MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

ARTIFICIAL INTELLIGENCE PROJECT
AI Memo 97

MAC-M-310
June 10, 1966

SYMBOLIC INTEGRATION

by Joel Moses

A program has been written which is capable of integrating all but two of the problems solved by Slagle's symbolic integration program SAINT. In contrast to SAINT, it is a purely algorithmic program and it has achieved running times two to three orders of magnitude faster than SAINT. This program and some of the basic routines which it uses are described. A heuristic for integration, called the Edge heuristic, is presented. It is claimed that this heuristic with the aid of a few algorithms is capable of solving all the problems solved by the algorithmic program and many others as well.

Slagle's program SAINT (Symbolic Automatic INTegrator)[1] which was finished in 1961 is currently the only published general integration program. It is unquestionably a tour de force in recursive programming, in the use of heuristics, and in the simulation of human behavior by a computer. However, many people who have read the description of SAINT have obtained the impression that one must use heuristics in order to solve integration problems on the level of SAINT (i.e. freshman calculus exam problems). We shall consider here two approaches to integration. First, we find that in order to perform integration problems on the level of SAINT (and even better), one can do very nicely with a purely algorithmic program. Second, we consider heuristics which are necessary in order to be able to handle those cases which fall outside the range of the algorithms.

We have experimented with a program which has nine algorithms and have found it to be able to solve all but two of the problems that SAINT solved. It was able to solve one of the two problems attempted but not solved by SAINT. It is also capable of solving many problems not attempted by SAINT. We do not believe that the converse is true to any appreciable extent. The program runs at speeds which are frequently two to three orders of magnitude faster than SAINT even though much of the program is still uncompiled and chaining takes half the running time. We have also done some hand simulation on a heuristic for integration which appears remarkably powerful. The heuristic is - guess at the form of the solution, differentiate the form, and determine the values of the variables in the form (possibly by guessing again). The heuristic (called the Edge heuristic - for EDucated GuEss) is an extension of the method of integration by parts. It is recursive, and tends to generate subproblems which are progressively simpler. With this heuristic, for instance, we can solve the problems that our algorithmic program was unable to solve and which SAINT was. It is our hope that with heuristics such as this we shall be able to get the solution to problems which are truly difficult. We consider integrals in standard integration tables to be fairly simple and obtainable with relative ease with a small number of algorithms which are easily coded. The algorithms for the integration program were coded and debugged in about two weeks.

The algorithms which have been coded are presented below. Here R(z) and S(z) stand for rational functions (ratio of two polynomials) in z.

I) a) $x^{n-1} R(x^n)$ substitute $y = x^n$

$\int x^7/(x^{12}+1)dx$ becomes $\int y/(y^3+1)dy$

b) R(x) chains to a program written by Manove[2] which integrates rational functions by a method described in Hardy.

II) R(exp(x)) substitute y=exp(x)

$\int \exp(x)/(2+3\exp(2x))dx$ becomes $\int 1/(2+3y^2)dy$

III) $R(x, \sqrt{a+bx})$ substitute $y = \sqrt{a+bx}$

$\int x\sqrt{x+1}\,dx$ becomes $\int 2(y^2-1)y^2dy$

This problem could not be solved by SAINT.

IV) $R(x, \sqrt{a+bx^2})$

i) $a>0$, $b>0$ substitute $\tan(y) = x$

$\int 1/(4+x^2)dx$ becomes $\int 1/2\ \sec(y)dy$

ii) $a>0$, $b<0$ substitute $\sin(y) = x$

$\int x/(1-x^2)^{\frac{1}{2}}dx$ becomes $\int \sin(y)dy$

iii) $a<0$, $b>0$ substitute $\sec(y) = x$

V) $R(x, \sqrt{cx^2+bx+a})$

i) $c>0$ substitute $y = \sqrt{cx} + \sqrt{cx^2+bx+a}$

ii) $a\geq 0$ substitute $(\sqrt{cx^2+bx+a} - \sqrt{a})/x$

VI) R(x)F(S(x)), where the integral of R(x) is rational also

i) F= log

Solution is $F(S(x))\int R(x)dx - \int\int R(x)dx\ S'(x)/S(x)dx$

$\int x\log(x)dx$ becomes $1/2\ x^2\log(x) - \int x/2dx$

ii) F= arctan

Solution is $F(S(x))\int R(x)dx - \int\int R(x)dx\,S'(x)/(1+S^2(x))dx$

III) F=arcsin

Solution is $F(S(x))\int R(x)dx - \int\int R(x)dxS'(x)/\sqrt{1-S^2(x)}dx$

$\int x^2\arcsin(x)dx$ becomes $1/3\ x^3\arcsin(x) - \int 1/3\ x^3/\sqrt{1-x^2}dx$

VII) $\int R(F_i(a+bx))$ where the F_i are trigonometric functions

Preparatory step: $y=a+bx$ yielding $1/b\int R(F_i(y))$

I) Transform all trigonometric functions into sines and cosines and test to see if the result is of the form

a) $\sin^{2n+1}(y)R(\sin^2(y),\cos(y))$ substitute $x=\cos(y)$

$\int\sin(y)dy$ becomes $\int -1dz$

b) $\cos^{2n+1}(y)R(\cos^2(y),\sin(y))$ substitute $x=\sin(y)$

$\int\cos(y)\sin^2(y)dy$ becomes $\int x^2dx$

II) transform all functions into secants and tangents and determine if the resulting expression is of the form

a) $R(\tan(y),\ \sec^2(y))$ substitute $x=\tan(y)$

$\int \sec^2(y)/(1+\sec^2(y)-3\tan(y))dy$ becomes $\int 1/(x^2-3x+2)dx$

b) $\tan^{2n+1}(y)R(\sec(y),\ \tan^2(y))$ substitute $x=\sec(y)$

III) substitute $x=\tan(y/2)$

$\int 1/(a+b\cos(y))dy$ becomes $\int 1/((a-b)x^2+(a+b))dx$

VIII) Derivative Divides - In a product if the derivative of one of the terms equals a constant times the rest of the terms in the product, an appropriate answer is looked up in a table. This is as close as the program gets to having a table of integrals. It should be made clear that no table whatsoever is really necessary. A special case is made when the term under consideration involves exponentiation. In that case the derivative is taken of the base or exponent whichever is nonconstant.

$\int x\exp(x^2)dx$ becomes $1/2\exp(x^2)$

$\int \tan(x)\sec^2(x)dx$ becomes $1/2\tan^2(x)$

IX) In a sum each of the summands is integrated separately.

The program does a remarkably fast job of recognizing which of the algorithms is appropriate. SAINT spent much of its times performing this task. Each problem is examined by

algorithms VIII and IX. If these are not applicable [illegible]
routine called FORM is called in order to [illegible]
the remaining algorithms may apply. FORM [illegible]
terms and looks for special subexpressions [illegible]
containing trigonometric functions, [illegible]
Once it has found one, the appropriate routine [illegible]
This routine is responsible for checking that [illegible]
by FORM is accurate. If so, the algorithm [illegible]
problem is integrated, if not, the program [illegible]
Unfortunately each time a new algorithm is added [illegible]
Integration program a nontrivial change to [illegible]
made. Nontheless, FORM requires only twenty lines of [illegible]
code. We believe that SAINT could well [illegible]
a preprocessing routine such as FORM.

The major example presented by [illegible] integration of $x^4/(1-x^2)^{3/2}$ FORM directs [illegible] routine which handles case IV. There [illegible] sin(y)=x is made which yields [illegible] problem is sent to the routine which handles [illegible] the substitution z=tan(y) is made yielding [illegible] problem is sent to Manova's program [illegible] integral $1/3\ z^3 - z + \arctan(z)$ [illegible] solution - $1/3\ \tan^3(\arcsin(x)) - \tan(\arcsin(x))$ [illegible] - the same solution that was obtained by [illegible].

SAINT performed the same basic [illegible] solution, but also considered subproblems [illegible] rewarding, such as $\cot^4(y)$. It took 11 minutes to obtain [illegible] solution. Our program did it in [illegible] seconds [illegible] -sharing. It used approximately 11 seconds of [illegible] of which 7 probably were spent chaining to Manova's [illegible] and communicating with it through the disk.

Considering the success of the program, [illegible] it is fair to predict that a program [illegible] near future which will integrate most of the [illegible] standard table of integrals in a matter of [illegible] PDP-6 computer at Project MAC will have [illegible] word memory the time spent on garbage [illegible] vastly reduced and chaining [illegible] subroutines used by the program [illegible] While SAINT undoubtedly could well have [illegible] a large memory, we do not believe [illegible] performed as well as an algorithmic [illegible] speed and in the range of problems [illegible] unless it were drastically altered.

The trouble with the algorithms for integration is that they usually deal with classes of expressions which are unique in that each member of these classes is integrable. Since many (probably most) problems are not integrable in closed form, there are likely to be few such classes, and indeed few have been found. Hence the value of a purely algorithmic program hinges on its ability to handle very large, but not very complicated, problems. This is useful but, we feel, not very interesting nor difficult.

The truly difficult problems in integration are the ones whose integrability is really in doubt. In general there can be no decision procedure for integration. This was recently proved by Richardson.[3] We have found a simple decision procedure for the case $R(x)\exp(P(x))$ where R is rational and P is a polynomial. It is reputed that there exists a decision procedure for integrands which are algebraic functions (these include rational functions and roots of rational functions). This is very surprising, if true. One must resort to heuristics in order to be able to solve efficiently the problems that remain. We might try to differentiate all possible well formed expressions and match these with the integrand. This is clearly inefficient, and may take forever. One heuristic which we have found (which was motivated by discussions with Professor M. Minsky) is the Edge heuristic mentioned above. It is surprising how well it performs by itself. With the addition of a few algorithms such as factoring it should be able to solve all problems which are integrable by the algorithms and many others.

The rule which governs the use of the heuristic is as follows: determine the most complicated subexpression in the integrand, guess at a form of the integral whose derivative contains this subexpression, then determine the value of the variables in the form by making further guesses. There is theoretical evidence in the works of Liouville[4] that this is a very reasonable approach. We shall describe the guessing rules (as far as we know them at present) and give some examples below.

Below a,b shall stand for undetermined functions of x. The expression involving g(x) is the most complicated subexpression in the integrand, and f(x) is the rest of the integrand. We assume that the integrand is a product of terms.

Form of the integrand	Form of the integral	
$f(x)(g(x))^n$	$a(g(x))^n + b$	$n>0$
	$a(g(x))^{n+1} + b$	$n<0$
$f(x)\exp(g(x))$	$a(\exp(g(x))) + b$	

f(x)sin(g(x))	acos(g(x)) + b
f(x)cos(g(x))	asin(g(x)) + b
f(x)log(g(x))	alog(g(x)) + b

When these forms fail to yield an answer, then the integral may contain logarithmic terms (i.e. constants times functions of log, arctan, or arcsin).

f(x)g(x)	alog(b+cg(x)) + d

One should switch to the arctan or arcsin formulations when encountering complex constants in the above.

An example of the use of the Edge heuristic.

$x^4/(1 - x^2)^{5/2}$

Try $a(1 - x^2)^{-3/2} + b$
Differentiation yields

$-3/2\ a(1 - x^2)^{-5/2}(-2x) + a'(1 - x^2)^{-3/2} + b' = x^4(1 - x^2)^{-5/2}$

Try $a = x^4/(3x) = 1/3\ x^3$, hence $a' = x^2$

and $b' = -x^2(1 - x^2)^{-3/2}$

In order to solve for b try

$c(1 - x^2)^{-1/2} + d$

$-1/2\ c(1 - x^2)^{-3/2}(-2x) + c'(1 - x^2)^{-1/2} + d' = -x^2(1 - x^2)^{-1/2}$

Try $c = -x^2/x = -x$

Hence $d' = (1 - x^2)^{-1/2}$

The solution for d is arcsin(x), which is obtained from the logarithmic term $\frac{1}{2}\log(-ix + (1 - x^2)^{1/2})$.

The final answer is

$1/3\ x^3(1 - x^2)^{-3/2} - x(1 - x^2)^{-1/2} + \arcsin(x)$

We must explain some of the steps above which are not obvious. In this problem the 'hardest subexpression' is $(1 - x^2)^{-3/2}$. This 'hardest subexpression' is chosen from terms in the integrand (assuming that the integrand is a product) which can not appear in the derivative of the other terms. If a logarithm appears only once in the integrand, the term containing it is a good candidate for the most

complicated subexpression. In the differentiated expression a or a' is selected such that the term involving it will equal the integrand. We will choose b' so that it equals the rest of the expression.

The two problems which SAINT was able to solve and which the algorithmic program was not able to solve are $x\cos(x)$ and $x\exp(x)/(1+x)^2$. Both of these problems can be solved with the aid of the Edge heuristic. The solution of the latter problem led us to a decision procedure for the case $R(x)\exp(P(x))$. The problem $\cos(\sqrt{x})$ which was attempted and not solved by SAINT can likewise be solved by the Edge heuristic.

There are two criteria by which one determines when to terminate the use of a guess made by the Edge heuristic. When one has generated the same subgoal as the original problem, then clearly one must quit the current guess. When a subproblem is generated which is a constant multiple (different from one) of the original problem, then transposition is used to obtain the answer. When the subproblems generated by the heuristic tend to grow, one should give up the current guess, although this is a situation which is not so clear cut.

There are some cases with which the heuristic will encounter difficulties. In the case of rational functions, the denominator must be factored before the heuristic has a chance of succeeding. Otherwise, there is no simple way to make a guess at the 'most complicated subexpression'. A similar difficulty occurs in the case of rational functions of trigonometric functions. The algorithms clearly can take over in these cases.

Another source of difficulty occurs in those problems whose solution contains more than one logarithmic term. We know of no simple way to break up the problem into subproblems which yield the logarithmic terms. This is the reason why the case of algebraic functions is considered so difficult.

The real source of difficulty to an integration program, and to the Edge heuristic in particular, is due to the fact that the character of the integral may not be what it seems because some transformation has taken place which masks it. Suppose $g(x)$ is equivalent to zero, and we are asked to integrate it. Unless we realize the equivalence, we may have great difficulties in obtaining the result. This appears to be the real reason why integration is, in general, recursively unsolvable. Every expression which is integrable is equivalent to an expression which is trivially integrable, namely the unsimplified derivative of its integral. The difficulty with integration can be said to be due to the transformation which has taken place in the derivative. A single program can only counteract a finite number of these transformations, but their number, in some sense, is infinite.

Finally, the applicability of the Edge heuristic stops with linear, first order, differential equations. Since the form of the solution of nonlinear differential equations critically depends on constants in the equation, other heuristics will be necessary to obtain the solution to these problems.

We shall now describe two of the basic routines used by the algorithmic integration program.

The matching program - SCHATCHEN

There is a set of routines in SAINT which Slagle did not describe at all. These routines comprise a recursive matching program called Elinst (ELementary INSTance). In about two pages of LISP code Slagle wrote a very powerful routine which is so recursive that in months of examining and using it we have not been able to figure it out completely. It is so powerful that in the six years since it was written no previously documented attempt at a matching routine has come close to equaling its power, not Formula Algol[5] nor Korsvold's simplification program[6] nor the FAMOUS system of Fenichel.[7] Most unfortunately, Slagle badly misused this routine. He was faced with enormous difficulties in fitting his program in core and thus many of his subroutines made use of the power of Elinst in order to gain space at the expense of time. Probably a gain of an order of magnitude in speed could have been attained by avoiding Elinst in some matching situations and by using special purpose matches as we have done.

We have written a matching program called SCHATCHEN (Yiddish for match-maker) which is a take-off on Elinst. It is about as powerful as Elinst but is certainly not as wildly recursive.

SCHATCHEN is a function of two arguments - an expression and a pattern. Its purpose is to determine whether or not there exist values for the variables in the pattern for which the pattern becomes equivalent to the expression. Its value is either NIL or a dictionary of the values of the variables. It is very similar in purpose to the left-hand side of a COMIT rule, but is specifically designed for algebraic expressions.

SCHATCHEN assumes that the operators PLUS and TIMES are commutative operators with variable number of arguments. It is aware of the usual identities involving 0 and 1 for PLUS, TIMES and EXPT. It is this fact which gives it and Elinst much of their power over other matching programs which are bothered by missing operators.

Below we present a somewhat fictionalized description of the program and the patterns that it accepts.

If the expression equals the pattern, the match succeeds.

If the pattern is of the form (VAR a g arg1 arg2 ...) and the expression is e, then g(e arg1 arg2 ...) is evaluated. If the value of g is false, the match fails. Otherwise, the match succeeds and ((a . e)) is appended to a dictionary of values.

If the pattern is of the form (op p1 p2 ... p_ℓ q) where op is either PLUS or TIMES, then the following takes place. If the expression e is not of the form (op e1 e2 ... ek) then it is converted to (op e).

For each subpattern p_i a search is made of the expression for a subexpression e_j which will match it. If one is found, the match continues with the next subpattern. If none is found, a match is attempted between p_i and 0 if op=PLUS or 1 if op=TIMES. If the attempt succeeds the match continues with the next subpattern. Otherwise the match fails.

If after all the p_i have been matched, the expression still contains some terms, an attempt will be made to match q with them. Hence q serves the function of the $ in a COMIT rule. If no terms remain, then q must match 0 if op=PLUS or 1 if op=TIMES.

If op=PLUS and if one of the p_i is of the form (TIMES* r1 r2 ... r_m (VAR a s args)) then what is desired is that a should become the coefficient of (TIMES r1 r2 ... r_m) in the sum. This is done by looping over the remainder of the expression and matching (TIMES r1 r2 ... r_m (VAR a s args)) with each summand in it. The value of the match is the sum of the individual matches.

If the pattern is of the form (EXPT p1 p2) and the expression e is of the form (EXPT e1 e2), then p1 must match e1 and p2 must match e2, or p1 must match e and p2 must match 1. If e is 1, then p2 must match 0 or p1 must match 1. If e is 0, then p1 must match 0.

If the pattern is of the form (op p1 p2 ... pk), and the expression is of the form (op e1 e2 ... ek), then each p_i must match e_i.

All other matches fail.

Suppose we wish to match for a linear expression in x. The following pattern may be used.

```
(PLUS(TIMES* x (VAR B FREE))(VAR A FREE))
```

Here FREE is the name of a routine which checks to see if its argument contains an x. This pattern when matched with the expressions below at the left will yield the results at the right.

```
3                                   ((A . 3)(B . 0))

x                                   ((A . 0)(B . 1))

(PLUS x PI 2 (TIMES y x)) ((A (PLUS PI 2))(B (PLUS 1 y)))
```

There is another facility in SCHATCHEN which allows the pattern to specify a loop over the expression. Suppose we were interested in performing trigonometric simplification such as

$$a \sin^2(b) + a \cos^2(b) + e = a + e \quad .$$

The following pattern will perform the matching necessary for the application of this rule. TRUE is a function which will accept any input.

```
(PLUS(LOOP(TIMES(EXPT(SIN(VAR B TRUE))2)(VAR A TRUE))
 (TIMES(EXPT(COS(VAR C SCHATCHEN B))2)
    (VAR D SCHATCHEN A)))(VAR E TRUE))
```

With the pattern above a loop will take place over all $\sin^2$ and $\cos^2$ expressions until one pair is found which meets the criteria or until all are exhausted. We do not necessarily recommend this approach to the simplification of trigonometric expressions.

It would be useful to make some of the following extensions to SCHATCHEN. Currently SCHATCHEN can report a succesful match without yielding a dictionary containing a value for each variable in the pattern. This could be changed, for example, by giving default values to the variables. On a more semantic level, one should consider the VAR pattern to be only of a number of possible modes of definition of variables. The VAR mode corresponds to the BUV mode of CONVERT[8]. There should also be a mode corresponding to the UAR mode of CONVERT. This would accept any value which meets some criterion the first time the variable is encountered, but each successive value of the variable must match the first one. This is similar to the numerical constituent of COMIT which must match a previous constituent. From an algebraic standpoint SCHATCHEN is limited in that it performs no division. For instance, it cannot match a pattern which would correspond to ax to an expression such as x^2. Some sort of division should be introduced, but its applicability should be governed by the user through a new operator such as TIMES**. It would also be wise to maintain the mode declarations such as VAR outside the pattern, either as separate arguments to SCHATCHEN as in Elinst or CONVERT, or as some global declarations as in Formula Algol. Furthermore there should exist some facility for performing the construction of an expression through the use of the dictionary supplied by the match. This can range from a simple construction routine to a very powerful one as in CONVERT.

The simplification program - SCHVUOS

We have written yet another simplification program. It is called SCHVUOS which is an acronym for SCHatchen's Version of an Unassuming Operational Simplifier. It is remarkable in that it is very short. The basic simplification routines are only two pages of LISP code long. It assumes no standard form of the expression. Hence it is inefficient on very large expressions. However the size of an expression occuring in integration is relatively small, and hence it is quite fast for our purposes.

In a sum the summands are simplified. A summand is first stripped of its constants and a match is made for a constant multiple of this summand using the TIMES* facility of SCHATCHEN. This is done until each summand has been accounted for. In a product, after the simplification of all the terms, the program collects exponents by using SCHATCHEN and matching for a variable power of the terms in the product. The usual transformations involving 0 and 1 with the operators PLUS, TIMES, and EXPT are all made. Furthermore an exponential term whose base is a product is expanded, and constants are multiplied through sums.

Functions such as log and sin have their own simplification routines which perform non-controversial simplifications such as sin(0)=0.

There is an indicator which tells SCHVUOS whether or not to simplify subexpressions. It is usually on except during differentiation where the differentiated expression is built from the bottom up and is simplified at each level, with no redundant simplifications being performed.[9] This is SCHVUOS' alternative to the AUTSIM bit of FORMAC[9] and to Martin's version thereof.[10]

The following routine is not part of the current algorithmic integration program.

An Integral TAble Look-Up - ITALU (read I-tell-you)

We had spent some time building an integral table look-up before realizing that for integration it would quite likely be useless since the program could, in all probability, integrate anything in the table quickly. Since this is not the case for differential equations, definite integration, or summation of series, we would like to present what we have done in ITALU.

The basic steps in ITALU are as follows: An expression to be integrated is first hash-coded. The code (a floating point number) is looked up in an array (binary search) and a disk address for it is found. The disk is read and forms of the integrands in it are matched with the given expression (probably only one exists with the right code) until one is found which matches the expression. The coresponding integral is evaluated after substituting in it the results of the match.

For example, the hash-code for (EXPT E (PLUS PI (TIMES 2 x))1)) would be the same as for the form

```
(EXPT(VAR C FREE)(PLUS(TIMES*(VAR  V VARP)(VAR B FREE))
   (VAR A FREE))
```

(where VARP tests to see whther its argument is equal to x) and the integral found by ITALU would then be 1/2 (EXPT E (PLUS 1 PI (TIMES 2 x))). The trick of evaluating the integral allows the integral to be a function and thus allows iterated integrals to be easily obtained.

The hash-code is designed to check for the form of the expression and to ignore constants with respect to the variable of integration. Thus x is coded like a+bx , but sin(2x) is coded differently from $\sin(x^2)$ or cos(2x). Coding is done recursively. The code of a sum is the sum of the codes of the summands, ignoring constants. The code of a product is likewise the product of the code of the terms, ignoring constants. Trigonometric and logarithmic functions are coded by exponentiating the code of the arguments by a constant which is different for each function. The code for an exponential term is more involved. Since the exponents -1,1/2,-1/2,2,-2 occur so frequently, these are made special cases. All other constant exponents are given the same code. The code for (EXPT a b) is the code of a raised to the code of b.

We have also experimented with a hash-code which notes that a constant occurred, but ignores the value of this constant. Thus 1+x codes differently from x, but the same as 2+x.

The advantage of the scheme above is that it is quite fast (we believe that one can get running times of about a second per integral, most of which is spent accessing the

disk). Furthermore the speed of the look-up is not appreciably deteriorated with an increase in the number of entries in the table. The scheme yields a powerful table look-up because of the use of SCHATCHEN to perform the match, which allows variables to appear in the integrand just as they do in the standard integration tables. The use of the device of evaluating the integral allows iterated integrals to be conveniently entered in the table.

Integral table look-up schemes reported in the literature are to be found in (11).

BIBLIOGRAPHY

1) J.R. Slagle, A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus, Symbolic Automatic Integrator (SAINT), Ph.D. Thesis, MIT, 1961.

2) C. Engleman, Mathlab: A Program for On-Line Machine Assistance in Symbolic Computations, MITRE Corp., Report MTP-18, October, 1965.

3) J. Moses, Solution of Systems of Polynomial Equations by Elimination, CACM, August, 1966, to appear.

4) J. Ritt, Integration in Finite Terms, Liouville's Theory of Elementary Methods, New York, Columbia University Press, 1948.

5) A. Perlis et al., A Definition of Formula Algol, CACM, August, 1966, to appear.

6) K. Korsvold, An On-Line Program for Non-numerical Algebra, CACM, August, 1966, to appear.

7) R. Fenichel, Ph.D. Thesis, to appear.

8) A. Guzman and H. McIntosh, CONVERT, CACM, August, 1966, to appear.

9) R. Tobey, Experience with FORMAC Algorithm Design, CACM, August, 1966, to appear.

10) W. A. Martin, Hash-Coding Functions of a Complex Variable, Artificial Intelligence Project Memo 70, MIT, June, 1964.

11) L. Clapp, A Syntax-Directed Approach to Automatic Aid for Symbolic Math, CACM, August, 1966, to appear.