THE USER-LEVEL SEMANTIC MATCHING CAPABILITY IN MACSYMA

Richard J. Fateman
Project MAC, M.I.T, and Harvard University
Cambridge, Mass.

Summary

The pattern matching facility of MACSYMA, an algebraic manipulation system, is described in this paper. By taking advantage of the special semantic properties of algebraic expressions, diverse expressions are recognized as occurrences of the same pattern. For example, a semantic pattern for "quadratic in x" matches both 3*x**2+4 and (x+1)*(x+6).

Patterns are created by declaring variables to satisfy predicates, and then composing, out of these variables, expressions which serve as templates for the pattern matching process. Efficiency is achieved by compiling programs corresponding to each pattern.

Specific examples show how this recognition capability is used in augmenting simplification rules and in writing algorithms for the solution of differential equations.

Other systems with related capabilities are compared with regard to their implementations and matching strategies.

Introduction and Overview

When complex algorithms are coded in an algebraic manipulation language, it is sometimes advantageous to supplement the command language with a pattern recognition capability. In effect, a pattern recognition facility simulates the action of a human mathematician who, by examining the structure of a formula, decides on his next step. It is to our advantage to make this recognition capability relatively independent of the particular style in which the formula is expressed. In particular, such details as whether products are distributed over sums or not, should be irrelevant to the matching process.

Consider the problem of solving linear differential equations with constant coefficients. Before we can apply our knowledge in any generally useful manner, we must be able to recognize when a given expression is an equation, a differential equation, a linear differential equation, and a linear differential equation with constant coefficients. Because pattern matching can perform this type of decision-making which might otherwise require human intervention, it is an important adjunct to a computer-aided mathematical laboratory. Often, only when the computer can recognize a given pattern and its components, can it proceed to the next step in processing. Furthermore, pattern-matching capabilities are essential to building useful additions to a mathematical laboratory. Through pattern matching, new simplification rules can be described, non-standard transformations can be made, and algorithms extended.

This paper describes pattern matching facilities designed and implemented by the author for MACSYMA, a system for algebraic manipulation currently being developed at MIT's Project MAC. Comparisons with other systems with regard to both implementation and strategy are included, as are many examples.

Patterns can be considered lexical entities, as in SNOBOL [3]. Inside an algebraic manipulation system, arbitrary strings of characters, e.g. /A+)(-X*, are rarely useful. The input-line editor of MACSYMA and the parser's lexical routines are the only portions of the system concerned with more-or-less arbitrary strings of characters.

Patterns can be considered syntactic entities, as in FAMOUS [5] or AMBIT/S [2]. Although syntactic correctness is necessary, it is not sufficient for algebraic expressions to be meaningful. For example, 0**0 (using FORTRAN notation) is syntactically correct, but semantically unclear. A syntactic pattern for "quadratic in x" would match expressions of the form a*x**2 + b*x + c, but might fail to match the expressions x**2 and (x + 1)*(x +6), which are, however, quadratic functions of x.

Patterns can be considered semantic entities, given a suitable context. We will be concerned primarily with the context and semantics of algebraic expressions. A semantic pattern for "quadratic in x" should match 3*x**2 + 4 or (x + 1)*(x + 6), but should not match a*x**2 + b*x + sin(x), which is not a quadratic function of x.

Slagle's SAINT [13] and Moses' SIN [11] demonstrated one important application of semantic pattern matching: large classes of expressions were mapped into forms with known integrals. Other applications, some of which are detailed below, range from adding new operations and simplifications to an algebraic manipulation system, to recognizing and solving special cases of differential equations.

The facilities used for pattern matching by Slagle and Moses were not user-oriented. By contrast, the programs described here give the MACSYMA user a powerful and sophisticated semantic matching capability, and the tools by which he can introduce these capabilities into the command level of the system and into his own programs. Of the other algebraic manipulation systems currently in use, it appears that only Hearn's REDUCE [8] has a user-level matching facility. REDUCE gives the user (through the LET command) a limited syntactic matching facility which is considerably restricted in its generality by its emphasis on efficiency. For example, patterns which are sums are not permitted. FAMOUS [5] and Formula Algol [12], neither of which is currently in use, provided matching facilities, although (as we shall see in section 8), they were largely syntactic, rather than semantic in approach.

In sections 1 to 4, methods for defining patterns in MACSYMA are described, largely through examples. Section 5 discusses MACSYMA's Markov algorithm-style (pattern-replacement) programming facility. Section 6 considers the problem of introducing new simplification rules into MACSYMA efficiently and effectively. Section 7 demonstrates how these techniques can be used to introduce rules for

non-commutative multiplication. Section 8 critically examines the pattern-matching facilities of SCHATCHEN, REDUCE, FAMOUS, and Formula Algol, and compares them to MACSYMA's facility. Questions of strategy and implementation are considered. Section 9 considers applications of pattern matching to solving differential equations. These sections are supplemented by an appendix (Appendix I) with precise, extended definitions of the matching procedures. Appendix II includes an example of a match program as compiled by the system. A theoretical discussion of the problem of defining classes of expressions over which these matching procedures can be considered effective can be found in [4].

1. Predicates and Declarations

An intuitive pattern for a quadratic in x is A*x**2 + B*x + C where A, B, and C are pattern variables which can match numbers or other expressions free of the variable x. In addition, A must not match zero, otherwise linear expressions would be included in the domain of the pattern.

Clearly we must be able to insist that variables in a pattern have certain characteristics (e.g. are nonzero or are free of x); that is we must be able to make the success of a match dependent on the matched values satisfying predicates. Predicates (for our purposes) are programs which return either TRUE or FALSE. In practice, we consider anything other than FALSE as TRUE. Patterns themselves are predicates since they return FALSE if applied to a non-matching expression. Predicates can take any number of arguments (usually at least one) and can be defined in LISP, (in which MACSYMA itself is written) or in the MACSYMA programming language, which resembles Algol 60.

FREEOF(X,Y) is a predicate with two arguments, X and Y, which answers the question, "Does the expression Y depend explicitly on the variable X?" Thus FREEOF(A,A**2+B) is FALSE; FREEOF(A,C+SIN(D)) is TRUE. TRUE(X) is a predicate which is always TRUE. This is useful because it is convenient to allow some variables to match anything. INTEGER(X) is TRUE when X is an integer.

FREEOF, TRUE, and INTEGER are already defined in the standard MACSYMA system. We might define NONZERO by the program:

```
NONZERO(X):= IF X=0 THEN FALSE ELSE TRUE@.
```

The function SIGNUM(X) returns -1, 0 or +1 respectively if X < 0, X = 0, or X > 0. SIGNUM, we should note, expands its argument using MACSYMA's rational function routines [10]. This produces a form which is canonical over rational functions (up to the order of the variables) and allows us to uniquely determine a sign for the coefficient of the highest power of the main variable (in the numerator). Thus it knows that the following expressions are negative: -4, -X, -X - Y, -(1 + X). Whether X - Y is negative or not depends on which variable (X or Y) the rational function package has been told is the main variable. It will choose a main variable itself if necessary. The only expression whose SIGNUM is 0 is 0. Using SIGNUM we can define:

```
NEGATIVEPRED(X):= IF SIGNUM(X)=-1 THEN TRUE
ELSE FALSE@.
```

A few more predicates which are used in examples to follow are:

```
INRANGE(LOW,HI,VAR) := IF (LOW < VAR) AND (VAR
< HI) THEN TRUE ELSE FALSE@

NONZEROANDFREEOF(X,Y) := IF NONZERO(Y) THEN
FREEOF(X,Y) ELSE FALSE@.
```

To associate a pattern variable with a predicate, we have the DECLARE command. It has the form:

DECLARE(name,predicate(arg1, ... , argn))@. ($n \geq 0$)

For example,

```
DECLARE(A,FREEOF(X))@
DECLARE(A,INRANGE(N,M))@
DECLARE(A,TRUE)@
```

Note that the last argument of each predicate is missing from the declaration. The value matching the declared variable will serve as the final actual argument. Thus if A were declared NONZERO and an attempt were made to match A with X**2 + 3, then NONZERO(X**2 + 3) would be evaluated. Since the result would be TRUE, the match would be successful, and A would be assigned the value X**2 + 3.

The binding times of the arguments to DECLARE must be clarified. The first argument is not evaluated; thus DECLARE(A,..) affects the declaration of A, even if the value of A is B + 2. The second (predicate) argument to DECLARE is treated as an undefined function: if we were to change the definition of INRANGE to some other function of three arguments, it would not be necessary to redeclare A. The extra arguments to the predicate (arg1, ..., argn) are bound at the time the predicate is applied. Thus if A were declared to be FREEOF(X), and the value of X at some later time were Z, an attempt to match A current with that assignment would invoke a test to see if the potential match for A were dependent on Z.

2. Match Definitions

The DEFMATCH command defines a new program (a predicate) which will succeed only if a particular semantic pattern is matched. The DEFMATCH command has the form:

DEFMATCH(programname, pattern, patternvar1, ..., patternvarn)@. ($n \geq 0$)

For example,

```
DEFMATCH(LINEAR, A*X + B , X)@
DEFMATCH(F3, X+ 3 + F(X,Y,5), Y)@
DEFMATCH(COSSIMP, COS(N*PI) )@
```

These examples will have different interpretations depending on the declarations (or lack of declarations) for A,B,X,N, and F. The result in each case will be a program with name programname (e.g. LINEAR, F3, COSSIMP) which will test to see if the pattern pattern (i.e. A*X + B, etc.) can be applied to its first argument. The program will have n

additional arguments, corresponding to the patternvars.

During the execution of these resulting programs, undeclared variables (i.e., those variables not appearing as the first argument in a DECLARE command) in the pattern are lambda-bound to the values in the program invocation if their names are among those variables listed in the DEFMATCH command. Variables not listed among the patternvari's are bound to their values in the environment at execution time. At the successful conclusion of a match, declared variables will be assigned the values that they match, and a list of the associations of variables and their values is returned.

An extended example should clarify this. The lines labelled Ci are typed by the user, the lines labelled Di are typed by the computer. Lines terminated by a $ suppress printing of the result. Lines terminated by an @ result in a computer generated display of the answer.

```
(C1) DECLARE(A,NONZEROANDFREEOF(X))$
(C2) DECLARE(B,FREEOF(X))$
(C3) DEFMATCH(LINEAR,A*X+B,X)@
(D3)      LINEAR

(C4) LINEAR(3*Y+4,Y)@
(D4)   (B = 4,A = 3,X = Y)

(C5) LINEAR(Z*Y+4+X,Y)@
(D5)  (B = X + 4,A = Z, X = Y)
```

At this point the value of A is Z, the value of B is X + 4. If the value of X previous to line C5 had been 4, the answer would have been (B = 8,A = Z, X = Y).

The X on line D4 is a completely separate entity from the X on line C5, in that the first is like a formal parameter to a subroutine, and the latter is a global variable with the same name. This distinction should be apparent on line D5.

The patternvar's may appear in the declarations also. Thus:

```
(C6) DECLARE(A,INRANGE(N,M))$
(C7) DEFMATCH(BETWEEN,A,N,M)@
A
IS THE PATTERN
(C8) BETWEEN(5,1,6)@
(D8)  (A = 5,N = 1,M = 6)
```

The message following line C7 is from the DEFMATCH compiler, indicating that it had evaluated A to see if perhaps A's value was the intended pattern. In this case, the value of A was A, thus the message, "A IS THE PATTERN" is printed. The pattern in the DEFMATCH command is generally not evaluated, since this (with its substitution of values for variables) tends to make patterns disappear. However, if (as in this example) the pattern is an "atom," or single variable, then it is evaluated. This allows a user to compose an elaborate pattern, say as a result of a computation, and then give its name to the DEFMATCH command, rather than having to type it in all at once. If A had had the value B + 4, the message "B + 4 IS THE PATTERN" would have been printed.

Now that we have shown how pattern programs are defined, we can clarify the use of the predicate TRUE. Recall that declaring A to be TRUE means that A in a pattern will match anything occupying the appropriate position in the expression. Thus

```
(C9) DECLARE(A,TRUE)$
(C10) DECLARE(B,TRUE)$
(C11) DEFMATCH(G,A*X+B*Y)$
(C12) G(3*X+I*Y+J*X)@
(D12)      (B = I,A = J + 3)
```

We can now enunciate another principle in matching patterns. If A is undeclared and not a pattern variable, A in a pattern will match only A's current value. (If A has no value, then MACSYMA provides "A" for the value of A. As a special case, constants match only themselves.)

3. Selectors

Sometimes it is not sufficient to find out whether or not a predicate succeeds on a given argument. Sometimes we wish to not only test, but separate components of a pattern which in ordinary circumstances would remain indivisible. We wish to permit a special form of predicate which (1) confirms that a subexpression satisfies a predicate, and then (2) hands back to the pattern program more information than just "the predicate succeeded." We will call such programs, when used in the place of predicates, selectors. The selectors that are of the greatest interest to us here always "succeed" in one form or another, but in so doing, return a particular part of the expression which is being matched. Aiding us in this venture is the convention that any result which is not "FALSE" is true.

Consider the predicate INTEGER. It returns TRUE when applied to an integer. A corresponding predefined "selector" WHOLE returns only the integer part of a number. Another selector, FRACTIONPART, might be defined:

```
FRACTIONPART(X) := X - WHOLE(X)$
```

It would then have to be designated a selector by:

```
SELECTOR(FRACTIONPART)$.
```

A dialogue would look like this:

```
(C1) FRACTIONPART(X) := X - WHOLE(X)$
(C2) SELECTOR(FRACTIONPART)$
(C3) DECLARE(A,WHOLE)$
(C4) DECLARE(B,FRACTIONPART)$
(C5) DEFMATCH(SEPARATE, A + B)$
B
MATCHES ALL IN
B + A
(C6)SEPARATE(5/2)@
                          1
(D6)          (A = 2,B = -)
                          2
```

The message following line C5 would normally indicate an error. Here it signifies that B's predicate (or selector) will be applied to what is left after A's predicate (or selector) is applied. Here, this is what is intended, but note that if both A and B had only predicates, SEPARATE would match one of them to 0 in every case. The following caution should be observed: if a selector is used, a complementary selector should generally be used with it, since, for example,

```
(C7) DEFMATCH(F3,A)$
A
IS THE PATTERN
(C8) F3(5/2)@
(D8)     (A = 2)
```

results. The "fractionpart" has (perhaps unintentionally) been discarded.

Another selector provided by MACSYMA is NUMFACTOR, which selects the numerical factor from a product (or 1, otherwise). A complementary selector, OTHERFACTOR might be defined by

```
OTHERFACTOR(X) := X/NUMFACTOR(X)$
```

Other selectors provide facilities for picking out items in a sum or product one by one. The notion of "extractor" in Formula Algol is weaker than this, in that extractors can only be used to attach labels to syntactically distinguishable subexpressions. Thus the numerator of a fraction can be labelled through "extraction" but the "whole part" of a ratio of two numbers cannot be labelled through Formula Algol.

4. More Match Details

Patterns can be more complicated. For example, with A and B declared TRUE, the pattern 3**A + B**4 will match

```
w**4 + 3**z          with A = z , B = w
w**4 + 1             with A = 0 , B = w
3**z                 with A = z , B = 0
3                    with A = 1 , B = 0
1                    with A = 0 , B = 0.
```

The expression 10, (which is 3**2 + 1**4) will not match. The exact limitations of the exponentiation treatment are described in Appendix I.

Any pattern, or part of a pattern, P which is entirely free of variables which are declared and as yet unmatched will match any expression E such that (when all free variables are given their assigned values) E - P = 0. To some extent this type of match depends on what algorithm is used to simplify the result of the subtraction. Ordinarily the MACSYMA simplifier is used, but rational simplification [10] is used when coefficients are being picked off, since expansion is often needed to produce proper results. We feel this is very important if we are to abide by our belief that the semantics of the expression, rather than the syntax, is the important aspect to model in pattern matching. Thus the following dialogue is possible:

```
(C1) DECLARE(A,NONZEROANDFREEOF(X))$
(C2) DECLARE(B,FREEOF(X))$
(C3) DECLARE(C,FREEOF(X))$
(C4) DEFMATCH(QUAD,A*X**2 + B*X + C , X)$
(C5) QUAD((Z+1)*(Z+2),Z)@
(D5)      (C = 2,B = 3,A = 1,X = Z)
```

Rational simplification must be used to compute (Z+1)*(Z+2) - (Z**2+3*Z+2), to convince QUAD that the match has succeeded. This is the only effective method at our disposal, if we wish to implement such matches as C5. The additional rational simplification is not particularly inefficient, since the coefficient routines [10] used have already converted the expression to a canonical rational form.

DEFMATCH has produced in QUAD a program which operates as follows. QUAD(E,X)

a. picks out the coefficient of X**2 in E, and if the coefficient is free of X and nonzero, assigns it to A, otherwise returns FALSE.
b. Sets E to E - A*X**2
c. Picks out the coefficient of X in E, and if the coefficient is free of X, assigns it to B, otherwise returns FALSE.
d. Sets E to E - B*X
e. If E is free of X, assigns E to C and returns a list of the values A, B, and C, otherwise returns FALSE.

Implicit in this algorithm are several basic principles of semantic pattern matching. For example, line (C5) above demonstrates that coefficients in an expression should be extracted effectively.

```
(C6) QUAD(3*X**2+4,X)@
(D6)      (C = 3,B = 0,A = 3)
```

Line (C6) demonstrates that summands in the pattern which are missing in the expression are matched with 0. This is what happened to the term B*X in the QUAD pattern. Furthermore, if a product is matched with 0, one of its factors must match 0. Thus for B*X to match 0, B must match 0.

```
(C7) QUAD(X**2+3*X+4,X)@
(D7)      (C = 4,B = 3,A = 1)
```

That is, factors in the pattern which are missing in the expression are matched with 1. This assigns to A the value 1.

Since DEFMATCH actually produces short programs (e.g. QUAD), the matching programs may be compiled by a LISP compiler into machine code for increased speed. The program, QUAD, produced above, is shown in Appendix II.

To help prevent the user from asking for ambiguous matches (where they can be detected), the match compiler used by DEFMATCH has a number of warning messages. Generally they indicate points where there is a likelihood that the user has submitted a pattern which is ambiguous, or could be more suitably constructed for optimal matching. In general, patterns should be expanded so that the full freedom of commutative operators can be exploited. The pattern x**2-y**2 will match a wider range of expressions than the pattern (x+y)*(x-y). The latter will match only expressions which are the product of two sums of the specific syntactic form used. This asymmetry with respect to patterns and expressions (the expressions x**2-y**2 and (x+y)*(x-y) will be treated identically by most pattern programs) is a consequence of the fact that it is far easier to multiply out sums and pick out coefficients, than it is to factor polynomials. We allow either pattern however, since it is possible that the latter, strictly syntactic match (like those available in Formula Algol or FAMOUS) might be of some use anyway.

Since backing up (i.e., abandoning assignments of values and trying new ones) is not done in the matching process, the user should consider whether his intentions will be

properly represented. While a back-up algorithm could have been adopted, the tremendous cost increase, combined with no assurance that the user would be happy anyway, make such an approach unattractive. There is the further argument that pattern-match problems can be easily constructed which are undecidable (in the Turing-Church sense), so back-up will not solve all our problems. SCHATCHEN uses back-up; back-up is expensive, and as is demonstrated by the examples in this paper, the lack of back-up is not generally noticed. This is discussed further in section 8.

An example which demonstrates how backing-up might be implied by a pattern follows:

```
(C1) DECLARE(A,TRUE)$
(C2) DECLARE(B,FREEOF(Y))$
(C3) DEFMATCH(NEEDBACKUP, SIN(A)+SIN(B))$
(C4) NEEDBACKUP(SIN(X)+SIN(Y))$
```

The final line may match with (A = Y, B = X); but, if A = X is tried first (succeeding), and then B = Y is attempted, the pattern will fail.

One method of circumventing this difficulty is as follows: (RETLIST returns its argument list as a sequence of equations, ":" is the assignment operator, and DUMMY is used as an indicator that the next list consists of local (i.e., "dummy") variables.)

```
(C1) DECLARE(A,TRUE)$
(C2) DECLARE(B,TRUE)$
(C3) DEFMATCH(PAT,SIN(A)+SIN(B))$
(C4) DOESBACKUP(Z):=IF PAT(Z)=FALSE THEN FALSE
ELSE IF FREEOF(Y,B) THEN RETLIST(A,B)
                    ELSE (DUMMY (TEMP),
                          TEMP:A,
                          A:B,
                          B:TEMP,
                          RETLIST(A,B))$
```

The purpose of the fancy ELSE clause in C4 is to reverse the assignment of values to A and B in the returned list. Thus, while a conscious design decision was made to prevent back-up, the possibility of simulating it, when necessary, is available.

Arbitrary n-ary functions may be used in a pattern, as is illustrated below:

```
(C1) DECLARE(F,TRUE)$
(C2) DECLARE(X,TRUE)$
(C3) DECLARE(Y,TRUE)$
(C4) DEFMATCH(F2,F(X,Y))$
(C5) F2(POINT(3,4))@
(D5)      (Y = 4,X = 3,F = POINT)
```

It is also possible to execute

```
(C6) F2(W+4)@
(D6)       (Y = W,X = 4,F = MPLUS)
```

This gives a facility for explicitly matching operators, if, for example, F is declared to match only MPLUS. This facility could be used to simulate simpler styles of pattern matching which are completely syntax based.

5. Markov Algorithms

Users of a mathematical laboratory may find that certain algorithms lend themselves to an organization based on the Markov algorithm formalism: a list of rules, each consisting of a pattern-replacement pair is applied to an expression. FAMOUS [5], PANON-IB [1], AMBIT/S [2], Formula Algol [12], and SNOBOL [3], among others, are based on such a formalism. In order to allow MACSYMA algorithms to be written in such a style, a command to define rules, DEFRULE, is provided, along with sequencing algorithms. The form of the DEFRULE command is:

DEFRULE(rulename,pattern,replacement)@

If the rule named rulename is applied to an expression (by one of the APPLY programs below), every subexpression matching the pattern will be replaced by the replacement. All variables in the replacement which have been assigned values by the pattern match are assigned those values in the replacement which is then simplified. The rules themselves can be treated as programs which will transform an expression by one operation of pattern-match and replacement. If the pattern fails, the value of the rule is NIL, displayed as ().

Applying Rules

Each of the programs described in this section applies its rules to the expression indicated by its first argument, recursively on that expression and its subexpressions, from the top down.

APPLY1(e,r1, r2,...,rn) applies the first rule, r1, to the expression e until it fails, and then recursively applies the same rule to the subexpressions of that expression, left-to-right, until the first rule has failed on all subexpressions. Then the second rule is applied in the same fashion. When the final rule fails on the final subexpression, the application is finished.

APPLY2(e, r1,r2,...,rn) differs from APPLY1 in that if the first rule, r1 fails on a given subexpression, then the second is applied, etc. Only if they all fail on a given subexpression is the whole set of rules applied to the next subexpression. If one of the rules succeeds, then the same subexpression is reprocessed, starting with the first rule.

APPLY1 corresponds to Formula Algol's [9,12] one-by-one sequencing mode, and APPLY2 corresponds to its parallel sequencing mode (with the inessential difference that Formula Algol processes from right to left).

Thus if R1, R2, R3, and R4 are rules defined by DEFRULE, a program might be written using them as follows:

```
PROGRAM(X):=APPLY1(APPLY2(X,R3,R4),R1,R2)$
```

and the Markov-style algorithm represented by PROGRAM could be executed on the expression Y by

```
Z:PROGRAM(Y)@
```

Here is an example of using rules to alter an expression. The symbol S is used as an abbreviation for e**z, RATSIMP [10] expands an expression into a ratio of polynomials and

cancels common factors, and the symbol % always denotes the most recently displayed expression.

```
(C1) DEFRULE(R1,SECH(Z),1/COSH(Z))$
(C2) DEFRULE(R2,TANH(Z),SINH(Z)/COSH(Z))$
(C3) DEFRULE(R3,SINH(Z),(S-1/S)/2)$
(C4) DEFRULE(R4,COSH(Z),(S+1/S)/2)$
(C5) SECH(Z)**2+TANH(Z)**2@
```

```
                    2          2
(D5)          TANH(Z)  + SECH(Z)
```

```
(C6) APPLY1(%,R1,R2,R3,R4)@
```

```
                                    2
                                   1
                              (S - -)
                 4                 S
(D6)          -------- + --------
                     2              2
                  1              1
              (S + -)        (S + -)
                 S              S
```

```
(C7) RATSIMP(%)@
(D7)                    1
```

6. Advising the Simplifier

When the user of a system like MACSYMA introduces new functions, or uses old functions in a way that is unfamiliar to the system, he may find himself battling certain "built-in" aspects of MACSYMA.

On one hand, he may find that the SIMPLIFY program does not simplify expressions the way he wants it to. While he can work at odds with the simplifier to some extent by using Markov-style algorithms on his data, the global and all-pervasive influence of the simplifier must sometimes be modified. Although the user could just turn off the simplifier, this solution is probably not very useful. The chances are that he still wants the simplifier to work on most of the expression under consideration, but not on some particular part in some particular fashion.

On the other hand, he may find that the SIMPLIFY program is just ignorant of functions of interest to him. For example, a user may wish to see SINH(0) replaced by 0 whenever it occurs, especially if it occurs inside a calculation.

Another useful possibility is the one typified by telling the simplifier that X**N is 0 for N greater than some number M. This, in effect, allows one to truncate while doing arithmetic on power series.

For these reasons, an advising facility, similar in certain respects to Teitelman's ADVISE [14] has been implemented. There are two commands to advise the simplifier: TELLSIMP, and TELLSIMPAFTER. They have the following forms:

TELLSIMP(pattern, replacement)@

TELLSIMPAFTER(pattern, replacement)@

The arguments are similar to those of DEFRULE, but the pattern must conform to certain restrictions described below.

TELLSIMP analyzes the pattern, and if it is either a sum, a product, or an atom (i.e. a single variable name or a number) it will complain. Sums and products are excluded by TELLSIMP because of the interdependence of the simplifier and the matching programs in this implementation. TELLSIMPAFTER, discussed at the end of this section, has no such restriction.

The exception for atomic variables is necessary because the advice is stored on the property list of operators, where SIMPLIFY looks for it. SIMPLIFY does not look on the property list of variables for simplification advice. This restriction, however, is hardly important, since setting a variable to its "simplified" form will give the same effect.

The simplification of sums and products should probably be attacked in ways other than through TELLSIMP or TELLSIMPAFTER. It is simple (but somewhat naive) to suggest that (sin x)**2 + (cos x)**2 ==> 1 be told to the simplifier as TELLSIMP (SIN(X)**2,1-COS(X)**2); what is really needed is a facility that demands the presence of both sines and cosines, and removes them in appropriate circumstances.

All the above rule does is remove sines in favor of cosines, sometimes. TELLSIMPAFTER(SIN(X)**2+COS(X)**2,1), although a legal command, does far less that the user may think. For example, it leaves out the possibility of a third term in the sum (e.g., 5+sin(y)**2+cos(y)**2), it does not back up (e.g., sin(y)**2+cos(2*y)**2+sin(2*y)**2) and it does not detect instances of the pattern implicit in such constructions as sin(y)**4+2*sin(y)**2*cos(y)**2+cos(y)**4. While patterns may be constructed for some of these expressions, it is our opinion that such substitutions as sin(x)**2+cos(x)**2 ==> 1 require much stronger methods than pattern matching. Methods for doing such simplifications effectively are available in the rational substitution facility of MACSYMA [10]. In it the approach used by REDUCE to handle products [8, p. 8], is implemented, but is extended to deal with sums also.

TELLSIMP piles new advice on top of old advice, but old advice is still accessible if the new advice is not appropriate (i.e. the pattern fails). This is exhibited in the following example.

```
(C1) COS(PI)@
(D1)                    COS(PI)

(C2) TELLSIMP(COS(PI),-1)@
-1
IS THE REPLACEMENT
(D2)                      COS

(C3) COS(PI)@
(D3)                      - 1

(C4) COS(-PI)@
(D4)                  COS( - PI)

(C5) MPRED(X):=IF (SIGNUM(X)=-1)THEN TRUE ELSE
 FALSE$
(C6) DECLARE(M,MPRED)$
(C7) TELLSIMP(COS(M),COS(-M))$

(C8) COS(-PI)@
(D8)                      - 1

(C9) COS(5*PI)@
(D9)                  COS(5 PI)
```

```
(C10) DECLARE(N,INTEGER)$
(C11) TELLSIMP(COS(N*PI), (-1)**N)$
(C12) COS(5*PI)@
(D12)                    - 1

(C13) COS(-6)@
(D13)                  COS(6)
```

The dialogue above shows (D1) that the simplifier (at that time) did not know the rules about pi (=3.1415+). If we tell it that the cosine of pi is -1, it can (D3) simplify COS(PI) to -1. Line (D4) demonstrates that the simplifier did not know about cosine being symmetric about 0. Lines (C5)-(C7) add this bit of information, as evidenced by line (D8). Line (C11), which makes superfluous the advice of (C2), but not of (C7), adds the capabilities shown in (D12). (C13) shows that the old advice is still accessible.

One of these rules happens to coincide with a "built-in" simplification COS(0) = 1, since N*PI for N=0 matches 0; however, since the answer will be (-1)**0, the ordinary operation of the simplifier underneath will not be affected. (System-defined simplifications will be tried, but only if none of the advice is applicable. Note that if any of the advice is applicable, the replacement part of the advice will have already triggered a further simplication, if such is possible.)

TELLSIMPAFTER is similar to TELLSIMP except that new rules are placed after old rules and "built-in" simplifications. Because of this, TELLSIMPAFTER cannot be used to drastically alter the action of the simplifer, whose "built-in" simplifications take precedence. On the other hand, these restrictions make it possible to apply TELLSIMPAFTER to sums and products.

TELLSIMPAFTER should be used on "built-in" operators whenever possible, since such rules will be applied only if the same operator is still the lead operator after the previous simplification has been performed. If the lead operator has been changed, all "after" rules are bypassed, producing faster operation.

7. Non-Commutative Multiplication

At this time,non-commutative multiplication simplification is not available in MACSYMA. This section describes how the author added such a fairly extensive facility by using the TELLSIMP commands. The group operation, represented by a period (.), is allowed by the parser in anticipation of the time when an efficient non-commutative multiplcation scheme is programmed in LISP. (Since the same symbol is used to denote the decimal point of a floating point number, extra parentheses may sometimes be required to avoid misinterpretation.)

Telling the simplifier about non-commutative multiplication requires a bit of knowledge of the internal representation. The input A.B is parsed to ((MCTIMES) $A $B), that is, a prefix representation (although with certain peculiarities of no importance to this discussion). The fact that MCTIMES is a binary operator rather than a "vari-ary" operator will complicate matters somewhat. We will abbreviate ((MCTIMES) $A $B) as (. A B).

The input A.B.C or (A.B).C is parsed to (. (. A B) C), but A.(B.C) is parsed to (. A(. B C)). Clearly one of the first jobs of the "MCTIMES" simplifier is to transform the second structure into the first. To do this (in effect, telling the simplifier about the associative law), we

```
DECLARE(A,TRUE)$
DECLARE(B,TRUE)$
DECLARE(C,TRUE)$
TELLSIMP(A.(B.C),(A.B).C)$
```

As an example of how this operates, consider (A.B).(C.D). This is parsed to (. (. A B)(. C D)) which is then simplified to (.(.(. A B) C) D). Since the simplifier is recursive, any depth of forced nesting is untangled. Any time two identical elements are adjacent, we want to combine them. That is, A.A = A**2; more generally, (A**n).(A**m) = A**(n+m). Since our pattern matcher is clever enough to recognize A as an occurrence of A**1, this one pattern would suffice, but for one difficulty: although A.A is parsed to (. A A), B.A.A is parsed to (. (. B A) A). These two situations differ sufficiently with respect to adjacency of the A's so as to require the two patterns below.

```
DECLARE(N,TRUE)$
DECLARE(M,TRUE)$
TELLSIMP((A**M).(A**N),A**(M+N))$
TELLSIMP(B.(A**M).(A**N),B.A**(M+N))$
```

Let us denote the inverse of A by INV(A), and the identity by 1. We might then

```
TELLSIMP(INV(1),1)$
TELLSIMP(INV(INV(A)),A)$
TELLSIMP(INV(A.B),INV(B).INV(A))$
```

Recall that these pieces of advice are placed on the property list of the function INV, and so are independent of the previous bits of advice, which are on the property list of ".".

Another piece of advice which will be needed goes on the property list of "**" -- this time, after other simplifications have been made:

```
TELLSIMPAFTER(INV(A)**N,INV(A**N))$
```

The major fact concerning inverses is their "cancellation" property. That is, A.INV(A) = INV(A).A = 1. To automate this, let us consider the more general situation, (A**n).INV(A**m) = A**j * INV(A**k) where at least one of j or k is 0.

Let us define MONUS(N,M), which will compute j and k:

```
MONUS(N,M):= IF N>M THEN N-M ELSE 0$
```

and INVPROG(A,N,M) which will compute the right hand side of the above reduction formula.

```
INVPROG(A,N,M):=
A**MONUS(N,M)*INV(A**MONUS(M,N))$
```

Thus:

```
TELLSIMP((A**N).INV(A**M),INVPROG(A,N,M))$
TELLSIMP(INV(A**M).(A**N),INVPROG(A,N,M))$
TELLSIMP(B.(A**N).INV(A**M),B.INVPROG(A,N,M))$
TELLSIMP(B.INV(A**M).(A**N),B.INVPROG(A,N,M))$
```

Finally,

```
DECLARE(N,INTEGER)$
TELLSIMP(N.A,N*A)$
TELLSIMP(A.N,N*A)$
```

gives us such useful notions as left and right zeros, identities, and multiplication by scalars. It may appear that we have left out some items, for example,

```
TELLSIMP(A**0,1)$
TELLSIMP(INV(A)**0,1)$
TELLSIMP(1.A,A)$
```

but this is not so. Since 1.A will be converted to 1*A, which will be simplified to A, the last rule is unnecessary. Since A**0 will (unless we tell the simplifier otherwise) always result in 1, the other two are also unneeded.

As examples of how this new simplifier operates, X.INV(X)**2 is simplified to INV(X), and A.B.(B**3).C.INV(C) is simplified to A.B**4. This last example used about .7 seconds of machine time when the simplification rules were in uncompiled LISP (on a PDP-10 computer using 2.75 microsecond cycle time memory), and when compiled by the LISP compiler, about .05 sec.

8. Comparisons with SCHATCHEN, FAMOUS, REDUCE, and Formula Algol

SCHATCHEN, Moses' matching program [11] is similar to our matching program in many respects. However, there are significant differences, both in implementation and in philosophy, between the two systems.

SCHATCHEN, demands patterns in a form resembling the internal form for expressions. It uses controls (called modes) on the pattern match to direct its highly recursive matching processes. Our "straight-line" matching programs preserve some, but not all, of the aspects of the mode facility.

A SCHATCHEN pattern corresponding to the intuitive notion of "quadratic in x" discussed in section 4 is:

```
(QUOTE
 (PLUS
  (COEFFPT
   (A
    (FUNCTION
     (LAMBDA (Y) (AND (FREE Y (QUOTE X))
                      (NOT (EQUAL Y 0))))))
   (EXPT X 2))
  (COEFFPT
   (B (FUNCTION (LAMBDA (Y)
                        (FREE Y (QUOTE X)))))
   X)
  (COEFFP
   (C
    (FUNCTION (LAMBDA (Y)
                      (FREE Y
                            (QUOTE X))))))))
```

This is not in the best possible form for SCHATCHEN, but it serves to illustrate several points. First, the pattern is written as a LISP S-expression which, upon close examination, has most of the components of a prefix representation of the algebraic expression A*X**2+B*X+C. Second, there are a number of extra notations in the pattern, some of which clearly depend on LISP's version of the lambda-calculus. A less obvious point is that the pattern implies an ordering on the subtasks required to match it to an expression.

There are two modes, COEFFPT and COEFFP, used in this pattern. They stand for "coefficient in plus and times" and "coefficient in plus" respectively, and their uses are best described through an example.

Consider the quadratic, Q = 2*X**2 + Y*X**2 + 3 + Z. There are two terms involving X**2. For the pattern A*X**2 + B*X + C to match Q, A must match 2 + Y. This is indicated to SCHATCHEN by using the mode COEFFPT. This modifies the action taken to match A by causing SCHATCHEN to traverse Q looking for coefficients of X**2 and assigning to A the simplified sum of those coefficients. Similarly, by matching B with mode COEFFPT, B is assigned the simplified sum of the coefficients of X (or is assigned zero if there are no coefficients, as is the case for Q).

SCHATCHEN requires that C in the quadratic pattern be matched using the mode COEFFP (that is, "coefficient in plus") so that in Q, C will match Z + 3, and not just one term (e.g. Z or 3). Since A*X**2 and B*X have been previously deleted from the expression by the matching procedure, C (by virtue of its being indicated a COEFFP) will match what is left in the sum, namely Z + 3.

SCHATCHEN also provides opportunities to apply predicates to A, B, and C; in this case they each are checked to make sure they are free of X. A is also checked to make sure it is nonzero.

Compared to the relatively casual definition of QUADRATIC in section 4, using these controls requires a high level of awareness on the part of the user, both of the representation of data, and the operation of SCHATCHEN. This burden of awareness is considerable. However, SCHATCHEN matches differ from the matches done here in a more fundamental sense. We find a particular subexpression and apply a predicate. If the predicate fails, the match fails. In a similar situation, SCHATCHEN will try to find another subexpression which matches the subpattern, which might satisfy the predicate. The match fails only if this exhaustive search fails to find any subexpression matching (and satisfying) the subpattern.

This difference, which would seem to indicate that SCHATCHEN is more powerful, is somewhat deceptive. We use more powerful tools to find an appropriate place to apply a predicate, and then apply it only once. (The coefficent-finding routine we use can find that the coefficient in (2*x)*(3*x+1) of x**2 is 6; SCHATCHEN would fail to notice this.) There is an increase in efficiency since the programs produced by the match compiler are "straight-line" code, and apply predicates (assuming success) only as many times as there are distinct variables in the pattern. In case the pattern fails, fewer predicates are applied. The number of times SCHATCHEN applies its predicates is much more dependent on the expression. While SCHATCHEN has certain types of iterative facilities within a single pattern, the programming language facility in MACSYMA can supply some of the same iterative machinery, as in section 5.

There are some instances where SCHATCHEN is undeniably more thorough (within the scope of a single pattern): if the pattern is A**B and the expression is 1, either B matching 0 or (B's predicate failing) A matching 1 will

cause the pattern to succeed. We insist that A match 1 and B match 0.

TELLSIMP gives essentially all the power of FAMOUS for flexibly altering an algebraic simplifier, yet allows one to have a quite competent "fall-back" facility. While using TELLSIMP excessively on commonly used operators might make the system run as slowly as did FAMOUS, it is unlikely that that point will be reached either frequently or quickly. Using TELLSIMP on new functions (e.g. SINH) does not affect the speed of the simplifier on old functions. The technique of compiling rules achieves a modest level of efficiency; using the LISP compiler further speeds up processing. Of course, advice requiring much computation (e.g., replace INV(A) where A is a square matrix, by its computed inverse) will slow up the simplifier in direct proportion to the length of the computation, and how often it is done. Easy advice, in this user's experience, has not caused a noticeable change in system response. More precise measurements can be made, of course, but very little unnecessary system degradation is introduced by the particular techniques used. (Some timing data appeared at the end of section 7) Furthermore, the TELLSIMPAFTER facility, potentially far more efficient than a last-in first-out rule organization, is available.

It is clear that flexible pattern matching results in an enormous decrease in the number of rules required to achieve a given match. Consider the rules that would be required to define "quadratic in x" in a purely syntactic manner, as in FAMOUS or Formula Algol:

```
x**2                    a*x**2
x**2 + x                a*x**2 + x
x**2 + b*x              a*x**2 + b*x
x**2 + c                a*x**2 + c
x**2 + x + c            a*x**2 + x + c
x**2 + b*x + c          a*x**2 + b*x + c
```

This also assumes

(1) + and * are commutative with respect to the match;

(2) a, b, and c may be declared free of x;

(3) a, b, and c may each match more than one term;

and (4) the minus sign is not a separate operator.

This is not meant to imply, however, that restricted styles of matching are never appropriate. By using restricted matches, Fenichel was able to justify his contention that arbitrary and precisely specified algorithms could be constructed in FAMOUS. Itturiaga [9] used similar techniques in Formula Algol to produce somewhat more practical results, but the syntactic (rather than semantic) nature of Formula Algol pattern matching prevented the tackling of difficult problems in a natural fashion. FAMOUS and Formula Algol insist that expressions look very nearly like the pattern which is used to match against them. By contrast, our semantic approach can match quadratics which do not resemble any of the above twelve forms.

Dependence on local syntactic transformations, another major thread in FAMOUS, has serious implications relative to efficiency. For example, the ad hoc treatment of "logsum" ([5] page 42) was necessary because local information, in some cases, has to be propagated outside of its immediate vicinity. (The logsum device separated sums into logarithmic terms and non-logarithmic terms. If the sum occurred in an exponent, the log term became a coefficient of the base. Thus e**(x+log(y)) ==> y*e**x. If the sum was not in an exponent, a great deal of time has been wasted.) Waste of this sort is avoided in MACSYMA (and no doubt in other algebraic manipulation systems not tied down to local syntactic transformations) by considering such analyses in a top-down fashion. This provides sufficient global context to distinguish sums occurring in exponents from sums occurring outside exponents.

To the concept of spatial or syntactic adjacency must be added the concept of adjacency along semantic dimensions. For example, if the properties of an exponent are adjacent to its base, then an efficient local "logsum" device might be constructed. In the expression f + g + h, it is clear that f and h should be syntactically just as adjacent as f and g. What is less clear is how one might note that f and g being integer-valued functions makes them adjacent along a semantic dimension.

MACSYMA allows information to be stored at operator nodes in the internal tree representation of expressions (e.g. "this expression and all its subexpressions are simplified") which has some aspects of this semantic dimension. This "property list" of operators has turned out to be an extremely useful design decision, one with applications to many difficult implementation problems. The types of information stored on these nodes will no doubt become more varied as MACSYMA continues to grow.

Another thread in FAMOUS is reliance on the Markov algorithm formalism. It is clear that some algorithms, (e.g. synthetic division of polynomials) are difficult to program in such a formalism. These algorithms benefit not only from a different style of program organization, but also from a radically different data representation. Fenichel, by not modeling any sophisticated polynomial manipulation capabilities, implicitly recognized this limitation.

In summary, FAMOUS and Formula Algol cannot compete with MACSYMA with regard to efficiency or ease of use in algebraic manipulation on several grounds:

(1) the lack of a competent base simplifier (FAMOUS assumes nothing about the characteristics of its data, and cannot assume, therefore, that any particular simplifications would always be valid; Formula Algol has only trivial built-in simplifications.),

(2) the inflexibility of the rules (a consequence of their syntactic, rather than semantic, nature),

(3) inefficient rule-sequencing techniques (they have no equivalent to TELLSIMPAFTER).

FAMOUS has additional problems because of:

(4) its requirement that the Markov algorithm formalism, and data types appropriate to it be used for all manipulations,

(5) the absence of facilities for global communication.

REDUCE has, in addition to objection (2) above, another problem. It considers the user-supplied rules only after it has done its own simplifications. Therefore a rule X**I ==> 0 for all I will not prevent X**0 ==> 1,

the action taken by the simplifier. Furthermore, REDUCE does not allow sums in rules at the top level. REDUCE, although probably more efficient within its domain [6], would require considerable programming to extend it to the realm of non-rational functions, a domain treated routinely here.

Finally, it is not certain that a closer model of SCHATCHEN, including back-up, but (of necessity) closely tied to the internal representation, would greatly aid a user (except perhaps a system programmer), considering the burden it would impose. The benefits of our implementation are clear: we give a user error and warning messages, the selector facility, and easy-to-use methods for declaring variables and defining patterns. For the most part, he can remain ignorant of the subtleties of LISP and the data representation (a sharp contrast with SCHATCHEN), and yet define powerful, flexible patterns.

9. Differential Equations

The following example of a dialogue with MACSYMA illustrates the usefulness of pattern matching in constructing more useful programs. We wish to program the solution of ordinary linear first-order differential equations. i.e.

$$F(X)\left(\frac{DY}{DX}\right) + G(X)*Y + H(X) = 0$$

where F, G, and H are functions of X, but not of Y. The solution can be written in terms of integrals, as demonstrated by the program defined on line C6, below. Note that commas are used instead of semi-colons to separate statements in a program, colons are used as assignment operators, and "DUMMY" is followed by a list of local (i.e., dummy) variables. The commands INTEGRATE and SOLVE are described in [10], but their intent should be clear. Also note that %E represents the base of the natural logarithms, and that D8 is correct, although in a somewhat unusual form.

```
(C1) DECLARE(F,NONZEROANDFREEOF(Y))$
(C2) DECLARE(G,FREEOF(Y))$
(C3) DECLARE(H,FREEOF(Y))$
(C4) P : F*DERIVATIVE(Y,X,1)+G*Y+H$
(C5) DEFMATCH(PAT,P,Y,X)@
```

$$F\left(\frac{DY}{DX}\right) + G\,Y + H$$

```
IS THE PATTERN
(D5)                    PAT

(C6) LINDEP(EQ,Y,X) :=(DUMMY(F,G,H,P,Q,SOL),
IF PAT(EQ,Y,X)=FALSE THEN FALSE
        ELSE
         P : %E**(INTEGRATE(G/F,X)),
         Q : H/F,
         SOL:Y*P+INTEGRATE(Q*P,X),
         EXPAND(SOLVE(SOL=CONST,Y)))$


(C7) DERIVATIVE(Y,X,1)+3*Y+4@
```

$$(D7)\qquad \frac{DY}{DX} + 3\,Y + 4$$

```
(C8) LINDEP(%,Y,X)$
```

$$(D8)\qquad Y = \frac{CONST}{3\,X\,\%E} - \frac{4}{3}$$

The program on line C6 could easily be altered to account for other types of equations. If the PAT pattern fails, other patterns could be tried, each with its own method of solution. If none of the patterns succeed, other analytic or numerical methods could be tried.

10. Conclusions

Although a pattern-directed interpreter (along the lines of SCHATCHEN or FAMOUS) could have been written to implement this algorithm, a compiler, which produces a LISP program from the pattern, was written instead. There are several advantages to this approach:

1. Elaborate checking is done at compile-time to help insure that patterns make sense. An interpreter can provide this only at considerable cost at execution time. This makes interpretation unattractive to a user who needs as much error-checking as possible.
2. When the match compiler is no longer needed, it can be removed from core memory, and the space it occupies, reclaimed. Only the pattern programs themselves are required at execution time. An interpreter must be present any time a pattern is matched. It is possible that a large number of pattern programs could collectively take more space than some other pattern represention, so that this advantage is not clear cut. However, judging from the size of the match compiler, we suspect that an interpreter performing the same tasks is likely to be sufficiently large so as to be more space consuming than perhaps 40 pattern programs.
3. With the exception of calls to the simplifier, the coefficient routines, and calls to subroutines to find exponents, bases, and unknown functions, the program produced by the DEFMATCH (or DEFRULE, TELLSIMP, etc.) command is self-contained. The application of predicates, the assignment of values, and sequencing of operations is rapid and efficient. Furthermore, each pattern program can be compiled into machine language by a LISP compiler, which (on the PDP-10) decreases the bulk of the program and may increase the speed by a factor of ten. It may appear that this possibility is independent of the question of compilation vs. interpretation, since the pattern-directed interpreter could also be compiled into machine code. This is not the point we are making. The patterns for the interpreter cannot be compiled since they are, of necessity, LISP data. On the other hand, the pattern programs of our system can be compiled completely into machine code.

Acknowledgements

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency (ARPA), Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0002, and by Harvard University sponsored by ARPA under Air Force contract F19628-68-0101, and by the National Science Foundation under their Graduate Traineeship program.

I wish to thank Professor J. Moses for his continuing interest, comments, and suggestions, which have guided me in this work. I would also like to thank Professors A. G. Oettinger and B. F. Caviness for their careful reviewing of earlier drafts of this paper.

References*

1. Carraciolo di Forino, A. et al, "PANON-IB -- A Programming Language for Symbol Manipulation," University of Pisa, Italy, 1966.

2. Christensen, C. "Examples of symbol manipulation in the AMBIT programming language," Proc. ACM 20th National Conference, Cleveland, Ohio, 1965. 247-261.

3. Farber, D. et al, SNOBOL, A String Manipulation Language, JACM 11, (1964), 21-30.

4. Fateman, R., "Essays in Algebraic Manipulation," doctoral dissertation, Harvard University, 1971.

5. Fenichel, R. "An On-Line System for Algebraic Manipulation," doctoral dissertation, Harvard University, July 1966, (also appeared as Report MAC-TR-35, Project MAC, MIT, Cambridge, Mass., Dec., 1966).

6. Hearn, A. "REDUCE, a Program for Symbolic Algebraic Computation," invited paper presented at SHARE XXXIV, Denver, Colorado, March, 1970.

7. --. "REDUCE Users' Manual," Stanford Artificial Intelligence Project, Memo 50, Stanford University, Stanford, Calif., Feb., 1967.

8. --. "The Problem of Substitution," Stanford Artificial Intelligence Report, Memo No. AI-70, Stanford University, Stanford Calif., Dec., 1968. (Also appears in Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation, R. Tobey, editor, IBM Boston Programming Center, Cambridge, Mass.,1969, 3-19.)

9. Itturiaga, R. "Contributions to Mechanical Mathematics," doctoral dissertation, Carnegie-Mellon University, Pittsburgh, Pa., April, 1967.

10. Martin, W., and Fateman, R. "The MACSYMA System," these proceedings.

11. Moses, J. "Symbolic Integration," (SIN) doctoral dissertation, MIT, 1967 (also appeared as Report MAC-TR-47, Project MAC, MIT, Cambridge, Mass., Dec. ,1967; now available from the Clearinghouse, document AD-662-666.) Also see, --. "Symbolic Integration - the Stormy Decade," these proceedings.

12. Perlis, A., Itturiaga, R., Standish, T. "A Definition of Formula Algol," a paper presented at the [first] Symposium on Symbolic and Algebraic Manipulation of the ACM, Wash., D.C., March, 1966.

13. Slagle, J. "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus, Symbolic Automatic Integrator (SAINT)," doctoral dissertation, MIT, 1961 (a paper based on this thesis appears in Computers and Thought, McGraw-Hill, New York, 1963.)

14. Teitelman, W. "PILOT: A Step Toward Man-Computer Symbiosis," doctoral dissertation, MIT, Sept., 1966,(Also appeared as MAC-TR-32, Project MAC, MIT, Cambridge, Mass., Sept., 1966, now available from the Clearinghouse, document AD-645-660.)

*Government contractors may obtain MAC-TR reports from the Defense Documentation Center, Cameron Station, Alexandria, Va. 22314. Specify AD number.

Others may obtain the reports from: Federal Clearinghouse, U.S Department of Commerce, Springfield, Va. 22151. Specify AD number. All copies are $3.

Appendix I

Detailed description of the MATCH processor.

Up to this point we have tried to show mainly by examples, what kinds of patterns can be compiled. This section describes the algorithm used to compile patterns into programs, and in so doing, explicates the nature of the semantic matching done by the resulting programs. Some details which are concerned only with "code optimization" are omitted -- as an example, the predicate "TRUE" is never actually called, since the result is known to the match compiler. However, the operation would be unaffected if a call to "TRUE" were actually used.

Definition: An unmatched variable in a pattern is a variable which is declared and for which no value has yet been assigned during this matching process. A variable may be assigned a value either by being in the list of patternvar's, or by being successfully compared to an expression. A pattern p is compared to an expression e by attempting a match between p and e. If the match succeeds, all unmatched variables in p will be assigned values. If the match fails, the value NIL, displayed (), is returned.

Definition: If a pattern p has no unmatched variables in it, it is called a fixed pattern, or is said to be fixed.

Remark: Any number is a fixed pattern. Any undeclared "atomic" name is a fixed pattern. A sum, product, (etc.) of fixed patterns is a fixed pattern.

One of the basic design decisions concerning the internal format of MACSYMA expressions pervades this algorithm. MACSYMA removes inessential operators such as division and negation: A/B is represented internally by A*B**(-1), and -A is represented by (-1)*A. Reducing all arithmetic operators to +, *, and ** has the disadvantage of causing a moderate increase in the size of internal representations, but has the overriding advantage of erasing small differences in appearance which might tend to obscure the matching process. (The MACSYMA input and output routines, in order to improve readability, do make use of quotients, differences, and unary minuses.) Markov algorithms written in Formula Algol seem to be largely concerned with juggling these redundant internal notations, a confirmation of the suitability of our design decision. (see [9] pp. 172-174)

With these preliminaries, we can define precisely what is meant when a pattern p matches an expression e.

I. If a pattern p is fixed, then it matches an expression e if and only if p - e, when simplified and evaluated, is 0. Of the simplification routines in MACSYMA, the general ("advisable") one is usually used. When coefficients have been picked out of an expression in the previous step, canonical rational simplification [10], which expands expressions and combines similar terms, is used. Note the heavy dependence on the power of the simplifier. If the user has (presumably by mistake) told the simplifier to replace an expression A by a larger expression which has A as a subexpression, this definition may become circular. We assume that no such errors have been committed.

II. If p is a sum, Σ ai, then all fixed ai are subtracted from e, and then the rest of the ai are examined as follows:

A. If ai is a product with more than one unmatched variable, it is ambiguous. Any of the variables might match the whole expression. Processing such a pattern will cause a warning to be printed, and the pattern will be treated as in E below, as an occurrence of the specific function "MTIMES" with a fixed number of arguments.

B. If ai is a product of a declared variable v and a fixed pattern f then v's predicate is applied to the coefficient of f in e. (The definition of "coefficient" used here may be found in [10].) If it fails, the match fails, otherwise it proceeds. (That is, v is compared to the coefficient of f in e.) Coefficients are extracted by the RATCOEF [10] program.

C. If ai is an unmatched variable, then it should be the only unmatched ai, since it will match the rest of the expression. If selectors are used, there might be more than one remaining ai, in which case they might correctly separate out the rest of the expression into several parts. A warning is printed in this situation.

D. If ai is an exponentiation, one of three possibilities exists. Either the base is fixed, the exponent is fixed, or neither is fixed. (If both were fixed, ai would be fixed, and thus be treated under I.)

1. The base is fixed: A search is made for an exponential operator with the given base. If it succeeds, the pattern for the exponent is compared to ai's exponent. Here, as elsewhere, if the comparisons of subexpressions fail, the match fails. If the search fails, the base may occur to the first power. If the base is found in e, then the pattern for the exponent is compared to the number 1. If the base is a sum itself, it is subtracted from e, and the pattern for the exponent compared to 1.

2. The exponent is fixed: A search is made for an exponential operator with the given exponent. If it succeeds, the pattern for the base is compared to ai's base. If the search fails and the exponent is a negative integer, 1 is subtracted from e and the pattern for the base is compared with 1 (the case of a missing denominator). Otherwise, (the exponent is not a negative integer) the pattern for the base is compared with 0. This means that the pattern a+1/b (with a and b declared TRUE) will match the expression X+1 with a=X, b=1, and will match the expression X with a=X-1, b=1. The pattern a+b**2 will match the expression X with b=0, a=X.

3. Neither is fixed: Any exponentiation is searched for. Exponentiation is treated as a two-argument function with name MEXPT as in E below.

4. If an exponentiation being searched for in a sum is actually the only item left in the sum (e.g. y**x + A after A has been matched and removed) then other special cases are considered. If the base B is fixed, then B**E matches 1 if $B \neq 0$ and E matches 0. If the exponent E is fixed, then B**E matches 0 if E is a number greater than 0 and B matches 0.

E. If ai is a specific function (e.g. SIN) then the first occurrence of that function is searched for. The arguments of the pattern are compared with the corresponding arguments in the expression, and a check is made that the same number of arguments appears in the pattern and in the expression. If all the component matches succeed, ai, the pattern, (now fixed) is subtracted from e.

F. If ai is a function whose name is an unmatched variable, then any function, (possibly +, *, or **) is searched for, and treated as in E.

III. If p is a product, Πai then the sum operations (except for II-B) are duplicated, with "divide" replacing "subtract" and "product" replacing "sum." Since products within products are not possible with the MACSYMA simplifier, the action taken in II-A has a correlate in III only if the simplifier is turned off; in such situations, semantic pattern matches will not succeed anyway.

IV. If p is an exponentiation, the p is treated as in II-D, 1, 2, and 4. If neither the base nor the exponent is fixed, (the situation of II-D-3), e is treated as follows:

A. If e is 1, p is compared to 1**0.

B. If e is 0, p is compared to 0**1.

C. If e is not an exponentiation, p is compared to e**1.

D. If e is an exponentiation, the respective bases and exponents of p and e

are compared.

V. If p is some specific function, it is treated as follows: The function name in p (e.g. SIN) must match the leading operator in e. The respective arguments of the pattern and expression are then compared and a check is made that the same number of arguments appears in the pattern and in the expression. If all the component matches succeed, the pattern succeeds.

VI. If p is an unspecified function whose name is unmatched, it is treated as in V, except that the unmatched function name of p is compared to the leading operator of e.

VII. If p is an atomic unmatched variable, it is compared to e.

These operations may be nested to an arbitrary depth, since comparing a pattern and an expression may invoke comparisons of subexpressions. Furthermore, this algorithm is exhaustive, in the sense that given any syntactically valid MACSYMA expression, a pattern matching process will be defined for it.

Appendix II

The following LISP listing of QUAD uses several system conventions which can be briefly summarized as follows:

All user variable-names have a dollar sign prefixed to them. The *KAR[ERRSET[...]] construction serves only to catch illegal operations or ERR[]'s and return NIL in such instances. MATCOEF[X,Y] returns the coefficient of Y in X. The definition of "coefficient" used here may be found in [10]. It corresponds to the usual intuitive notions, but supplies answers when intuition generally falters. MEVAL[X] is the MACSYMA evaluator. It substitutes values for variables in the expression X, evaluates the result, and returns a simplified expression as an answer. RATSIMP[X] rationally simplifies X. RETLIST returns a list of its arguments and their values.

The G00n names are symbols produced to meet the need for unique new variable names.

```
(DEFPROP $QUAD
         (LAMBDA (G0042 $X)
 (*KAR (ERRSET (PROG (G0043 G0044)
    (SETQ G0043
          (MATCOEF G0042
           (MEVAL (QUOTE ((MEXPT SIMP)
                                  $X
                                  2)))))
   (COND ((MEVAL (QUOTE (($NONZEROANDFREEOF)
                         $X
                         G0043)))
                  (SETQ $A G0043))
                 ((ERR)))
   (SETQ G0042 (MEVAL (QUOTE (($RATSIMP)
                              ((MPLUS)
                               G0042
                               ((MTIMES)
                                -1
                                G0043
                           ((MEXPT SIMP)
                                  $X
                                  2)))))))
    (SETQ G0044 (MATCOEF G0042 $X))
    (COND (($FREEOF $X G0044) (SETQ $B G0043))
          ((ERR)))
    (SETQ G0042 (MEVAL (QUOTE (($RATSIMP)
                               ((MPLUS)
                                G0042
                                ((MTIMES)
                                 -1
                                 G0044
                                 $X))))))
    (COND (($FREEOF $X G0042) (SETQ $C G0042))
          ((ERR)))
  (RETURN (RETLIST $C $B $A $X))))))
EXPR)
```