mid (p_{3.2} * p_1 - p_3 * p_{1.2}) / ((p_1) \uparrow 2)\}$

$\langle$expression$\rangle$ * $\langle$expression$\rangle$ = :: $\langle$expression$\rangle$

$\{p_3 * p_1 \mid p_{3.2} * p_1 + p_3 * p_{1.2}\}$

$\langle$expression$\rangle$ - $\langle$expression$\rangle$ = :: $\langle$expression$\rangle$

$\{p_3 - p_1 \mid p_{3.2} - p_{1.2}\}$

$\langle$expression$\rangle$ + $\langle$expression$\rangle$ =:: $\langle$expression$\rangle$

$\{p_3 + p_1 \mid p_{3.2} + p_{1.2}\}$

($\langle$expression$\rangle$) = :: $\langle$expression$\rangle$ $\{(p_2) \mid (p_{2.2})\}$

+ $\langle$expression$\rangle$ = :: $\langle$expression$\rangle$ $\{+ p_1 \mid + p_{1.2}\}$

$\langle$independent variable$\rangle$ = :: $\langle$expression$\rangle$ $\{p_1 \mid 1\}$

$\langle$dependent variable$\rangle$ = :: $\langle$expression$\rangle$ $\{p_1 \mid p_1'\}$

$\langle$constant$\rangle$ = :: $\langle$expression$\rangle$ $\{p_1 \mid 0\}$

$\langle$number$\rangle$ = :: $\langle$expression$\rangle$ $\{p_1 \mid 0\}$

. $\langle$number$\rangle$ = :: $\langle$number$\rangle$ $\{.p_1\}$

$\langle$number$\rangle$. $\langle$number$\rangle$ = :: $\langle$number$\rangle$ $\{p_3 \cdot p_1\}$

$\langle$number$\rangle$ $\langle$digit$\rangle$ = :: $\langle$number$\rangle$ $\{p_2 p_1\}$

```
<digit> = :: <number>   {p1}
 0 = :: <digit>  {0}
   .
   .
   .
 9 = :: <digit>  {9}
 a = :: <constant>   {a}
   .
   .
   .
 w = :: <constant>   {w}
 x = :: <independent variable>   {x}
 y = :: <dependent variable>   {y}
 ß = :: <constant>   {ß}
```

The way the compiler uses this syntax table will be illustrated by the following example:

```
y + x = a/x + 1.0

 y     +    x     =    a     /    x     +    1 . 0
d.v.        i.v.       const.       i.v.       digit digit
 y           x          a           x          1     0
exp.        exp.       exp.        exp.       numb. numb.
y|y'        x|1        a|o         x|1         number
 \_____/    \_____/    \_____/     \_____/      1.0
  exp.       exp.       exp.        exp.        exp.
y + x | y' + 1   a/x↑(0*x-a*1)/((x)↑2)          1.0|0
                 \_____________/
                       exp.
                 a/x + 1.0|(0*x-a*1)/((x)↑2)+0
\_____________/
  statement
y+x+a/x + 1.0 | (0*x-a*1) /((x) ↑ 2)+0

     program
```

The precedence relations used by the compiler are:

standard functions

exponentiation

unary minus

division

multiplication

subtraction

addition

Schorr points out that by keeping the syntax table short the results generated contain many redundancies. To eliminate some of these, he suggests including a "constant expression" in the syntax table. In the writer's opinion this will not eliminate all redundancies since the derivative of the independent variable remains unity and multiplication by 1 is redundant.

Further suggested syntax changes include:

1. increase the number of standard functions
2. allow comments
3. allow more than one formula to be differentiated at a time
4. provide better output by expanding the syntax table
5. use declarations to specify constants, independent variable, and dependent variable

Although the modified input specifications are given by Schorr, the transformation rules are not supplied.

An example of input that would be accepted by this syntax directed compiler is:

```
begin independent variable (x); dependent
variable (y); constant (a, b, c);
   begin y = ax ↑2 sin cx + bx / cos x end
end
```