MAC TR-95

ESSAYS IN ALGEBRAIC SIMPLIFICATION

RICHARD J. FATEMAN

APRIL 1972

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE

MASSACHUSETTS 02139

ESSAYS IN ALGEBRAIC SIMPLIFICATION

A Revision of

A Thesis Presented

by

Richard J. Fateman

APRIL 1972

PROJECT MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Cambridge Massachusetts

ABSTRACT

This thesis consists of essays on several aspects of the problem of algebraic simplification by computer. Since simplification is at the core of most algebraic manipulations, efficient and effective simplification procedures are essential to building useful computer systems for non-numerical mathematics. Efficiency is attained through carefully designed and engineered algorithms, heuristics, and data types, while effectiveness is assured through theoretical considerations.

Chapter 1 is an introduction to the field of algebraic manipulation, and serves to place the following chapters in perspective.

Chapter 2 reports on an original design for, and programming implementation of, a pattern matching system intended to recognize non-obvious occurrences of patterns within algebraic expressions. A user of such a system can "teach" the computer new simplification rules.

Chapter 3 reports on new applications of standard mathematical algorithms used for canonical simplification of rational expressions. These applications, in combinations, allow a computer system to contain a fair amount of expertise in several areas of algebraic manipulation.

Chapter 4 reports on a new, practical, canonical simplification algorithm for radical expressions (i.e. algebraic expressions including roots of polynomials). The effectiveness of the procedure is assured through proofs of appropriate properties of these simplified expressions.

Chapter 5 is a brief summary and a discussion of potential research areas.

Two appendices describe MACSYMA, a computer system for symbolic manipulation, an effort of some dozen researchers (including the author) which has served as the vehicle for this work.

PREFACE

This thesis describes a number of contributions to the art and science of manipulating algebraic expressions by computer. All the experiments were performed using MACSYMA, a computer system for symbolic manipulation of algebraic expressions now under development at the Massachusetts Institute of Technology's Project MAC. The contributions to MACSYMA of some 12 people are detailed in (31). My contributions are as follows.

I designed and programmed the rational function package, the radical simplifier, the semantic matching subsystem, "SOLVE", the rational "substitution" and "coefficient" routines, and portions of the supervisor and top-level simplifier. I also designed and implemented a major revision of the polynomial package incorporating the fast modular greatest common divisor algorithm (3). This revision makes possible the implementation of the much improved factorization algorithm now in progress (2).

Previous theses which describe parts of MACSYMA or its logical predecessors ((30), (35)) have included LISP (33) listings of the programs used. At this point it is becoming impractical to include such listings, constituting several hundred printed pages. Furthermore, such publication is of doubtful usefulness since listings and an operational system will be available in the near future to a community of users through the ARPA computer network. The system presently occupies some 110,000 36-bit computer words and will undoubtedly continue to grow.

Work reported herein was supported in part by Project MAC, an M.I.T. interdepartmental laboratory sponsored by the Advanced Research Projects Agency (ARPA), Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0001. The author was supported, while a graduate student, by ARPA under Air Force contract F19628-68-0101 with Harvard University, by the National Science Foundation under their Graduate Traineeship program and by Bell Telephone Laboratories under a contract with Harvard University. Revision of this thesis prior to publication as a MAC Technical Report has been supported in part by the Department of Mathematics, M.I.T. through National Science Foundation Grant GP 22796.

I wish to thank Professor Joel Moses of M.I.T, for his continuing interest, comments, and suggestions, which have guided me in this work.

I would also like to thank Professor A. G. Oettinger of Harvard, for providing support, advice, encouragement, and valuable criticism, while allowing me considerable freedom in my area of research.

CONTENTS

Chapter 1 - Introduction

> Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical, rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraic notation, were provisions made accordingly.
>
> --Ada Augusta, Countess of Lovelace (1844) ((26), p. 1)

During the past decade, developments in computer hardware and software have started to accomplish what Lady Lovelace envisioned over a century ago. By dealing with algebraic expressions, equations, and functions in terms of their symbolic representations, without reference to specific numerical values, computers are aiding working scientists and engineers facing a variety of non-numeric mathematical tasks. Some of the problems and potentials of algebraic manipulation by computer, and its most central process, simplification, are the topics of this thesis.

1.1. Algebraic Manipulation

To illustrate the difference between numeric and symbolic processing, consider a FORTRAN program which, given A, B and C, can apply the quadratic formula to approximate the roots of $Ax^2+Bx+C = 0$. A, B and C must, of course, have numerical values at run-time. This is strictly numerical processing. If A had as its run-time value the expression "Q," B had value "(-P*Q-1)," and C had value "P," the FORTRAN program would be useless. Nevertheless, by applying the quadratic formula symbolically, the two roots,

$$\frac{-(- P Q - 1) \pm \mathrm{SQRT}(P^2 Q^2 + 2 P Q + 1 - 4 P Q)}{2 Q}$$

can be represented. By further efforts, this expression can be reduced to

$$\frac{(1 + P Q) \pm (1 - P Q)}{2 Q}$$

or the two values P and 1/Q. One computer system for algebraic manipulation system, MACSYMA, which is now under development at M.I.T's Project MAC (31) and is the test-bed for most of the work described in this thesis, can be coaxed into performing this calculation through the following dialogue. The lines labelled Ci are typed by the user, those labelled Di and Ei by the MACSYMA system. (This, along with most of the other examples in this thesis consists of a file produced directly by MACSYMA which was later merged with the remainder of the text.)

```
(C1) EXP:Q*X**2-(1+P*Q)*X+P=0@

                 2
(D1)        Q X   - (P Q + 1) X + P = 0

(C2) SOLVE(EXP,X)@

SOLUTION
                             1
(E2)                     X = -
                             Q

(E3)                     X = P

(D3)                    (E2,E3)
```

It should be emphasized that all of the work described here is wedded to MACSYMA by convenience, not necessity. The techniques which are considered are of interest because of their relevance to mathematical problem solving in general, and to algebraic manipulation by computer most particularly. Although details of implementation will differ, the algorithms presented here should be useful in a number of computer systems now under development (1). Since it serves as a concrete base for comparing our techniques with those of other systems, we will make frequent references to MACSYMA; however, the philosophy and algorithms, rather than the programs themselves are really the topics of interest. Details of the implementation have been included when they serve to illustrate particular points in dealing with problems of algebraic manipulation.

1.2. Algebraic Manipulation by Computer: Prospects and Realities

I fully agree with R. W. Hamming that "the purpose of computing is insight, not numbers." ((38), p. viii). Mathematical analysis has traditionally been preferable to numerical approximation techniques because the resulting exact symbolic answers often represent a more direct path to insight than sets of approximate numbers. In the search for insight into mathematical and physical problems, difficult analytical and algebraic tasks should be delegated to computers just as difficult numerical tasks have been delegated in the past. I believe that computers can serve an important function in analysis analogous to the role they have come to serve both in bringing numerical analysis to its present state of refinement, and in producing answers to real problems.

An algebraic manipulation system is able to rapidly and reliably "massage" expressions orders of magnitude larger than ones comfortably handled by humans. For example, computers have demonstrated their facility in handling numbers, hundreds of digits in length, and equations requiring several pages for display.

These advantages are fairly obvious. Unfortunately, attempts to harness these advantages have often ignored a number of major problems (detailed below) which must be tackled in order to provide useful services to working mathematicians. Most of the early "systems" and "languages" for algebraic manipulation, having failed to consider these problems, disappeared shortly after their introduction. In many cases, the relevant problems were not yet recognized, much less solved. An unfortunately large number of newer efforts in algebraic manipulation systems have fallen into the same traps (e. g., (32)) and have not recognized the significant contributions of many of the researchers of the past ten or so years. Some have taken the attitude that a slightly more flexible programming language is all that is needed to suddenly open up the realm of algebraic manipulation capabilities. These researchers (most often programming language designers) should examine their claims in the light of the Formula Algol (37) experience; namely, that language features alone, regardless of their variety, do not make a useful algebraic manipulation system. Algorithms (23) and data structures are most important, and unless these are carefully considered, researchers entering the field will continue to repeat the mistakes of others; they will stand on the feet, rather than the shoulders, of the earlier contributers.

We do not wish to embark on a survey of algebraic manipulation systems since there are several easily accessible references. One is the exhaustive annotated bibliography of the field begun by Jean Sammet and continued by John Wyman (42). Since many of the listed papers are of historical interest only (even many recent ones, for the reasons given above), a more selective source on recent work is a better introduction to the field. W.A. Martin, in (30), critically surveys the progress in algebraic manipulation systems up to 1967. Max Engeli, in (11), gives his views on achievements and problems in the field to 1968. It is an indication of the rapidity of change in the field that some of the break-throughs mentioned by Engeli have been eclipsed by more recent developments. (Specifically, calculating factorizations and greatest common divisors can now be done much faster than by using methods mentioned by Engeli.) Perhaps the most useful index to the field to this time is the "Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation" (March, 1971) (1). It is a collection of tutorial and research papers describing important current work in most areas of the field. Chapters 2 and 3 of this thesis were presented at this symposium in slightly different forms (14) (31).

1.3. Problems and Goals

To some extent the major problems in algebraic manipulation depend on one's viewpoint. The broad view is to look at algebraic manipulation as a problem in artificial intelligence, the eventual goal being the construction of an expert mathematician (e.g. see (31)). The view taken here is much more limited, but can be considered as a preliminary to the broader problem. We wish to provide a tool capable of performing a wide range of services for a mathematician or engineer. These can perhaps best be envisioned as a spectrum of facilities ranging from a fancy desk-calculator, to (in some specific areas) an expert problem solver.

The system has facilities for indefinite precision integer and rational number arithmetic and finite field (modular (9)) arithmetic, in addition to the usual floating-point facilities of a modern digital computer. It has the ability to perform all elementary operations on multivariate polynomials and rational functions. It is capable of factoring polynomials, finding their greatest common divisors, calculating partial fraction expansions, derivatives and integrals of rational functions. It can perform routine substitutions, transpositions, etc. It incorporates the most efficient algorithms available, and may have several methods for performing a task, providing different types of efficiency, or efficiency over a wider domain than is possible with a single method.

As we understand larger classes of functions and operations, the practical power of the system will be expanded. Radical expressions (e.g. roots of polynomials) constitute one class which has been added to MACSYMA by this author. Recent additions include inequalities, polynomial arithmetic over finite fields, and power series generation and manipulation.

Further along the spectrum toward an expert mathematician, we can envision an ideal system as follows. It understands scientific notations and can be taught special notations. It is clever at presenting results in easily readable form. It can understand instructions (e.g. an algorithm presented as an Algol procedure) and follow them precisely. It can learn new methods for solving problems, but it already knows how to apply a large number of procedures (algorithmic and heuristic) which are useful for solving differential equations or sets of linear equations, finding indefinite or (improper) definite integrals, limits, etc. It has large amounts of data (e.g. tables, textbooks, simplification rules) at its disposal, and can be told to modify them for particular purposes. It will (if required) save all its calculations, and keep track of generated data for future references. It will (if required) provide additional information (e.g. timing data, intermediate results, procedures used) about the methods applied to solve the problem. It will work interactively with the user, or perform long calculations (correctly) in its "master's" absence. It understands enough about the problem domain to detect inconsistencies in its instructions and will balk at meaningless expressions or operations (e.g. divison by zero). It can numerically evaluate expressions and produce plots of functions.

We do not pretend that this view is, in fact, a listing of sufficient components of a modern algebraic manipulation system, nor do we claim that any implementation of such features will model the internal structure of a mathematician. We do feel, however, that the facilities noted above are important goals for a system like MACSYMA. Furthermore, a reasonable number of these goals have either been achieved, or are being approached.

MACSYMA is a large hierarchical computer system run in an interactive, time-shared environment. The real-time response of such a system is, we believe, necessary if a computer is to assume the role of a mathematical assistant. The user interacts with MACSYMA through its supervisor, a program which accepts character strings in a language resembling Algol-60. These character strings are parsed into LISP (33) s-expressions and passed to the programming language evaluator. This, along with the general simplifier, forms the heart of the system. The supervisor calls upon the rest of the resources of the system in carrying out the requests of the user.

Most commands invoke specific command programs which in turn draw upon the lower level routines to evaluate, process, simplify, and otherwise produce an answer, which is then returned to the supervisor. The supervisor displays the answer in a two-dimensional textbook-like format, and waits for the next user command. Generally some side effects will also occur, corresponding to the assignment of values to variables, the definition of programs, the setting of switches affecting future system behavior, etc. Other available side-effects include additional displays of expressions of interest and X-Y plots of numerical values. The commands draw on a wide range of facilities oriented about the several data types within MACSYMA. These facilities include algorithms for setting up and manipulating variabledimensioned arrays of symbolic elements, algorithms for performing definite and indefinite integration, algorithms for calculating limits of functions of a real variable, algorithms for the efficient manipulation of power series, polynomials, and rational functions. Additionally, a subsystem for the introduction of pattern-directed transformations on algebraic expressions is included. Appendix I, The Language and Commands of MACSYMA, offers specific examples of the forms in which these facilities are available. At present, the desk-calculator end of the spectrum is approximated by the facilities in MACSYMA while the more esoteric components are approximated only in some quite specific areas. Figure 1.1 indicates, in basic outline, the present components of MACSYMA and their interdependencies. The rectangles indicate subsystems which are still under development.

1.4. Specific Goals of the Thesis

This thesis is primarily a discussion of several facilities, designed and implemented by the author, which augment the abilities of MACSYMA, and in several cases, provide capabilities unique among current algebraic manipulation systems. Chapters 2 and 3 are concerned with the engineering of better algebraic manipulation systems, while chapter 4 presents the theoretical basis for some of the algorithms.

Chapter 2 discusses a user-level semantic matching capability, as implemented in MACSYMA. This subsystem consticompiler. Through this facility a user can specify new information and algorithms to the system in a manner which is concise, general, and straightforward. By simple top-level commands to the semantic matching subsystem, new programs are compiled and adjoined to the basic structure of the system.

By taking advantage of the 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.

Chapter 3 is concerned with expanding the usefulness of algebraic manipulation systems by taking advantage of canonical simplification programs. In this case we refer specifically to the rational function and radical canonical form facilities. First the data types and basic facilities are described, and then a number of new results are presented. The ease with which these can be used is a result of a critical design decision that algorithms (regardless of their origin) should be able to interact easily with the special data types available in MACSYMA. The new facilities include a routine to solve for a variable in an equation which is more powerful (in a practical sense) than that of any other system; programs which are more sophisticated in their ability to substitute values for sub-expressions which occur implicitly in a larger expression; and programs, used extensively for pattern matching, capable of finding "coefficients" (suitably defined) in an expression.

Chapter 4 describes our radical canonical simplification algorithm. With this, many algorithms can be successfully applied to larger classes of expressions than had previously been possible. The theoretical results behind the approach are developed, and compared to the work of Caviness (5) and others. The simplification procedure itself is shown to be quite practical (in contrast to Caviness'), and for many purposes, at least as useful. Extensions to exponential and logarithmic situations are pointed out and those which can be implemented at reasonable cost have been added to the algorithm.

Chapter 5 summarizes the current capabilities, both theoretical and practical, of computer aids to non-numerical mathematics, and then discusses research problems which appear at this moment to be both interesting and important from our point of view.

The two main appendices serve as documentation for parts of the MACSYMA system. They are not intended to be complete, since MACSYMA will be in a continual state of development for at least several years. Appendix I describes the outward view of some of the MACSYMA commands. Appendix II describes the MACSYMA rational function package in sufficient detail to make its transfer to other LISP systems simple. The rational function package is of particular interest in that it is self-contained, and sufficient for many polynomial "crunching" tasks. It includes a number of particularly efficient algorithms, and may be of interest to mathematicians who prefer to dispense with the amenities provided by a total system in order to make more core storage available.

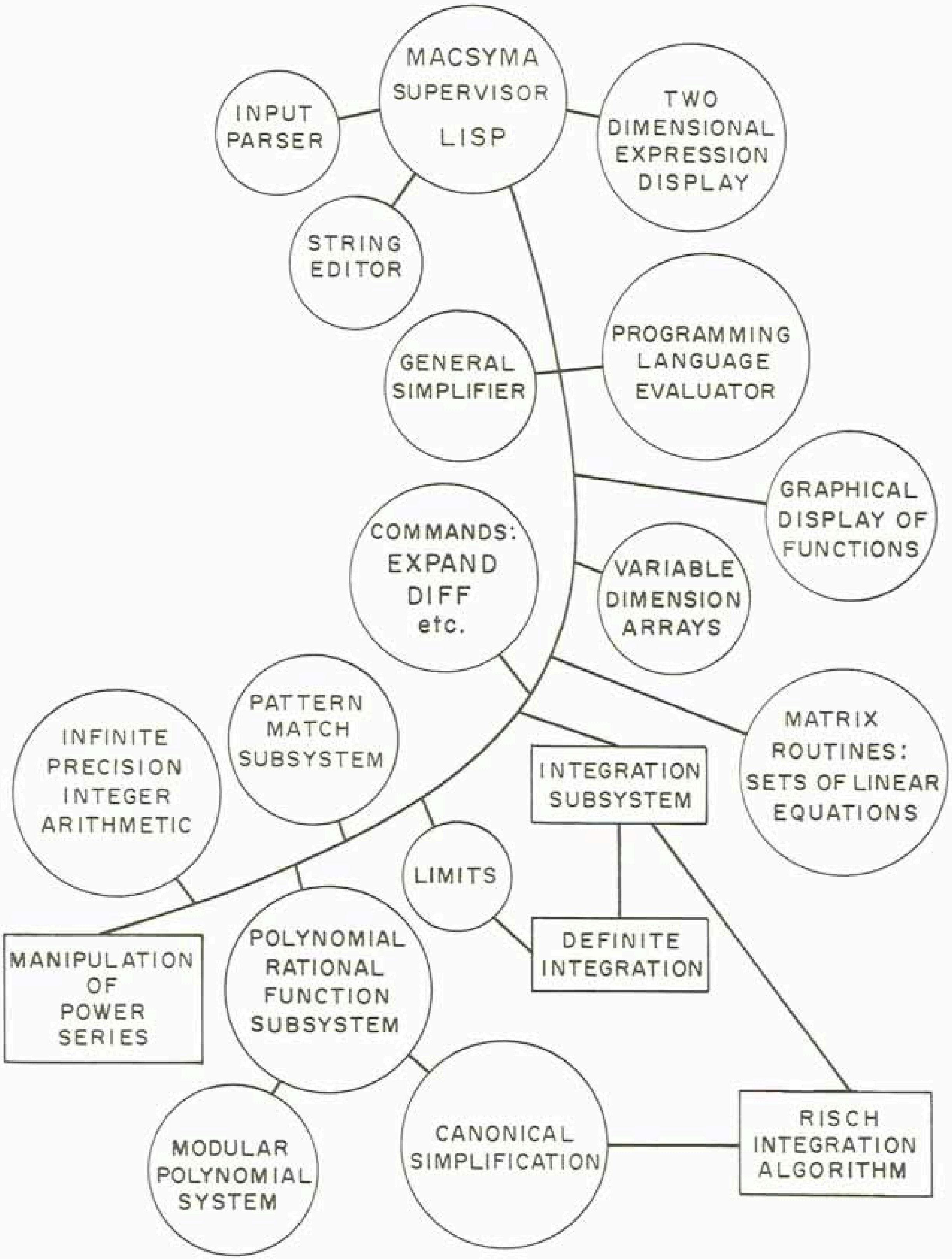

FIG. 1. THE MACSYMA SYSTEM.

Chapter 2

The User-Level Semantic Matching Capability in MACSYMA

2.0 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, in some cases, 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 chapter describes pattern matching facilities designed and implemented by the author for MACSYMA. 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 (12). Inside an algebraic manipulation system, such 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 (16) or AMBIT/S (8). 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.

The notion of using the semantics of an algebraic expression requires explanation. Some properties of ordinary addition and multiplication can be usefully included in the design of a program intended to recognize algebraic expressions as instances of more general patterns. For example, knowledge of the fact that addition is commutative and has identity 0 and the fact that multiplication is commutative and has identity 1, clearly improves the probability of finding a mapping between parts of a pattern and instances of that pattern in an expression.

In addition to these elementary properties, it is particularly useful for us to work with the fact that for any polynomial, P, a unique form can be derived such that the coefficient of any variable in P to some integer power can be found. Over a larger class of expressions, a simplified form will often display this characteristic of having "obvious" coefficients with respect to sub-parts of the expression.

We will refer to these, and similar properties of algebraic expressions as semantic properties. By the use of the semantic notions already mentioned, a pattern A*X+B might be matched to the expression X, with A matching 1, and B matching 0.

Additional semantic notions become more difficult to choose (and implement in a systematic fashion). For example, interpretations involving exponents must be carefully restricted to avoid conflict. Thus, if the pattern A**B is to match the expression 1, either A is 1 and B is undetermined <u>or</u> B is 0 and A is non-zero. Some (somewhat arbitrary) decisions concerning acceptable values for A and B are necessary. MACSYMA makes such a decision, which is described in the first appendix to this chapter.

We have chosen to implement the arithmetic interpretations of our matching programs using basically these semantic notions.

A less elaborate interpretation would prevent us from matching a pattern A*X+B to the expression X, with A matching 1 and B matching 0.

A more expansive interpretation of the possibilities leads into difficulties: allowing the coefficient of X**3 in the expression X**2 to be 1/X; allowing 2**n to match the expression 0 with n matching negative infinity, etc.

The exact limits chosen for any given implementation's ability to enlarge upon the elemental syntactic statement of a pattern has been, and will, no doubt, continue to be largely pragmatic. Furthermore, it is our belief that any attempt to produce a concise formalism for a pattern matching interpreter is bound to unnecessarily limit the power of the implementation.

Those matching formalisms cannot take advantage of the many useful, but non-systematic "tricks" which can be cleanly added to a pattern matching program. Therefore we will continue to take a pragmatic approach to semantic pattern matching, and try to reveal the reasoning behind our design features, and the methods used to implement them.

We will refer to those pattern matching programs with facilities which take into account at least the basic properties of addition and multiplication, as semantic.

Historically, Slagle's SAINT (43) and Moses' SIN (35) were the first demonstrations of a significant application of semantic pattern matching: large classes of expressions were mapped into forms with known integrals. Other, more general 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 (19) has a user-level matching facility. REDUCE gives the user (through the LET command) a limited matching facility which is considerably restricted in its power by its emphasis on efficiency. For example, patterns which are sums are not permitted. FAMOUS (16) and Formula Algol (37), neither of which is currently in use, provided matching facilities, which (as we shall see in section 8), were 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. Section 10 suggests other areas of usefulness in mathematics and man-machine communication. These sections are supplemented by appendices _to this chapter_: Appendix I contains precise, extended definitions of the matching procedures; Appendix II includes an example of a match program as compiled by the system; Appendix III considers the problem of defining classes of expressions over which matching procedures can be considered effective -- that is, under what circumstances a pattern match can determine membership in formally defined classes of algebraic expressions.

2.1. 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 <u>we must be able to make the success of a match dependent on the matched values satisfying predicates</u>. 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. INT(X) is TRUE when X is an integer.

FREEOF, TRUE, and INT 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 (see Chapter 3). 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(<u>name</u>,<u>predicate</u>(<u>arg</u>$_1$, ..., <u>arg</u>$_n$))@.   ($n \geq 0$)

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 ($\underline{arg}_1$, ..., $\underline{arg}_n$) 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.

2.2. 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(<u>programname</u>, <u>pattern</u>, $\underline{\text{patternvar}}_1$, ..., $\underline{\text{patternvar}}_n$)@

($n \geq 0$).

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 <u>programname</u> (e.g. LINEAR, F3, COSSIMP) which will test to see if the pattern <u>pattern</u> (i.e. A*X + B, etc.) can be applied to its first argument. The program will have n additional arguments, corresponding to the <u>patternvar</u>s.

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 $\underline{\text{patternvar}}_i$'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 <u>patternvar</u>'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 <u>pattern</u> 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 <u>pattern</u> is an "atom," or single variable, then it <u>is</u> 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)
```

This illustrates another principle in matching patterns. <u>If A is undeclared and not a pattern variable, A in a pattern will match only A's current value</u>. (If A has no value, then MACSYMA provides "A" for the value of A. As a special case, <u>constants match only themselves</u>.)

2.3. 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.

2.4. 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 this chapter's 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 (see Chapter 3) 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 described in Chapter 3 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 non-zero, 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 <u>coefficients in an expression should be extracted semantically</u> (i.e. the coefficient of Z must be extracted using the semantics of the operators + and *).

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

Line (C6) demonstrates that <u>summands in the pattern which are missing in the expression are matched with 0</u>. This is what happened to the term B*X in the QUAD pattern. Furthermore, <u>if a product is matched with 0, one of its factors must match 0</u>. 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, <u>factors in the pattern which are missing in the expression are matched with 1</u>. 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 this chapter's 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 potentially great increase in cost, combined with no assurance that the user would be happy anyway, make such an approach somewhat unattractive. (It should be said, however, that in cases where heuristics and back-up are part of the processing itself, as in early stages of SIN (35) it may be convenient to use the pattern matching program for the basis of heuristics.) 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 often not even 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 [] is used to enclose a list consisting of local (i.e., "dummy") variables within a BLOCK.)

```
(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 BLOCK ([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.

The fact that we insist on completely directed or "anchored" (12) searches in a pattern is both a strength and a weakness. Some patterns are inherently ambiguous, and all possible types of matches must be explored. This is the case in symbolic integration. If such ambiguous patterns are the rule, rather than the exception, we would be seriously inconvenienced by having to simulate back-up (as above), in every case.

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.

2.5. 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 (16), PANON-IB (7), AMBIT/S (8), Formula Algol (37), and SNOBOL (12), 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(<u>rulename</u>,<u>pattern</u>,<u>replacement</u>)@.

If the rule named <u>rulename</u> is applied to an expression (by one of the APPLY programs below), every subexpression matching the <u>pattern</u> will be replaced by the <u>replacement</u>. All variables in the <u>replacement</u> which have been assigned values by the pattern match are assigned those values in the <u>replacement</u> 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 FALSE.

2.5.1 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($\underline{e}$,$\underline{r}_1$, $\underline{r}_2$,...,$\underline{r}_n$) applies the first rule, $\underline{r}_1$, to the expression $\underline{e}$ 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($\underline{e}$, $\underline{r}_1$,$\underline{r}_2$,...,$\underline{r}_n$) differs from APPLY1 in that if the first rule, $\underline{r}$ 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 (23), (37) 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)@
```

2.5.2 An Example

Here is an example of using rules to alter an expression. The symbol S is used as an abbreviation for e^Z, RATSIMP (see Chapter 3) 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
```

2.6. 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. He may also wish to tell 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 (44) has been implemented. There are two commands to advise the simplifier: TELLSIMP, and TELLSIMPAFTER. They have the following forms:

TELLSIMP(<u>pattern</u>, <u>replacement</u>)@
TELLSIMPAFTER(<u>pattern</u>, <u>replacement</u>)@

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

TELLSIMP analyzes the <u>pattern</u>, 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 <u>or</u> 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 described in chapter 3. In it the approach used by REDUCE to handle products (17, 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 <u>after</u> 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.

2.7. Non-Commutative Multiplication

At this time, a standard non-commutative multiplication simplification program is not generally included in MACSYMA. There are several different programs available, but it may be the case that none of them does exactly what is required in a given problem area. This section describes how one might add a fairly extensive hand-tailored 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 multiplication 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 ; more generally, A .A = A   . Since our pattern matcher is clever enough to recognize A as an occurrence of A , 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 have

```
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, $(A^n).\mathrm{INV}(A^m) = A^j * \mathrm{INV}(A^k)$ 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.

2.8. Comparisons with SCHATCHEN, FAMOUS, REDUCE, Formula Algol

SCHATCHEN (35), Moses' matching program 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 AX^2+BX+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, $Q = 2X^2 + YX^2 + 3 + Z$. There are two terms involving X^2. For the pattern $AX^2 + BX + C$ to match Q, A must match 2 + Y. This is indicated to SCHATCHEN by using the indicator 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 AX^2 and BX 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 assure 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 coefficient-finding routine we use can find that the coefficient in $(2x)(3x+1)$ 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 (23) 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. Fenichel's "super-match" proposal, implemented in (32), changes each single pattern into a large number of similar patterns by transformations of commutative operators (etc.). This is scarcely an improvement in efficiency, and appears to be useful only as a shorthand in writing out long rule sets. 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" ((16) 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 $e^{(x+\log(y))} \Longrightarrow y\ e^{x}$. If the sum was not in an exponent, a great deal of time has been wasted.) Waste of this sort is avoided by 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 considered just as adjacent as f and g. What is less clear is how one might note that f and g, being integer-valued functions, make 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 (19), 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.

2.9. 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.

```
          DY
   F(X) (--) + G(X)*Y + H(X) = 0
          DX
```

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. (Details of the programming syntax are described in Appendix I to this thesis.) Note 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)+G*Y+H$
(C5) DEFMATCH(PAT,P,Y,X)@

   DY
F (--) + G Y + H
   DX

IS THE PATTERN
(D5)                    PAT

(C6) LINDEP(EQ,Y,X) :=BLOCK([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)+3*Y+4@

                   DY
(D7)               -- + 3 Y + 4
                   DX

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

                   CONST    4
(D8)          Y = ------- - -
                     3 X    3
                   %E
```

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.

2.10. Other Applications

One of the major problems of algebraic manipulation systems has been the lack of substantial tools to aid in human comprehension of large expressions. Hearn, in (20), explores this problem. He displays an expression with a large number of dependent variables, and by properly choosing substitutions of expressions for variables, produces a new expression reduced in size and complexity. This requires a high degree of human experimentation and interaction with the computer. In chapter 3 we describe more sophisticated substitution methods which relieve the user of some of his headaches, but still require explicit "substitute A for B" type commands. By contrast, the Markov algorithm processing of expressions, combined with semantic pattern matching, can lead to more general styles of substitution: e.g. For any Z, substitute Y(Z) for COS(w*t+Z).

Another approach toward improving comprehension has been the automatic "breaking-up" of expressions at (computer-chosen) positions. The parts are then easier to display (30), or manipulate further (10), (20), (34). Unfortunately, except for special cases, the computer-chosen break points tend to obscure the underlying structure. By breaking an expression up at points suggested by user-supplied patterns, and renaming the pieces (say by allocating coefficients of certain types and locations to a matrix), inherently bulky expressions can be reduced to more tractable sizes. As a simple example, the pattern A + B*%I, for A and B declared free of %I serves to separate real and imaginary parts of an expression.

2.11. 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 <u>patterns</u> for the interpreter cannot be compiled since they are, of necessity, LISP data. On the other hand, the pattern <u>programs</u> of our system can be compiled completely into machine code.

The advantages of semantic (as opposed to syntactic) matching are clear. Semantic matching as implemented in MACSYMA allows the user to introduce new information relying on a wide range of previously developed information and simplification rules. Syntactic methods would require considerably more efforts (since all information would have to be encoded in syntax only) and result in a less powerful extension.

Chapter 2 - 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. By describing the algorithm used to compile patterns into programs, this appendix 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 FALSE 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.

Definition: A pattern is anchored if after all fixed parts have been subtracted, divided out, or otherwise removed from an expression instance of the pattern

(1) The remaining pattern consists of an isolated unmatched variable not in a sum or product.

or (2) There is at least one fixed subpart of the pattern such that any expression instance may be separated into at least two parts, each part, furthermore, corresponding to an anchored sub-pattern of the original pattern.

The pattern compiler in MACSYMA seeks out anchors, and successively compiles program segments to remove those parts which can be unequivocally identified. If the remaining parts provide no anchor, or if several not distinct anchors are provided, the compiler will not be able to take advantage of its built-in knowledge. In some cases, warning messages will be produced.

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 complicate the matching process. (The MACSYMA input and output routines, in order to improve readability, reintroduce 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 (23) pp. 172-174)

The remainder of this appendix describes in detail the methods used to seek out anchors. These methods vary depending on the context, so that an anchor within a sum is different from an anchor within a product. Although we have tried to make this description as clear as possible, it is not our intention that a user of MACSYMA read this as a prerequisite to using the pattern matching system. A user should compose patterns in the interactive MACSYMA environment, and by viewing the explicit actions of the patterns themselves, he should judge their suitability. This is similar to the philosophy of other parts of MACSYMA: a user will rarely know _a priori_ whether or not an integration can be performed by the system, or whether an indicated command is powerful enough to accomplish his task. Although it is desirable to describe capabilities in a clear manner, it is unreasonable to restrict the capabilities to that which can be so described.

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, 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, 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, $\sum a_i$, then all fixed a_i are subtracted from e, and then the rest of the a_i are examined as follows:

A. If a_i 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 a_i 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 Chapter 3, in the description of the RATCOEF command.) If it fails, the match fails, otherwise it proceeds. (That is, v is compared to the coefficient of f in e.)

C. If a_i is an unmatched variable, then it should be the only unmatched a_i, since it will match the rest of the expression. If selectors are used, there might be more than one remaining a_i, 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 a_i 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, a_i would be fixed, and thus be treated under 1.)

1. The base is fixed: A search is made for an exponential operator with the given base. If the search succeeds, the pattern for the exponent is compared to a_i'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 a '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 ≠ 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 a_i 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 to assure that the same number of arguments appears in the pattern and in the expression. If all the component matches succeed, a_i, the pattern, (now fixed) is subtracted from e.

F. If a_i 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, $\prod a_i$ then the sum operations (except for II-A and 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 or II-B 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, then 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 as found by the RATCOEF of chapter 3. 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.

```
(DEFUN $QUAD
 (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 $B G0044))
                      ((ERR)))
                (SETQ G0042 (MEVAL (QUOTE (($RATSIMP)
                                           ((MPLUS)
                                            G0042
                                            ((MTIMES)
                                             -1
                                             G0044
                                             $X)))))))
                (COND (($FREEOF $X G0042) (SETQ $C G0042))
                      ((ERR)))
                (RETURN (RETLIST $C $B $A $X))))))
```

Appendix III

This appendix considers the question of pattern matching from a more theoretical standpoint. It answers some questions about the formal power of pattern matching in determining membership of an expression in a class, and the ability of the pattern match to uniquely determine the values of the variables in the pattern. Many of these results may seem trivial or obvious; nevertheless, they are not expressed elsewhere. Some especially trivial results are (1) Any expressions E, synthesized by MACSYMA can be matched by a pattern, namely, the pattern E. and (2) Any expression in MACSYMA can be completely decomposed by some pattern: By using explicit matches for operators (as in section 4, line C6), every single component of any expression can be given a name. (Since we usually seek to define <u>general</u> patterns, such explicit matches are rarely of great use.)

<u>Definition</u>: A pattern match program (PMP) is a program produced by the implementation of the algorithm described in Appendix I, given a valid MACSYMA expression.

<u>Theorem</u> 2.III.1. A PMP for a finite pattern is finite in specification.

<u>Proof</u>. A finite pattern, written as a tree, has a finite number of nodes. The algorithm of Appendix I traverses the tree once, emitting a finite number of finite steps at each node (in practice, fewer than 3 LISP "S-expressions" per node). The algorithm terminates when the tree has been traversed. (A

more rigorous proof is possible, but would require a detailed analysis of the correctness of our programming implementation. This could be done by case analysis, as described in (28).)

We will assume, for the remainder of this appendix that all PMPs are defined for finite patterns, and are therefore finite in size.

Theorem 2.III.2. A PMP always terminates given finite expressions for each of its arguments if its predicates always terminate, and the evaluator (MEVAL) always terminates (with a finite result) on finite expressions.

Proof. A PMP is a finite non-looping sequence of steps. Each step terminates, since it is either an application of a predicate, an evaluation of an expression, or the extraction of the ith argument of an n-ary function. The last of these clearly terminates since i and n are finite by hypothesis, and the first two terminate by hypothesis.

Since patterns are themselves predicates, it is possible (and often useful) to use them recursively. In such cases termination will be difficult to guarantee by this theorem. Although this theorem states sufficient conditions for termination, these conditions are not necessary, since, for example, non-terminating predicates may be present in a pattern, but may never be applied if the pattern fails first.

For the remainder of this appendix, all PMP's are assumed to be terminating. Thus a PMP divides the set of finite algebraic expressions into two classes, P, the expressions which satisfy the pattern, and N, those which do not.

Now, a result which determines a theoretical limit (but not necessarily the best limit) on the power of pattern matching:

Theorem 2.III.3. Unless N or P is empty, there is a MACSYMA expression p in P which is functionally equivalent to a MACSYMA expression n in N.

Proof. Let us assume for the moment that the MACSYMA simplifier is unaware of special angle simplifications, and let a PMP program pass only expressions which match zero (0). Neither P nor N is empty, so this theorem asserts there is an element in N equivalent to zero. We can show there are many. One of them is COS(PI/2). To see if COS(PI/2) matches 0, (see the first line of the algorithm, Appendix I) we simplify COS(PI/2)-0. The result, COS(PI/2), is not identically the expression "0" and therefore is in N.

One might blame the MACSYMA simplifier for this inadequacy, except for the following lemma, which proves that the simplifier cannot be made sufficiently adequate.

Lemma. (Richardson, (36), see (6) also) Let R be the set of expressions generated by

(i) the rational numbers, and the real numbers pi and log 2,

(ii) the variable x,

(iii) the sine, exponential and absolute value functions.

Then if E is an expression in R, the predicate "E is equivalent to 0" is recursively unsolvable.

Since all of these operations and constants are permissible in MACSYMA, there is no computation which can exclude 0-valued functions from N. Furthermore, since "0" can be added to any pattern whatsoever, the same analysis holds for any PMP.

This concludes the proof of theorem 2.III.3.

Our only hope is that some sub-domains within R have less disastrous consequences for pattern matching. Such is the case. Let us use the convention that $A_1,\ldots,A_m$ are MACSYMA variables which have been declared TRUE, and $X_1,\ldots,X_n$ are constants. (This convention may seem odd, but consider that AX+B as a pattern has variables A and B and constant X.)

Since we are restricting the domain of expressions handed to PMP's, we will be affected by the power of the simplifier. Thus while E = X + COS(PI/2)*SIN(X) does not look like a polynomial in X, a suitable simplifier will transform E into the equivalent expression X, which _is_ a polynomial in X. All expressions mapped into a domain D by a simplifier s, are members of the class Ds. No rigorous definition of Ds's will be attempted, since the simplifier in MACSYMA defies simple analyses, and in any case, it can be altered by the user.

<u>Theorem</u> 2.III.4. Let D be the domain of polynomials in any finite number of variables $\{X_i\}$ with integer or symbolic coefficients. A pattern consisting of any (expanded) member of D with variable coefficients $\{A_i\}$ will match uniquely any member of Ds.

<u>Proof</u>. An expanded member of D will look like

$$\sum_i A_i\,(X_1)^{M_{1,i}} * \ldots * (X_n)^{M_{n,i}}$$

A PMP for this pattern will consist of a finite set of calls to the coefficient-finding routine, which will assign to each A_i the coefficient of $X_1^{M_{1,i}} * \ldots * X_n^{M_{n,i}}$. These coefficients can be extracted because this representation of polynomials is canonical, and the coefficients are obviously unique in any expression (regardless of its original form) which can be transformed into an equivalent polynomial in $X_1, \ldots, X_n$.

<u>Theorem</u> 2.III.5. Let D be the domain of rational expressions (i.e., ratios of polynomials as in theorem 4). A pattern consisting of the ratio of two expanded polynomials with variable coefficients will match uniquely any member of Ds.

Proof. The numerator is matched as in theorem 4, and the denominator, which appears as a polynomial raised to the -1 power, is matched as in theorem 4. The absence of a denominator will cause the pattern for the denominator to be matched against 1.

These results can be extended in various directions, but results become more specialized and less illuminating. For example,

Theorem 2.III.6. Let f be an n-ary function with no simplification rules in the simplifier s. Let Ds be the Ds of the previous theorem, and $\{d_i\}$ elements of Ds. Let Df be the set of expressions

$$d_0 * f(d_1, \ldots, d_n) + d_{n+1} .$$

Then an expression in Df with variable coefficients will match uniquely any member of Dfs.

Proof. The single occurrence of f can be found, and its "coefficient" d_0 and "constant term" d_{n+1} can be matched as in theorem 5.

Typical of statements which are true, but are of only limited interest is: Let Ds be the Dfs of the previous theorem, and let $\{d_i\}$ be in Ds. Let E be the set of expressions $d_0^{d_1}$. The pattern matches uniquely any member of Es, but the uniqueness is imposed by the match algorithm. Thus 1 is a member of Es, as 1**0, but the possibilities 1**1 or x**0 are not considered.

When we restrict ourselves to matching expressions composed over classes for which canonical forms exist, as in theorem 4, quite neat results can be obtained if the simplifier is able to compute these canonical forms. For many areas of interest, canonical form algorithms do not exist, yet being able to recognize members of particular classes within the confines of a simplifier can still be useful. For example, recognizing the parameters of a differential equation, even if it can be done only in some standard form, is useful, even if some other, rarely encountered, but equivalent form is not recognized at all.

In comparing various systems in this context, the principal point is that for the same domain D, the ability of the simplifier, s, to reduce an expression to essential components, strongly influences the size of the set Ds. MACSYMA's simplifiers (the "ordinary" one which can be modified by the user, the RATSIMP rational simplifier, and the RADCAN radical canonical simplifier of Chapters 3 and 5), provide a range of possibilities larger than that of any other existing system. While any system which is in theory equivalent to a Turing machine could in theory do as well (eventually) as MACSYMA, or even better, theorem 3 provides a bound on their theoretical capabilities just the same.

Chapter 3 - Extending the Power of the Rational Function Facilities

This chapter concerns the practical implications of what I believe to be the most significant design decision in MACSYMA. MACSYMA was designed with the intention of <u>not</u> necessarily restricting its components to the same data representation. The rational function package embodies the essentials of a special data type which, by suitable treatment, has yielded a number of new results. These results include particularly powerful techniques for extracting coefficients (section 3.3), for substitution (section 3.5), and for solving for a variable in an expression (section 3.6).

Since a number of other current systems (e.g. REDUCE (19)) also include analogous special rational function representations, the new ideas and techniques discussed here could, no doubt, be implemented elsewhere with relative ease.

By using the rational function representation (as opposed to the general representation), extremely fast processing is possible. For typical calculations which can be done either way, the rational function representation can easily reduce the time requirements by a factor of five or ten, becoming far more efficient as the problem increases in size. This in itself can be a significant asset. In order to make this point more concrete, and to demonstrate how MACSYMA compares to similar efforts elsewhere, some timing information has been compiled. Only the crudest efforts at making the cross-system comparisons truly comparable have been attempted; no doubt an extensive study could be conducted in balancing the differences of word-length, CPU cycle time, memory access time, size of storage, CPU instruction set, etc. The timings in table 3.1 are for the calculation of the first 10 polynomials in the "f and g" series, the details of which may be found in (11) or (31). The calculation is of two sets of polynomials in sigma, mu and epsilon, defined recursively in terms of each other and derivatives of lower order terms. The calculation can be indicated in MACSYMA's rational function representation through the following input:

```
X1: RAT(-SIGMA*(MU + 2*EPS))$
X2: RAT(EPS-2*SIGMA**2)$
X3: RAT(-3*MU*SIGMA)$
F[0] : RAT(1)$
G[0] : RAT(0)$
F[I] := -MU*G[I-1] + X1*DIFF(F[I-1],EPS)
          + X2*DIFF(F[I-1],SIGMA)
          + X3*DIFF(F[I-1],MU)$
G[I] := F[I-1] + X1*DIFF(G|I-1|,EPS)
          + X2*DIFF(G[I-1],SIGMA)
          + X3*DIFF(G[I-1],MU)$
F[10]@
G[10]@
```

Timings for systems other than MACSYMA are interpolated from (11) or from conversations with the authors of various systems presented at SYMSAM/2 (1).

Table 3.1 - A Cross-System Timing Comparison

System	Computer	Time (sec)	Adjusted Time*	Notes
	Machine-Language Systems			
PM	IBM 7094	4.4	18	
FORMAC	IBM 7094	39.0	156	
SAC-I	CDC 1604	25.8	26	
SYMBAL	CDC 1604	52.2	53	
SYMBAL	CDC 6600	4.3	65	
CAMAL	ATLAS-2	2.0	4	(1)
	Systems Written in Higher-Level Languages using Special Rational Function Representations			
REDUCE2	IBM/360-67	4.5	45	(2)
		10.	100	(3)
	PDP-10	10.5	55	(4)
IAM	PDP-10	152.	760	(5)
MACSYMA's rational function routines	PDP-10	14.8	30	(6)
	Systems Written in Higher Level Languages using General (Tree) Representations			
Korsvold's System	IBM 7094	119.0	475	
MACSYMA's general representation	PDP-10	76.	152	(7)

Notes for Table 3.1

*For a direct comparison, we have used the following somewhat controversial speed factors: CDC 1604 = 1, PDP-10 (with 2.8 microsecond memory) = 2, ATLAS-2 = 2, IBM 7094 = 4, PDP-10 (with 1 microsecond memory) = 5, IBM 360/67 = 10, CDC 6600 = 15. In addition, the notes must be taken into account in computing the adjusted time. REDUCE2, Korsvold's System, and MACSYMA are all written in LISP, and are subject to variations depending on the efficiency of the underlying language implementation.

(1) A particularly rough interpolation; the actual time was 7.4 seconds for F and G to index 19. CAMAL uses a representation which packs a great deal of information in a single node; it thus uses less space, and less pointer-following time than the other systems listed here.

(2) Using Stanford University LISP.

(3) Using IBM's Scratchpad LISP which is slower than Stanford LISP, since it packs two addresses in a 32-bit word, thus requiring shifts to adjust the addresses.

(4) Using 1 microsecond memory.

(5) Using 1 microsecond memory. IAM is written in AMBIT/L, and is interpreted, rather than compiled.

(6), (7) Using 2.8 microsecond memory.

These times (and, no doubt, other LISP times) can be decreased by some 40 percent by methods unrelated to the algorithms: By using a larger core allocation, LISP garbage collection time can be reduced; also, a cleverer LISP arithmetic statement compiler (now being implemented) would reduce calculation time further.

3.0 An Introduction and a "Political" Digression

Moses, in (34), describes a spectrum of attitudes towards algebraic manipulation ranging from the "radical" to the "conservative." According to this classification, a radical system will transform a user-supplied expression into an internal format which consists of an encoding of the expression in a special unique simplified form. This drastic transformation generally destroys superficial resemblances between the input and output. The only attribute necessarily preserved is the functional value of the expression. Polynomial and rational function systems generally fall in the "radical" category. The contrasting "conservative" approach does almost nothing but that which is specified by the user; it keeps the internal form as nearly the same as the external form as is possible, and generally accepts a wide variety of expressions (wider than polynomials and rational functions).

The top-level ("liberal" in Moses' terminology) "general" simplifier and evaluator in MACSYMA takes a stance in the middle. It has some built-in rules (e.g. concerning zero and one, collecting terms) and by ordering terms in sums and products, does a fair job of simplifying a large class of expressions. Its importance lies in the fact that it allows certain subsystems to explore the far reaches of the "political" spectrum. Because of the conjunction of different approaches, radical simplification algorithms can be applied to expressions which would not ordinarily be considered proper inputs. For example, the ability to manipulate $e^{2X} + 2e^{X} + 1$ as a quadratic in e^{X} (and apply polynomial "radical" processing) is quite useful, even though the expression is not quite fair game for ordinary polynomial systems. MACSYMA is capable of factoring the above expression into $(e^{X}+1)$, and treating it as a polynomial for various purposes; however, it is also capable of noticing that e^{X} can reduce to y when x=log(y). Polynomial or rational function systems are rarely aware of such possibilities in their data.

This chapter discusses the "radical" data handling facilities of MACSYMA, and their relation to the MACSYMA command level. In one particular instance (the SOLVE command) we show how radical and conservative handling of different parts of the same expression can lead to an end result which could be produced with either approach alone only with great difficulty. Other commands where rational simplification or other radical approaches are essential to programming effective algorithms are also discussed.

By an unfortunate coincidence in terminology, we will use the word "radical" in two senses. In one case, we will discuss a class consisting of algebraic extensions adjoined to the field of rational functions. This class is generally called the class of radical expressions (in the sense that a square root is a radical). In the second case, our approach to simplifying radical expressions is, in Moses' terminology, radical (i.e. drastic).

In this and later chapters, the algorithm and the command used for invoking the algorithm used to simplify _radical expressions_ will be referred to as RADCAN. RADCAN and the two commands to be described in section 3.1, RATSIMP and FULLRATSIMP are all classified as radical (i.e. _drastic_) simplifiers.

3.1 Basic Rational Function Commands

In order to clarify the discussion, it is necessary to distinguish between the two major internal forms for expressions in MACSYMA. Ordinary MACSYMA form is a delimiter prefix form which is typical of many list-processing implementations of algebraic manipulation systems. For example, $3x^2$ would be represented (glossing over inessential details) as (times 3 (expt x 2)), and x+y as (plus x y). By contrast, the _canonical rational expression_ (CRE) form in MACSYMA is an internal form especially suitable for rapid manipulation of sparse polynomials and rational functions.

In CRE form, $3x^2$ is represented, (again, glossing over details) as (x 2 3). The first element of the list is the variable, the second is its highest exponent, and the third, the coefficient of the just preceeding exponent. Thus $6x^2+4$ is represented as (x 2 6 0 4), and, allowing coefficients themselves to be polynomials, x^2y+7xz is (x 2 (y 1 1) 1 (z 1 7)). Since (y 1 (x 2 1) 0 (x 1 (z 1 7))) is an equivalent CRE representation, it should be clear that the ordering of variables must be specified to insure that equivalent CRE's are identical, that is, they are in <u>canonical</u> form.

CRE's in general represent <u>rational</u> expressions, that is, ratios of polynomials, where the numerator and denominator have no common factors, and the denominator is positive. Thus a CRE has three essential parts: a variable list (VARLIST), specifying the ordering of the variables, and two polynomial parts.

With these preliminaries, we can describe the actions of the rational function commands.

RATVARS(a,b,...) orders the variables listed in its argument list on a global variable list (VARLIST) so that the rightmost element of the list a,b,... will be the main variable of future rational expressions in which it occurs, and the other variables will follow in sequence. If a variable is missing from the RATVARS list, it will be given lower priority than the leftmost element. If several variables are missing, they will be ordered by the MACSYMA function GREAT, which uses an implementation of the ordering algorithm described in (34). The arguments to RATVARS can be either variables or non-rational functions (e.g. SIN(X)).

RATSIMP(EXP) rationally simplifies the expression EXP. That is, EXP is converted into a single fraction, whose numerator and denominator are polynomials over the integers, with no common factors. EXP is written in a recursive form: a polynomial in the main variable whose coefficients are polynomials in the next-higher-order variable, ..., whose coefficients are integers. This is accomplished by converting EXP into CRE, and then converting back to ordinary MACSYMA form for display.

For example:

```
(C1) (X**2-Y**2)*(Z**2+2*Z)/((X+Y)*W)@

                    2    2     2
                  (X  - Y ) (Z  + 2 Z)
(D1)              --------------------
                       W (Y + X)

(C2) RATSIMP(D1)@
                          2
                (X - Y) Z  + (2 X - 2 Y) Z
(D2)            --------------------------
                            W

(C3) RATVARS(X)$

(C4) RATSIMP(D1)@
                     2               2
                X (Z  + 2 Z) - Y Z  - 2 Y Z
(D4)            ---------------------------
                             W
```

FACTOR(EXP) factors the expression EXP into factors irreducible over the integers. If EXP is a rational expression (with a denominator not 1) both numerator and denominator are factored. If FACTORFLAG is set to TRUE, the integer multiplier, if any, is factored also. The algorithm can be used to factor polynomials in any number of variables. GFACTOR(EXP) factors polynomials over the Gaussian integers.

For example,

```
(C5) FACTOR(X**6+1)@

                  2         4    2
(D5)            (X  + 1) (X  - X  + 1)
```

SQFR(EXP) is similar to FACTOR except that the polynomial factors are "square-free" that is, have no multiple roots. This algorithm, which is also used by the first stage of FACTOR, utilizes the fact that a polynomial has in common with its nth derivative all its factors of degree > n. Thus by taking derivatives with respect to each variable in the polynomial, all factors of degree > 1 can be found. Several special cases are also factored, including the removal of polynomial contents.

PARTFRAC(EXP,VAR) expands the expression EXP in partial fractions with respect to the main variable, VAR. The algorithm employed is based on the fact that the denominators of the partial fraction expansion (the factors of the original denominator) are relatively prime. The numerators can be written as linear combinations of denominators, and the expansion falls out.

```
(C6) PARTFRAC(X/(X**2-1),X)@

               1         1
(D6)        ------- + -------
            2 X - 2   2 X + 2
```

3.2. Contagious CRE Commands

The above commands represent no new capabilities; MATHLAB (29) has almost identical facilities, although its internal equivalent of our CRE's is less efficient for sparse polynomials. Other systems, by limiting their universe of discourse to canonical representations, make the explicit RATSIMP commands unecessary. Nevertheless, an algorithm equivalent to RATSIMP must be present in order to maintain the canonical representations during a computation.

The commands in this and the following sections represent significant departures from the usual use of rational function routines.

RAT(EXP) is indistinguishable on command level from RATSIMP; however, RAT _leaves_ its internal result in rational function (CRE) form, so that operations used by the rational function commands described here can be more rapidly performed on it. Furthermore, any time the user adds to or multiplies by a CRE, the result is a CRE. That is, the CRE form is "contagious." This enables a user to easily force his entire calculation to be done in CRE form by converting one of his inputs into CRE by simply multiplying by RAT(1). Some problems require excessive amounts of storage and/or time if intermediate results are converted back into prefix form at each step of the calculation. The RAT facility, by being integrated into the simplifier, permits a user to compose a program and try it out (without any changes) on ordinary prefix form arguments _or_ on CRE arguments. In this manner it is simple to compare the timing of "general" versus CRE methods on the same task. This very often demonstrates that CRE methods, when appropriate, are much faster.

RATDISREP(EXP), which appears to do nothing on the command level, changes its argument from rational function form (CRE) to ordinary MACSYMA form. This is sometimes necessary in order to use some of the other MACSYMA commands. If RATDISREP is not given a CRE for an argument, it does nothing.

3.3. The Rational Coefficent Program

RATCOEF(EXP,PART) returns the coefficient, C, of the expression PART in the expression EXP. C will be free (except possibly in a non-rational sense) of the variables in PART. If no coefficient of this type exists, zero will be returned. RATCOEF will give reasonable answers to reasonable requests, and will often produce reasonable answers to poorly stated requests. Generally, when PART includes a "+" or a "/", results may seem odd. (see lines D7, D8, D10, and D11 in the examples to follow). Since EXP is rationally simplified before it is examined, coefficients may not appear quite the way they were envisioned. The effect of RATCOEF should be clarified by the following examples.

```
(C1) S:A*B*X**2+B*X+2*X+5@
                 2
(D1)       A B X  + B X + 2 X + 5

(C2) RATCOEF(S,X)

(D2)                B + 2

(C3) RATCOEF(S,A*B)@
                     2
(D3)                X

(C4) RATCOEF(S,B)@
                   2
(D4)             A X  + X

(C5) RATCOEF(S,2*X)@
                 B + 2
(D5)             -----
                   2
```

```
(C6) RATCOEF(S,B/2)@
                          2
(D6)               2 A X   + 2 X

(C7) RATCOEF(A*X+B*X+C,A+B)@
(D7)                      X

(C8) RATCOEF(3*A+2*B,A+B)@
(D8)                      2

(C9) RATCOEF(S,-A)@
                             2
(D9)                     - B X

(C10) RATCOEF((A*B+C)/D, B/D)@
(D10)                      A

(C11) RATCOEF(3*A/D+A/D**2, A/D**2)@
(D11)                      0
```

Let us first define RATCOEF(EXP,PART) where EXP is a polynomial and PART has the form v^k for v a variable, k a number. This case is clear: we expand EXP as a CRE, and pick off the coefficent of v^k. If there is no occurrence of v^k, the coefficent is 0. If EXP is not a polynomial, but a ratio of polynomials, then we must make a decision about how to treat occurrences of v in the denominator.

Let EXP =num/denom, where num = $\sum_i a_i v^i$. If the coefficient of v^i, namely a_i, is zero or if a_i/denom depends on any variable in the original PART, then the response is zero. Otherwise the response is a_i/denom.

RATCOEF of a product can be defined recursively as follows. Consider RATCOEF(EXP,PART). If PART = $v_1^{n_1} * v_2^{n_2} * \ldots * v_k^{n_k}$, then RATCOEF(EXP,PART) =

$$\mathrm{RATCOEF}(\mathrm{RATCOEF}(\mathrm{EXP}, v_k^{n_k}), v_1^{n_1} * \ldots * v_{k-1}^{n_{k-1}}).$$

If PART = A/B then RATCOEF(EXP,PART) = RATCOEF(EXP*B,A).

If PART = -A, RATCOEF(EXP,PART) = RATCOEF(-EXP,PART).

If PART = $\sum_i A_i$ (possibly after removing multipliers, as above), then EXP is divided by PART with respect to the main variable in PART. If the quotient depends on any variable in the original PART, the response is zero. Otherwise the answer is the quotient.

The coefficient produced in this manner may depend, in the last case, on the ordering of the variables within EXP. For example, the coefficient of (Y+Z)X in Z^2X^2+(Y+Z)X+A is clearly 1. The similar problem of finding the coefficient of XZ+XY in X^2Z^2+XZ+XY+A yields the answer 0, since X^2Z^2+XY+XZ+A divided by XZ+XY is XZ+1, with remainder $-X^2$YZ+A. The quotient depends on X, and thus the coefficient is taken to be zero.

This illustrates both the ability of the user to ask for coefficients of sums, and the ability of RATCOEF to sometimes answer correctly. We could have defined RATCOEF only for products, but it seems more in keeping with the spirit of an interactive system to avoid such restrictions on the user. Note that if the user were disappointed with the answer 0 to the above request, first executing RATVARS(X) would correct the situation.

In summary, RATCOEF will find the coefficient of PART when PART is a factor of the expression, or of some part of the expression such that the other factor has none of the same variables.

The returned value is in CRE form.

An alternative to RATCOEF is available in situations where its generality is not needed. The COEFF command can operate on CRE forms or on ordinary MACSYMA forms which have been expanded. COEFF(EXP,VAR,POWER) will extract the coefficient of VAR**POWER (where POWER may be 0) from EXP. COEFF returns a CRE form if and only if it is given a CRE form.

3.4. Simple Extensions to Rational Simplification

FULLRATSIMP(EXP) is an expanded version of RATSIMP which is recursive on the arguments of non-rational functions. It also removes zero exponents, and converts forms like (x**y)**z to x**(y*z). Although these last two operations are generally performed by the simplification program, FULLRATSIMP must repeatedly simplify the results of such transformations until no more rational simplifications can be made. FULLRATSIMP is no more time-consuming than RATSIMP if EXP is an algebraic expression with no non-rational functions. FULLRAT(EXP) is a program which operates similarly, but allows the user to specify a varlist as does RAT.

A more extensive expansion of the concept of global simplification is embodied in RADCAN. While FULLRATSIMP does not apply any identities concerning logs, radicals, and non-numeric exponents, RADCAN does.

RADCAN(EXP) converts the expression EXP into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, RADCAN produces a normal form; that is, all forms equivalent to zero are mapped into zero. For purely rational expressions, RADCAN is no more time-consuming than RATSIMP or FULLRATSIMP; however, for more general expressions including radicals, logs, and non-integer exponents, RADCAN can be quite expensive. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial-fraction expansions of exponents.

A description of the method, and proofs of the canonical properties of the RADCAN algorithm are discussed in chapter 4. Examples should, however, give a rough feel for the capabilities of RADCAN. (% always refers to the just-previously displayed expression, %E is the base of the natural logarithms):

```
(C1) SQRT(98)@
(D1)                  SQRT(98)

(C2) RADCAN(%)@
(D2)                  7 SQRT(2)

(C3) (SQRT(X**2-1))/(SQRT(X-1))@
                             2
                     SQRT(X   - 1)
(D3)                 ------------
                     SQRT(X - 1)

(C4) RADCAN(%)@
(D4)                  SQRT(X + 1)

(C5) (LOG(A**(2*X)+2*A**X+1))/(LOG(A**X+1))@
                      2 X      X
                 LOG(A    + 2 A  + 1)
(D5)             --------------------
                          X
                     LOG(A  + 1)

(C6) RADCAN(%)@
(D6)                    2

(C7) (%E**X-1)/(%E**(X/2)+1)@
                          X
                        %E  - 1
(D7)                    ---------
                         X/2
                       %E    + 1

(C8) RADCAN(%)@
                         X/2
(D8)                   %E    - 1
```

3.5. The RATSUBST (rational substitution) Commands

RATSUBST, or RATSUBSTn(A,B,C) where n = 1, 2, 3, 4 is a set of similar commands to substitute A for each occurrence of B in the expression C. In those cases where it is clear where B occurs, the result will correspond to the intuitive notion of substitution.

If B is an atom, occurrences of B are obvious. The action taken is simply substitution followed by simplification.

If B is a quotient, say b_1/b_2, then RATSUBSTn(A,B,C) is entirely equivalent to RATSUBSTn($A*b_2$,b_1,C).

If B is a product, all coefficients of powers of B can be detected in C by a technique similar to that used by RATCOEF. Hearn in (20) suggests this approach.) If B is a sum, we must define what we mean by an occurrence of an expression B in a polynomial expression C. (If C is not a polynomial, we can consider its numerator and denominator separately.)

If $C = \sum_i S_i B^i$, then B is said to occur in C with coefficient S_1 and exponent 1, coefficient S_2 and exponent 2, ..., and remainder S_0. If B occurs in such a fashion we wish to replace C by $\sum_i S_i A^i$. Unfortunately, finite power series expansions for an expression in terms of a non-atomic subexpression are not unique. For example, $C = x^2+3xy+y^2$ has (among others) the following expansions in (x+y):

1. $1*(x+y)^2 + 0*(x+y)^1 + x*y*(x+y)^0$

2. $1*(x+y)^2 + x*(x+y)^1 - x*(x+y)^0$

3. $1*(x+y)^2 + y*(x+y)^1 - y*(x+y)^0$

What is needed is a set of restrictions on the coefficients S_i so that the expansion is unique and appropriate to the problem at hand. This is the basic problem in substitution for simplification, and this solution is based on a set of heuristics for achieving what appear to be, in some instances, more desirable results than have been possible in the past. We will separate out only the highest power of B, and discuss at each stage (recursively on lower powers of B) the situation $C = SB^n + r$, where r contains the lower order terms.

As we have pointed out earlier in our discussion of RATCOEF, the ordering of variables is sometimes quite critical. "Sum"-hood, which is a property of a <u>form</u>, not of a <u>function</u>, sometimes depends on ordering. For example, xz+yx is a sum, but (z+y)x is (for purposes of RATSUBST) <u>not</u> a sum, but a product, although the two expressions are functionally equivalent.

Let B be a polynomial containing variables $v_1, v_2, \ldots, v_k$, where the highest power of each v_i is m_i. For all but condition 2 below, the only restriction on r, the remainder consisting of lower order terms, is that it has lower degree than C does in some particular variable (namely, the most important on the varlist that is also in B). The conditions below are embodied in the commands RATSUBST1,2,3, and 4, respectively. Their effects can best be gauged by frequent reference to the examples in figure 3.2 following. RATSUBST (without a number) is a quicker program than the numbered ones, which short-cuts many of the (rarely needed) conversions and re-conversions required for strictly following all the conditions.

Conditions

1. The highest power of some v_i in S that appears in B is less than the corresponding m_i.
2. The highest power of <u>each</u> v_i in S that appears in B is less than the corresponding m_i, and the highest power of each v_i in r that appears in B is less than the corresponding m_i.
3. S is a polynomial.
4. S contains no sum.

The value of n ranges from the highest possible (the ratio of the highest coefficient of some v_i in C which is also present in B, to the corresponding maximum coefficient of that v_i in B, namely m_i) to the lowest possible (when some v_i in B is no longer present in C to a power as high as it is in B, or 1.). To avoid the possibility of looping, occurrences of B in C are replaced, as found, by a special dummy variable, which is subsequently replaced by A. Cases in which B occurs in A (probably an error on the user's part) or where simplification of C results in new

Figure 3.2 Examples of RATSUBST (substitute argument 1 for argument 2 in argument 3)

Argument 1	Argument 2	Argument 3	RATSUBST Versions	RATSUBST(Arg_1,Arg_2,Arg_3)
A	XY^2	$X^4Y^8+X^4Y^3$	1,2	$X^4Y^3+A^4$
			3,4	AX^3Y+A^4
1	S+C	S	1,3,4	-C+1
			2	S
A	B(X+Y)	$B^2+BX+BY+1$	1,3,4	B^2+A+1
			2, with RATVARS(Y)	$BY+BX+B^2+1$
			2, with RATVARS(X)	$BX+BY+B^2+1$
			2, with RATVARS(B)	B^2+A+1
A	XY^2	X^2Y	1,2,3	X^2Y
			4, with RATVARS(X)	A^2/Y^3
			4, with RATVARS(Y)	X^2Y
A	X+Y	(X+Y)(Z+W)	1,2,3	(Z+W)A
			4	(Z+W)y+(Z+W)X
1	I^2	I^4+1	1,2,3,4	2

occurrences of B can be treated with repeated calls to RATSUBST. This can be easily programmed in MACSYMA.

If C contains non-rational functions, substitution proceeds on the arguments of the non-rational functions, recursively. Thus A, B, and C need not be rational expressions.

By noting when B has non-rational components (e. g., e^{x}, or $x^{1/2}$), RADCAN can be called on B and C, and they can be left in a special expanded format, which tends to reflect more clearly the similarities of the two expressions. Thus RATSUBST(A,E**X,E**(2*X)) is A**2.

An example of an extension to the RATSUBST framework might serve to illustrate its generality. If there is a canonical ordering on all expressions submitted to RATSUBST, and on all intermediate expressions, then a RATSUBST5 could be programmed with the following condition:

5. $SA^{n} + r$ has a lower canonical order ("is simpler") than $SB^{n} + r$.

By using the RATSUBST commands selectively, such substitutions as $\sin^{2}(x) + \cos^{2}(x) \to 1$ can be performed more nearly in the sense in which they are intended. If one RATSUBST command does not do the job, perhaps another will.

3.6. The SOLVE Program

The SOLVE command in MACSYMA uses several techniques for solving for a given variable in an equation. Each of these techniques is open to extension in a straightforward manner. The roots and their multiplicities are available to other programs, and are used as building blocks for more complicated facilities, such as contour integration.

The format of the SOLVE command is:

SOLVE(<u>equation</u>, <u>variable</u>)@

where the <u>equation</u> may also be an expression (which is assumed to be set equal to zero), or a set of polynomial equations linear in some set of variables. This last case is a straight-forward problem in Gaussian elimination, and will not be discussed further here.

SOLVE(E,X) puts its first argument E, in radical canonical form, and attempts to factor it with respect to the <u>variable</u> X, and all non-rational functions in E containing X. Each factor is examined for being linear, quadratic, cubic, or biquadratic with respect to X and the non-rational functions containing it. If the factor is of degree five or more, then it is considered unsolvable unless it is of the form $a(F(X))^n + b$ in which case the n nth roots of a/b are generated, and the n equations $F(x)-(a/b)^{1/n} = 0$ are solved. Any remaining unsolved factors and their multiplicities are put on a list which is returned along with the roots.

Linear terms of the form F(X)-C are examined to see if C, the constant term, is actually free of elements containing X; if so, USOLVE is called. Otherwise the term is added to the list of unsolved factors. USOLVE knows the inverses of SIN, COS, ASIN, ACOS, TAN, ATAN, LOG, etc. and powers of e. It could be extended to other functions. Once the inverse has been applied, a new equation results. It may be of the form X = FINVERSE(C), in which case the term has been solved, or it may be of the form G(X) = FINVERSE(C), in which case SOLVE is called again. This recursive algorithm allows for solution of, for example, SIN(COS(X)) = 0 for X.

The quadratic (cubic, biquadratic) formula is applied to quadratic (etc.) factors, and the same sort of recursive treatment as described above is used if the equation is, for example, quadratic in SIN(X) instead of X.

The simplification done by the quadratic (etc.) routines is of some interest, in that the roots in the formulae are simplified by a special program (SIMPNRT) which takes out perfect n*k powers of a kth root. (i.e. even powers in a square root, multiples-of-three powers in a cube root, etc.) Thus SQRT(8) is simplified to 2*SQRT(2). SIMPNRT calculates a square-free factorization of the radicand, and takes appropriate multiple factors, if any, outside the radical.

The following examples illustrate the capabilities of SOLVE:

```
(C1) SOLVE(Y**(2*X)-3*Y**X+2=0,X)@
SOLUTION

(E1)                        X = 0

                              LOG(2)
(E2)                      X = ------
                              LOG(Y)

(D2)                         (E1,E2)

(C3) A:X**2-12*X+3@
                         2
(D3)                    X  - 12 X + 3

(C4) SOLVE(SIN(A)**2-5*SIN(A)+3,X)@

SOLUTION

                                   5   SQRT(13)
(E4)    X = 6 - SQRT(ARCSIN(- - --------) + 33)
                                   2      2

                               5   SQRT(13)
(E5)    X = SQRT(ARCSIN(- - --------) + 33) + 6
                               2      2

                               SQRT(13)    5
(E6)    X = 6 - SQRT(ARCSIN(-------- + -) + 33)
                                  2        2

                           SQRT(13)    5
(E7)    X = SQRT(ARCSIN(-------- + -) + 33) + 6
                              2        2

(D7)              (E4,E5,E6,E7)
```

```
(C8) SOLVE(ARCSIN(COS(3*X))*(F(X)-1),X)@
SOLUTION
```

(E8) $$X = \frac{ARCCOS(0)}{3}$$

THE ROOTS OF

(E9) $$F(X) = 1$$

(D9) (E8,E9)

```
(C10) SOLVE(5**X=125,X)@
```

(D10) $$X=3$$

Note that SOLVE has taken advantage of radical approaches but is still able to step back and treat fairly general expressions. In order to use the "radical" polynomial factoring program, it uses RADCAN to expand unlikely-looking expressions into polynomials. Thus the expression $Y^{2X}-3Y^{X}+2$ in C1 is expanded into a polynomial in Z, where $Z=Y^{X}$ (actually $Z=e^{X\log(Y)}$), which is then factored into (Z-1)*(Z-2). By setting each of these factors equal to zero, the following sequence of steps is followed:

$e^{X\log(Y)}-1 = 0$ is converted by USOLVE to

$X\log(Y) = \log(1)$ which the simplifier changes to

$X\log(Y) = 0.$

SOLVE is called recursively, and factors this; SOLVE throws out the log(Y) factor since it does not depend on X, and the factor "X" is recognized as a linear expression of the form aX+b where a=1 and b=0, which has solution X=-a/b, or in this case, X=0. The other root is handled in an analogous fashion.

3.7 Conclusions

By using several distinct approaches to attack different phases of the same problem, particularly powerful algorithms can be obtained. Although _ad hoc_ procedures can, in some circumstances, yield similar results in other systems for algebraic manipulation, MACSYMA's SOLVE, RATSUBST, RATCOEF, and FULLRATSIMP commands provide a generality and power not available elsewhere.

These foundation blocks allow the building of new facilities. SOLVE is used by programs which find limits, compute definite integrals, and expand functions in power series. RATCOEF is used by the semantic pattern matching subsystem. FULLRATSIMP is used by RADCAN, and RADCAN, in turn is used by SOLVE. RADCAN, furthermore, can be used as the basis for implementing the Risch (41) integration algorithm.

Chapter 4 - Simplification of Radical Expressions

4.1. Introduction

The simplification of algebraic expressions is a many-faceted problem. On one hand, all of the work in simplification (and algebraic manipulation in general) is circumscribed by the work of Richardson (39), which shows that for a sufficiently large class of expressions the question of zero-equivalence is undecidable. Furthermore, some researchers (e.g. Fenichel (16), Moses (34), (35)) argue that (regardless of computability) the concept of simplicity has no generally acceptable meaning. On the other hand, Brown (3), Caviness (5), (6) and others show that within certain classes of expressions the rigorous notions of canonical forms and zero-equivalence tests can serve as useful measures of simplicity. For a survey of these and other attitudes and achievements in algebraic simplification, see Moses (34).

The importance of the simplification problem in algebraic manipulation is quite basic: A "simplified" expression generally exhibits its most significant properties in a systematic fashion. This can make mechanical (or human) processing of the expression much easier.

This chapter discusses simplification algorithms for the class of radical expressions. These are, roughly speaking, ratios of multivariate polynomials, some of whose "variables" are nth roots of polynomials. These expressions commonly occur in representing roots of algebraic equations in several variables, and are rarely treated adequately in algebraic manipulation systems. The only current alternative to the treatment we provide in MACSYMA (and describe here) is a computationally impractical procedure suggested by Caviness in (5).

In the following sections we will proceed to define the problem of simplification of radical expressions in more exact terms and contrast our approach with that of others who have had similar goals. In sections 4.2 and 4.3 we discuss basic concepts and define the class of radical expressions more precisely. In section 4.4 we survey previous algebraic approaches to radical simplification and a promising alternative, zero-equivalence testing.

Sections 4.5 and 4.6 discuss the specific methods we developed for MACSYMA. Section 4.7 proves some properties of the simplified form; 4.8 discusses the canonical form implications of this work; 4.9 points to other related efforts in MACSYMA, and 4.10 summarizes its usefulness.

4.2. Basic Concepts

Following Caviness (6), to be given a <u>class of expressions</u> $\mathcal{E}$ means to be given rules, such as a Backus-Naur Form (BNF) grammar, for determining the well-formed expressions in the class. The expressions must be formed from a finite set of atomic symbols, a subset of which must be designated as <u>variables</u>. A member of $\mathcal{E}$ not containing any variables is a <u>constant</u>.

Expressions are interpreted as functions over the domain $\mathcal{D}$ of constants.

If R and S are members of an expression class $\mathcal{E}$, R is said to be <u>identical</u> to S if R and S are the same string of atomic symbols. This relation is denoted by $R \equiv S$. R and S are said to be functionally equivalent or simply <u>equivalent</u>, if for all assignments of values in $\mathcal{D}$ to their variables for which they are defined, they are equal. This relation is denoted by $R = S$. Of course $R \equiv S$ implies $R = S$.

One concept related to simplicity which is of particular usefulness is that of a canonical form.

<u>Definition</u> 4.2-1 A <u>canonical form</u> algorithm f for a class of expressions $\mathcal{E}$ is a mapping from $\mathcal{E}$ into $\mathcal{E}$ such that for all R, S in $\mathcal{E}$,

(i) $f(R) = R$

(ii) $R = S \implies f(R) \equiv f(S)$

<u>Definition</u> 4.2-2 A <u>zero-equivalence test</u> algorithm f for a class of expressions $\mathcal{E}$ is a function from $\mathcal{E}$ into $\mathcal{E}$ such that for all R, S in $\mathcal{E}$,

$$f(R) \equiv 0 \iff R = 0$$

The <u>constant problem</u> consists of determining the zero-equivalence of an expression containing no variables.

A third concept, that of a normal form, is used by Caviness.

Definition 4.2-3 A normal form algorithm f has the same strong property of the zero equivalence test algorithm, but has the additional properties

(i) $f(R) = R$ (whether or not $R = 0$)

(ii) $f(R)$ fits a "pattern."

This pattern concept is not generally defined but can be clarified in a particular situation. For example, Brown's "simplified" form for rational exponential expressions (3) is normal.

A more useful concept than the normal form is that of a regular elementary (or just regular) form as introduced by Risch (40).

Definition 4.2-4 If $\theta = e^f$ or $\log(f)$ for $f \in \mathcal{E}$ and is transcendental over $\mathcal{E}$, θ is said to be a monomial over $\mathcal{E}$. If θ is a root of a polynomial with coefficents in $\mathcal{E}$ irreducible over $\mathcal{E}$ and of degree $d(\theta)$ at least 2, then θ is said to be non-trivial algebraic over $\mathcal{E}$. Let $\mathcal{S} = \mathcal{E}(\theta_1, \ldots, \theta_n)$, that is, $\mathcal{E}$ with n normal algebraic extensions. $\mathcal{S}$ is regular elementary over $\mathcal{E}$ iff each θ_k is a monomial or is non-trivial algebraic over $\mathcal{E}(\theta_1, \ldots, \theta_{k-1})$. An expression $g \in \mathcal{S}$ is regular elementary if the degree of g in any algebraic θ is less than the defining degree of θ, $d(\theta)$.

Clearly if g contains θ to some higher degree than $d(\theta)$, reductions can be made to remove this condition. Any $g \in \mathcal{S}$ fits the implied "pattern" of a rational function (ratio of two polynomials) because any expression is rational once a regular field description is found. Thus the vagueness of the normal form is removed.

Section 4.8 returns to exponential and logarithmic monomials briefly, but for the bulk of this chapter we will be concerned only with the non-trivial algebraic extensions.

A class of expressions is called a canonical (normal, regular) class or is said to possess a canonical (normal, regular) form if there exists a canonical (normal, regular) form algorithm for it. It is conventional to assume that if $R = 0$, then $f(R) \equiv 0$.

4.3. Radical Polynomials and Expressions

Radical polynomials, $\mathcal{P}$, are formed from

(i) the integers

(ii) the variables $x_1, x_2, \ldots, x_N$ (collectively called X)

(iii) the operations of addition, subtraction, multiplication

(iv) the un-nested operation of exponentiation to a positive rational number.

Radical expressions $\mathcal{R}$, are formed from radical polynomials with the added operation of division.

Examples of radical expressions are

$$\frac{1}{x_1^{1/3} + x_2^{5/4}} \quad \text{and} \quad \frac{(-x_1 + 4)^{1/2}}{(x_3/2 + x_4)^{2/5}}$$

The expression

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

is not in the class $\mathcal{R}$ because of the nested exponents.

This definition is a slight generalization of one given by Caviness (5) in that it allows more than one variable.

The interpretation given to radical expressions is one which we believe corresponds, in its implications, to the most common valid usage. As does Caviness, we interpret radical expressions as algebraic functions: For each expression $E \in \mathcal{R}$, there must exist an irreducible polynomial $P(z,X)$ such that $P(E,X) = 0$. Caviness notes the necessity of simplifying $(x+1)^{1/2}(x-1)^{1/2} - (x^2 - 1)^{1/2}$ to 0 in spite of the following situation:

If we let y_1 be a root of $y^2 = x+1$

y_2 be a root of $y^2 = x-1$

y_3 be a root of $y^2 = x^2-1$

then $y_1 y_2 - y_3$ can just as easily be $\pm 2\,(x^2-1)^{1/2}$ as 0. Perhaps a complete answer would include all these possibilities. Any interpretation "consistent" (but unspecified by Caviness) should produce 0, since admitting the other possibilities is tantamount to declaring all algebraic extensions transcendental over the base field (and therefore subject to no simplifications at all). Caviness requires that some branch of multiple valued roots be chosen. Thus $(x^2)^{1/2}$ is either x or -x, depending on the branch of the square root chosen. We differ from Caviness on this point: a particular branch, the positive real branch, to be defined shortly, will be automatically chosen as the interpretation of the radical.

In general, single-valued branches of radicals are not analytic everywhere, and hence their domains must be suitably restricted in either Caviness' or our interpretation.

We now define the particular interpretation of radicals which we use.

Definition 4.3-1 A polynomial p(X) is said to be positive if its leading (integer) coefficient is positive, when p is written in some canonical form. In such a case we shall write $p \geq 0$.

Definition 4.3-2 A polynomial or integer p is said to be square-free if it has no repeated factors (or roots).

If p is a positive square-free polynomial and m is a positive integer, then $p^{1/m}$ has a positive real interpretation

(<u>PRI</u>). All other instances of radicals will be reduced to this case in defining <u>their</u> interpretations.

<u>Definition</u> 4.3-3 If p is a positive square-free polynomial and m is a positive integer, then $p^{1/m}$ has the <u>positive real interpretation</u> (<u>PRI</u>) defined as follows:

case 1: p is a positive square-free integer. $p^{1/m}$ is interpreted as the positive real mth root of p.

case 2: p is a polynomial in one variable, say x. The coefficient of the leading term in p(x) is a positive integer, so that there exists a real number L such that for all $\xi > L$, $p(\xi)$ is a positive real number. By case 1, for each ξ, $(p(\xi))^{1/m}$ has a PRI. The PRI for $(p(x))^{1/m}$ is then this branch of the solution to $z^m - p(x) = 0$ which has positive real values for $x > L$.

case 3: M-variable polynomial ($M > 1$). Assume a recursive polynomial representation as in chapter 3. It is possible to fix values for all but the main variable, say x, such that the coefficient of the leading term in x is positive. Then a PRI for p(x) is defined as in case 2. For example, consider the 3-variable polynomial

$$p(x,y,z) = (y^2 - (z + 1)\ y)\ x^2 - 3\ x\ y + 2$$

choose $z = 0$ (arbitrary)

choose $y = 2$ to make $y^2 - (z + 1)\ y$ positive (namely 2)

then for $x > 2$, p has a PRI.

Now let us define interpretations for more complicated radicals. We can assume that any radicand is at worst a rational expression $p(X)/q(X)$ where $p(X)$ and $q(X)$ are relatively prime polynomials in canonical form, and $q(x)$ is positive and nonzero.

$(p(X)/q(X))^{n/m}$ is interpreted as the ratio of the interpretations of $p(X)^{n/m}$ and $q(X)^{n/m}$. Thus all radicands can be assumed to be polynomials.

If $p(X)$ is not positive, $p(X)^{n/m}$ is interpreted through the use of a primitive root of unity $\omega_{2m} = e^{i\pi/m}$ as $(\omega_{2m})^n$ times the interpretation of $(-p(X))^{n/m}$. Thus all radicands can be assumed to be positive.

If $p(X)$ is positive but not square-free, it is easy to prove that $p(X)$ may be factored into positive square-free factors. Thus if

$$p(X) = \prod_{i=1}^{k} (p_i(X))^i,$$

the interpretation of $p(X)^{n/m}$ is the product of the interpretations of

$$(p_i(X))^{in/m}$$

for $i = 1, \ldots, k$. Thus all radicands can be assumed to be square-free.

If $n \geq m$, then for $n = q\,m + r$, for $0 \leq r < m$, $p(X)^{n/m}$ is interpreted as the product of $p(X)^q$ and the interpretation of $p(X)^{r/m}$. Thus we can assume $n < m$.

If n/m is not in lowest terms, it can be converted to lowest terms. Thus we may assume the greatest common divisor of n and m is 1.

If $n = 0$, $p(X)^{n/m}$ is interpreted as 1.

If $n > 1$, $p(X)^{n/m}$ is interpreted as the nth power of the interpretation of $p(X)^{1/m}$. Thus all interpretations are based on definition 4.3-3.

According to the PRI interpretation, $(x^2)^{1/2}$ means x and $(20)^{1/2}$ means $2\ 5^{1/2}$, a positive number. We believe this corresponds to the most common usage.

We should point out that some of the transformations used for the algorithms to follow are the basis for innumerable false "proofs" These proofs are generally based on inconsistent interpretation of radicals, and will not occur in our usage. For example, using

$$a^{b} = a^{b}\ a^{c} \quad \text{and} \quad (ab)^{c} = a^{c} b^{c}$$

we can "prove"

$$1 = 1^{1} = 1^{1/2}\ 1^{1/2} = 1^{1/2}\ (-1)^{1/2}\ (-1)^{1/2} = (-1)^{1/2}\ (-1)^{1/2}$$

$$= (-1)^{1} = -1.$$

Since MACSYMA will impose positive real interpretations on radicals, it will not factor 1 into (-1)(-1) and fallacies of this sort will not occur.

It is interesting to compare our interpretation of radicals with one which is, some would argue, most common, namely, that the expression $(x^2)^{1/2}$ means $|x|$. For example, the modulus of $c = a+bi$ is written as $|c| = (a^2 + b^2)^{1/2}$; if $b = 0$, we are left with the convention that $|a| = (a^2)^{1/2}$. Since this holds only when a assumes real values, and the square is computed <u>before</u> the square root, the usage is, in fact, consistent with a positive real interpretation. In general $x^{1/2}$ meaning $|x^{1/2}|$ is restricted to the domain of non-negative real x.

In summary, there are (at least) three interpretations for radicals.

1. Caviness', which does not choose a branch of the algebraic function;
2. Ours, which chooses the PRI;
3. The "common" square root which implies absolute value with restricted domain.

The last two are equivalent on a restricted domain, and the first two are equivalent up to the choice of a branch. Computationally, interpretation (2) has a distinct advantage over either of the others in that it is consistent over a larger domain than (3), and does not unnecessarily involve arbitrary roots of unity as in (1).

4.4. Comparisons with Previous Work on Radical Expressions

4.4.1 Algebraic approaches

Caviness proves in (5) that for an expression $E \in \mathcal{R}$, "E = 0" is decidable. Unfortunately, the application of his constructive proof relies on an impractical (and largely unnecessary) computation. The problem lies in the difficult task of factoring over algebraic extensions of a polynomial ring. Caviness points out that the need for factoring is a result of the lack of irreducibility criteria for the radical expressions. He develops a few; we extend his results and show that satisfactory results can generally be obtained without any factoring.

The results here appear to conform more closely to intuitive notions of simplification than does Caviness'. More important, they are far more easily computable, since the only calculation needed is that of the greatest common divisor (gcd) of multivariate polynomials with integer coefficients.

The difficulty in Caviness' approach, from a practical standpoint, is his interpretation of radicals as written in an expression. His approach can be most easily seen in Van der Waerden (47), section 36. Briefly, given any finite number of radicals, an algebraic extension to the field of rational expressions may be constructed to which all the radicals belong.

Each expression in this field will have a unique representation within the field. The construction of this field takes a finite number of steps. Given a radical expression E, it is only necessary to explicitly construct a suitable field which contains E, and find the unique representation of E in that field. This representation can always be found in a finite (but possibly large) number of steps. This does not produce a canonical form since there are an infinite number of fields which will contain E, and the representation of E in the different fields may differ. However, given two non-identical equivalent expressions, a field may be constructed which contains them <u>both</u>, and in which they are identically represented.

An unpublished report by S. L. Kleiman (25) proposes a canonical form for rational expressions in several algebraically dependent variables (e.g. $f(x,y)$ where $y^2+x=1$). The procedures he suggests have never been implemented, nor would they be computationally efficient; nevertheless, his discussion of the problems involved is quite thorough. He avoids the question of interpretation of radicals by introducing new variables which satisfy certain polynomial equations.

By contrast, our approach (by applying irreducibility criteria and simplifications) is to produce a field which allows all permissible simplification to be performed. Many, but not all expressions are mapped into canonical forms by this approach. Those not in canonical form are easily distinguishable from the others by the presence of roots of -1.

4.4.2 Zero-equivalence tests

It has been shown by Richardson (39) and Johnson (24) that zero-equivalence tests for the class of expressions treated here (and other, larger classes) can be reduced to the "constant" problem; that is, all references to variables can be removed in determining zero-equivalence, assuming the expressions are totally defined over the domain of interest. The constant problem is non-trivial, since very little is known about such specific constants as $e+\pi$ or e^{e}; also if $(-1)^{1/4}$ stands for a primitive fourth root of -1, for example, $e^{i\pi/4}$

$$(-1)^{1/4} - (-1)^{3/4} - 2^{1/2} \tag{4.4.2-1}$$

is a constant which is 0, but not obviously so. The constant problem does not concern us here because it is decidable for radical constants by methods used by Caviness, while using our interpretation of radicals it only crops up with roots of -1.

We discuss zero-equivalence tests in some detail because they serve, in some instances, as a potentially very powerful tool in simplification. In some cases decisions as to zero-equivalence may be all that is needed. Secondly, given a zero-equivalence test, we can produce a canonical form algorithm in the following way: Assume we wish to find a canonical form algorithm for a class of expressions but only have a zero-equivalence test over that class. We can produce, in lexicographic order, all legal members of the class (say, in size place, up to and including the length of the expression f under consideration). The first generated expression g, such that f-g is 0, is the canonical form. Although this is clearly unsatisfactory as a practical computational approach, it does provide some theoretical unity to the concepts. Furthermore, research along the lines of the approach illustrated below promises to provide especially useful insight into the ways expressions can combine. This is particularly relevant for classes of expression much larger than $\mathcal{R}$.

Let us illustrate the approach of Johnson's (24) zero-equivalence test. Let $\mathfrak{F}$ consist of expressions of the form A^B, A a rational function in one variable, x, and B a rational number. Let $\mathfrak{E}$ consist of products of elements of $\mathfrak{F}$. Radical polynomials are sums of elements of $\mathfrak{E}$. Define the function L(u) = (du/dx)/u for $u \in \mathfrak{E}$. Any element u of $\mathfrak{E}$ is called an eigenvector (of the derivative operator) whose eigenvalue is L(u). Eigenvalues are always rational functions of x, since

$L(A*B) = L(A) + L(B)$

$L(A/B) = L(A) - L(B)$

$L(A^B) = B*L(A)$ B a rational number

$L(x) = 1/x$

$L(B) = 0$ B a rational number

$L(A+B) = (dA/dx + dB/dx)/(A + B)$ A, B rational functions of x.

Since we can always decide whether or not a rational function of x is zero, we can always tell whether or not L(u) for $u \in \mathcal{E}$ is zero. The basis for the algorithm is the fact that for u equivalent to a constant, L(u) = 0.

Suppose we can decide if a constant is zero. Assume we have a set of eigenvectors u_i, i= 1, ..., n (and have calculated their eigenvalues by the above rules). We may decide if

$$S = \sum_{i=1}^{n} u_i$$

is zero as follows:

STEP 1: If n = 1 and $L(u_1)$ is not zero, $S \neq 0$. Otherwise S is a constant. By assumption we can decide if the constant is zero. Return.

STEP 2: If n > 1 then consider

$$T = \sum_{i=1}^{n} u_i/u_n .$$

T is a sum of eigenvectors (whose eigenvalues are known) and whose last term is 1. Test T for being equivalent to zero (see below).

STEP 3: If $T \neq 0$, $S \neq 0$. Return.

STEP 4: If T = 0, then $K - S/u_n$ is a constant. By assumption we can test if a constant is zero. If $K \neq 0$, then $S \neq 0$. If K = 0, S = 0. Return.

We must now explain step 2. Consider

$$T = \sum_{i=1}^{n-1} u_i/u_n + 1$$

If any of the eigenvalues of u_i/u_n, i = 1, ..., n-1 are zero, delete them from T. (The eigenvalue of 1 is 0, and so 1 is deleted.) If all the eigenvalues are zero, T = 0. Otherwise T is a sum of at most n-1 eigenvectors (with known eigenvalues) so this algorithm can be applied recursively to determine whether or not T = 0.

An Example:

Consider

$$S = 2\,x^{1/2} - (4x)^{1/2}$$

The eigenvalues for $2\,x^{1/2}$ and $-(4x)^{1/2}$ are the same, namely $1/(2x)$. In step 2 of the algorithm we set

$$T = \frac{2\,x^{1/2}}{(4\,x)^{1/2}} + 1$$

The eigenvalue for $(2\,x^{1/2})/(4x)^{1/2}$ is $1/(2x) - 1/(2x) = 0$. This implies that T (and thus S/u_n) is a constant. The particular constant value of S/u_n must be determined by other means. Such means should reveal that S/u_n is in fact 0.

Several important facts should be noted. First, the problem of deciding when a constant is zero is not solved. Second, if an expression is <u>not</u> zero, a "simplified" equivaluent expression is not generally produced. Third, the class of eigenvectors can be extended to other expressions (e.g. e^A, for A rational in x).

Richardson's (39) scheme, which is somewhat more complicated, does, however, allow for composition of functions. His method has been extended to a large class of functions defined by first order non-linear differential equations by Moses, Rothschild, and Schroeppel (36).

Zero-equivalence tests, although an area of theoretical importance, cannot at present be considered as useful as some other notions of simplification, especially canonical forms, within the context of algebraic manipulation systems. We are hopeful however, that research in this direction will produce useful information for algebraic manipulation system designs, and have for this reason included this section.

4.5. Simplified Radical Polynomials

In this section we present two closely related simplified forms for a radical polynomial. Each looks like a multivariate polynomial, some of whose variables are radicals.

Let v_k, $k = 1, \ldots, N$ be a set of radicals of the form

$$v_k = (p_k)^{1/m_k} \tag{4.5-1}$$

<u>Definition</u> 4.5-1. In form (1), each m_k is an integer > 1 and each p_k is a positive square-free integer or polynomial with no (integer or polynomial) factors in common with any p_j, $j \neq k$.

<u>Definition</u> 4.5-2. In form (2), the p_k are distinct positive prime numbers or positive, primitive, irreducible (over the integers) polynomials. Form (2) is a special case of form (1).

With this definition of $\{v_k\}$, a simplified radical polynomial has the form:

$$(4.5\text{-}2) \qquad Q(v_k) = \sum_{i=0}^{m_k - 1} a_{i,k} v_k^i$$

where each $a_{i,k}$ is an integer, a polynomial, or a simplified radical polynomial in other radicals v_j, $j < k$.

For example,

$$2^{1/2} + 3^{1/4} + 6^{1/2} + (x_1 x_2)^{1/2}$$

can be represented as a form (1) simplified radical polynomial by:

$$2^{1/2} + 3^{1/4} + 2^{1/2} (3^{1/4})^2 + (x_1 x_2)^{1/2}$$

and as a form (2) simplified radical polynomial by:

$$2^{1/2} + 3^{1/4} + 2^{1/2} (3^{1/4})^2 + (x_1)^{1/2} (x_2)^{1/2}$$

There are some radical polynomials which cannot be represented in either of the above simplified forms, e.g.

$(-1)^{1/2}$. To allow for representing such expressions, we define forms (1') and (2'). In these, a single primitive nth root of unity, ω_n, may be adjoined to the set $\{v_k\}$. With this addition, $(-1)^{1/2} = \omega_4$.

We initially excluded ± 1 from the set $\{p_k\}$ because expressions involving roots of unity cannot be handled as authoritatively (by the methods we use) as other expressions. By agreeing that any root of +1 is 1, we are left only excluding roots of -1. By writing these roots in terms of other expressions, even these symbols may be effectively removed. For example, the expression (4.4.2-1) mentioned earlier will not be reduced to zero automatically. On the other hand, MACSYMA allows $e^{i\pi/4}$ or $2^{1/2}(1+i)/2$ to be substituted for $(-1)^{1/4}$, and would then simplify the resulting expression to 0.

Since many expressions can be represented in the forms (1) and (2), it is significant that each of these is the basis for a canonical form. Furthermore, algorithms for

(a) Converting any radical expression into a ratio of radical polynomials

and (b) converting a radical polynomial into forms (1) or (2) are relatively straightforward, given programs for factoring polynomials and computing polynomial greatest common divisors. Details of such algorithms are contained in the next section.

4.6 Algorithms

4.6.1 Removing quotients from radicals

Let us first consider the radicands in a given radical expression. A radicand, if not already in the form of the ratio of two polynomials with integer coefficients whose gcd is 1 may be straightforwardly transformed into such a form by rational simplification. Chapter 3 describes how this can be done by the RATSIMP program.

A and B below are relatively prime polynomials over X, with integer coefficients, and r is a rational number.

If $r > 1$, say $r = s + q$, for $0 \leq q < 1$, then

$$\left(\frac{A}{B}\right)^r = \frac{A^s}{B^s} \cdot \frac{A^q}{B^q}$$

Otherwise,

$$\left(\frac{A}{B}\right)^r = \frac{A^r}{B^r}$$

4.6.2 Producing a ratio of radical polynomials

To transform the expression into a ratio of radical polynomials requires one further step. The expression must be expanded over a common denominator into the ratio of two polynomials. RATSIMP can be used for this purpose.

4.6.3 Simplifying radical polynomials

The algorithm below produces a simplified form for the radical expression by treating its numerator and denominator as radical polynomials. Strictly speaking, if the result includes powers of roots of unity, we do not consider it truly simplified.

STEP 1: Make a list of all radicands in the expression $\{S_i\}$, $i = 1, \ldots, N$.

STEP 2: For $i = 1, \ldots, N$, factor S_i into a product of positive prime numbers and a polynomial in canonical form (such as that produced by RATSIMP). If the polynomial has a negative leading coefficient, add -1 to the list of factors, and multiply the polynomial by -1.

If form (2) is required, factor the polynomial into irreducible factors.

If form (1) is required, factor the polynomial into square-free factors (the SQFR command of Chapter 3). This uses the operations of differentiation and polynomial GCD only, and is much faster than full factorization.

STEP 3: Now for each $g_{i,k} = \gcd(S_i, S_k)$ $i > k$ such that $g_{i,k} \neq 1$, "factor" both S_i and S_k into $g_{i,k}$ and another factor which is computed by polynomial division.

STEP 4: Reduce powers of a common base to powers of the base to the lowest common degree of the radical powers. That is, if $2^{2/12}$ and $2^{1/2}$ are the only occurrences of the base 2, replace them by $2^{1/6}$ and $(2^{1/6})^3$ respectively. For some expressions, one of the bases will be "ω_n" in order to represent roots of -1 as powers of ω.

STEP 5: Rationally simplify the expression, considering each distinct radical as a "variable." Thus

$((2^{1/6})^3 + (2^{1/6}))/(2^{1/6})$ would be simplified to $(2^{1/6})^2 + 1$.

STEP 6: Simplify the resulting expression by the transformation

$(a^b)^c \dashrightarrow a^{bc}$ for $bc \geq 1$. Thus $(2^{1/6})^7$ would be simplified to $2\ 2^{1/7}$.

STEP 7: If step 6 has caused any changes, go to step 5.

STEP 8: Simplify $(a^b)^c$ to a^{bc}, bc a proper fraction in lowest terms: e. g. $(2^{1/6})^3$ is simplified to $2^{1/2}$.

STEP 9: (optional) Rationalize the denominator. (see section 4.6.4 below)

STEP 10: (optional) Combine products of radicals. E. g.

$(2^{1/3}\ 3^{2/3}) \dashrightarrow (18)^{1/3}$.

This algorithm terminates for any finite radical expression. Clearly all the individual steps can be done in a finite time. The only loop depends on step 6 causing a change in the expression. If we consider a recursive polynomial representation of either the numerator or denominator of the expression, step 6 can be applied once and only once for each of the different v_k in the polynomial. Thus the loop of steps 5, 6, and 7 will also terminate.

4.6.4 Rationalizing denominators

A radical polynomial, $Q(v_k)$, is a root of a polynomial $S(z)$ with rational (in fact, integral) coefficients. If ω_m is a primitive (m_k)th root of unity, the other $(m_k - 1)$ conjugate roots of S are

$$Q(\omega_{m_k} v_k), \ldots, Q((\omega_{m_k})^{m_k - 1} v_k).$$

The product of these other $m_k - 1$ roots, R_k, is a rationalizing multiplier in that $Q(v_k)\, R_k$ is free of the radical v_k. That is, radicals involving roots of p_k have been eliminated. This procedure can be repeated for each of the v_k, $k = N, \ldots, 1$, and all of the radicals can be eliminated.

Multiplying the numerator by

$$R = \prod_{i=1}^{N} R_i$$

completes the procedure in theory, but an additional task remains. A simple example illustrates the problem. Let us rationalize the denominator of

$$(4.6.4\text{-}2) \quad 1/Q = 1/(x^{2/3} + x^{1/3}).$$

Proceeding as above, we see that there is just one radical, namely $x^{1/3}$ in the expression. Thus $k = 1$, and $v_1 = x^{1/3}$. Then the denominator, Q, in terms of v_1 has the form $Q(v_1) = (v_1)^2 + v_1$. The conjugate roots, $\omega_3 v_1$ and $\omega_3^2 v_1^2$, when substituted in Q, produce two other polynomials. R, the product of these three polynomials (noting the simplification $(\omega_3)^3 = 1$) becomes:

$$(4.6.4\text{-}3) \quad x^{1/3} x + (\omega_3^2 + \omega_3)x + x^{2/3} .$$

This is not satisfactory because the fact that

$$(4.6.4\text{-}4) \quad \omega_3^2 = -\omega_3 - 1$$

must be used to effect a further simplification. Equation (4.6.4-4) can be deduced from the cyclotomic polynomial (see (47) p. 113):

$$(4.6.4\text{-}5) \quad \phi_3(x) = x^2 + x + 1$$

whose roots are cube roots of unity.

We can generate any cyclotomic polynomial by a procedure described in (47) as follows. Define the "Mobius function" $\mu(u)$ by

$$\mu(n) = \begin{cases} 1 \text{ if } n = 1 \\ 0 \text{ if for some integer } p,\ p^2 | n \\ (-1)^L \text{ if } n = p_1 \ldots p_L \text{ (i.e. } n \text{ is square-free)} \end{cases}$$

Then

$$\phi_h(x) = \prod_{d|h} (x^d - 1)^{\mu(h/d)}$$

$\phi_h(x)$ is easily calculated in MACSYMA. Now with the added step of simplifying roots of unity in R according to the cyclotomic equation, and resimplifying the result, the procedure is almost complete. For even roots, there is an additional step of eliminating symmetric values. (e.g. $\omega_8^5 = -\omega_8$) Although this does not guarantee a canonical form for the numerator because of such relations as (4.4.2-1), it will result in the simplification of the denominator. In our example, we find that

$$R = x^{1/3} x - x + x^{2/3} \text{ and } R\,Q = x^2 + x.$$

Thus

$$\frac{1}{x^{2/3} + x^{1/3}} = \frac{x^{1/3} x - x + x^{2/3}}{x^2 + x}$$

4.7 The Properties of the Simplified Form

4.7.1 Overview and History

Our object is to prove that any radical polynomial which may be written in form (2) is in canonical form. Let $\underline{Z}$ be the ring of integers. Basically we wish to show that sufficient irreducibility criteria may be obtained to show that each v_k is non-trivial algebraic over $\underline{Z}[X](v_1,\ldots,v_{k-1})$, and that we have a regular form for radical polynomials. It is clear that roots of -1 will have to be dealt with in another manner. With suitable restrictions to the class of representatives of the extensions, the regular form can be strengthened to a canonical form.

Caviness in (5) stated and proved several irreducibility criteria which, for his approach to the problem, seemed quite reasonable. That they are not strong enough is primarily due to his interpretation of radicals, which caused him to first extend the base field by a primitive nth root of unity. This is not necessary in many instances, and much stronger results can be obtained without this first extension.

4.7.2 The theorem

Theorem 4.7.2-1

Let $n > 1$ be any integer, $X = \{x_1, \ldots, x_N\}$ be a set of indeterminates, and $p_1, \ldots, p_k$ be a set of (distinct) positive prime numbers or primitive polynomials over X, irreducible over the integers. Let $\underline{Q}$ denote the rationals, and $p_i^{1/n}$ denote the positive (or any fixed) nth root of p_i. Then the field $\underline{Q}(X)(p_1^{1/n}, \ldots, p_k^{1/n})$ is of degree n^k over $\underline{Q}(X)$.

Equivalently, Let $E_k = \{e_i\}$ denote the set of n^k elements

$$p_1^{m_1/n} * \ldots * p_k^{m_k/n} \qquad \begin{array}{l} 0 \leq m_h < n \\ 1 \leq h \leq k \end{array}$$

Then the set E_k is linearly independent over $\underline{Q}(X)$.

The p_i which are polynomials may be square-free and relatively prime in pairs instead of irreducible without altering the result. Also, n need not be fixed over the p_i: then the n of the theorem can be the least common multiple of any number of distinct n's. (Thus if fifth and cube roots appear, we deal with fifteenth roots.) Note that roots of unity are <u>not</u> included in this theorem; 1 is not a prime number.

<u>Proof</u>

STEP 1: Reduction to the case n is a power of a prime number.

Let $F_n = \underline{Q}(X)(p_1^{1/n}, \ldots, p_k^{1/n})$ for $n = 1, \ldots$. Clearly $F_{mn} \supset F_n$ and F_m. Hence if F_n is of degree n^k and F_m is of degree m^k the degree of F_{mn} is divisible by $(mn)^k$, hence equal to $(mn)^k$ since it is obviously not greater. Thus it suffices to consider n a power of a prime number.

Let $F_{n,0} = \underline{Q}(X)$, $F_{n,1} = F_{n,0}(p^{1/n})$, etc., so that F_n of the previous paragraph is $F_{n,k}$. We must show that the degree of the extension $F_{n,i}/F_{n,i-1}$ is n for i = 1, ..., k; or in other words, we must show that the equation $y^n - p_i$ is irreducible in $F_{n,i-1}$.

STEP 2: Proof for n the power of a prime q.

There are two cases, q = 2 and q an odd prime. We can summarize some useful theorems ((46) pp. 291-293, Satz 7,8) as follows:

Let q be a prime number, ν a natural number, and α a member of a field F.

Except for q = 2 and $\nu > 1$,

$y^{q^\nu} - \alpha$ is reducible over F iff $\alpha = \beta^q$ where β is in F.

For q = 2 and $\nu > 1$,

$y^{2^\nu} - \alpha$ is reducible over F iff $\alpha = \beta^2$ or $\alpha = -4\beta^4$ where β is in F.

Note that within our context, β is either a positive prime number (<u>not</u> 1) or an irreducible (or square-free) primitive polynomial over X. What will be shown here is that the degree of $F_{n,i}/F_{n,i-1} = n$.

Consider the case i=1. For $y^n - p_1$ to be reducible, p_1 must be β^q, β^2 or $-4\beta^4$ for β in $F_{n,0} = \underline{Q}(X)$. Any of these possibilities would imply that the polynomial p_1 has multiple roots in

$F_{n,0}$. Since p_1 has real integer coefficients, complex roots of $y^n - p_1 = 0$ would occur only in conjugate pairs, and real (integer) roots cannot occur since p_1 is irreducible over the integers. (If p_1 is only square-free, any real roots could not be multiple.) Thus the polynomial $y^n - p_1$ is irreducible over $F_{n,0}$.

The rest of the proof is by induction on i, the index of p, for i = 2, ..., k.

Assume that previous polynomial extensions have been successfully adjoined. We must show the impossibility of

$$p_{i+1} = \beta^q, \beta^2, \text{ or } -4\beta^4$$

where β is in $F_{n,i}$, i = 2, ..., k-1.

In fact, we will even allow β to be in $G_{n,i} = F_{n,i}(\omega_n)$ without changing the results. Clearly, if a polynomial is irreducible in G, it is irreducible in the corresponding F. The proofs for β^2 and $-4\beta^4$ are essentially similar to the proof below for β^q.

Let us assume the falsity of the theorem. Then $p_{i+1} = \beta^q$ where β is in $G_{n,i}$. Then we can express β in this form:

$$(4.7.2\text{-}1)\qquad \beta = \sum_{j=1}^{n^i} c_j e_j \qquad c_j \in G_{n,0}$$

If all but one of the c_j is zero, $p_{i+1} = c_j^q \cdot e_j^q$ which contradicts the hypothesis that p_{i+1} is irreducible over the integers. It even contradicts the weaker hypothesis that p_{i+1} is square-free and relatively prime to each of the other polynomials. The contradiction is obtained as follows: e_j^q must be rational, since it is the ratio of p_{i+1} and c_j^q, both rational. But then p_{i+1} has factors c_j^q and e_j^q. Even if c_j is 1, p_{i+1} and e_j^q must have a non-trivial GCD, since their ratio is then 1. Finally, the square-free condition on p_{i+1} eliminates the possibility that e_j^q itself is (somehow) rational.

Assume there are at least two terms, $c_j, c_h \neq 0$. By the induction hypothesis, the Galois group of $G_{n,i}:G_{n,i+1}$ is transitive, so there is an automorphism Ψ of $G_{n,i}$ over $G_{n,i+1}$ where $\Psi(e_j)/\Psi(e_h) \neq e_j/e_h$. For some set of integers $\{r_m\}$, this automorphism maps e_m into $\omega^{r_m} e_m$. Since $\beta \in F_{n,0}$, the q qth roots of β^q all look like $\omega^{r_q}\beta$, and any automorphism will just interchange the roots. Applying Ψ to equation (4.7.2-1) yields

$$(4.7.2\text{-}2) \qquad \omega^{r_0}\beta = \sum c_j \,\omega^{r_j} e_j$$

Since ω is in $G_{n,i}$, equations (4.7.2-1,2) contradict the assumed linear independence of the e_k over $G_{n,i}$. The same result holds for $F_{n,i}$.

Since the argument for β^2 and $-4\beta^4$ is quite similar, this concludes the proof.

4.8 Canonical Forms

Recall that Caviness does not provide a canonical form for radical polynomials. Two equivalent expressions may be non-identical because each is expressed in a different algebraic extension of $\underline{Q}(x)$. In our simplified form (2) we are dealing with the same field representatives so that we are always dealing with the same extensions for those expressions which can be put into form (2). Thus, except for minor points dealing with such questions as the ordering of terms in a sum or product, form (2) is a canonical form for that subset of $\mathcal{P}$ which can be represented in form (2). With suitable recombinations of square-free factors, form (1) can also be made canonical.

One of the consequences of this canonical form is that we can define a content and primitive part for a simplified radical polynomial. Let $F = \underline{Z}(X)$, where $\underline{Z}$ is the ring of integers. Then E, a simplified member of ρ is in $F[v_1, \ldots, v_k]$. The content c of E is a polynomial in G such that $E = c \cdot P$ where P, the primitive part, is a member of G whose coefficients in G are relatively prime polynomials (or integers). This c is unique up to a unit multiple. (see, e.g. (27), p. 366)

Now we may draw a few conclusions about the canonical properties of the radical expressions with rationalized denominators. Basically, we wish to show that the denominator is unique, regardless of the order in which the v_i are removed. In fact, the value of the denominator is not unique, but it can be made unique.

With the use of this concept of content, we can transform a radical expression with a rational denominator into a reduced radical expression; one such that the content of the numerator has no factor in common with the denominator. Any such expression can then be written as $c \cdot N/D$ where c and D are relatively prime polynomials, and N is a radical polynomial in simplified form.

Theorem 4.8-1

If P and P' are reduced radical expressions with numerators in form (2) such that $P' = P$, then $P \equiv P'$.

Proof

Let $P \equiv c \cdot N/D$ and $P' \equiv c' \cdot N'/D'$ as above. Cross multiply to get $c \cdot N \cdot D' \equiv c' \cdot N' \cdot D$. The primitive part of two equivalent radical polynomials must be the same, so $N \equiv N'$. But then c/D and c'/D', both rational expressions, are clearly identical also.

4.9. Additional Radical Proposals

An extension to variable exponents has been made in the simplification algorithm described in section 4.6. In order to make more apparent the relationships between, for example, e^x and $e^{x/2}$, non-numeric exponents of the base e are collected and examined for common factors. Bases different from e are converted: $y^x \dashrightarrow e^{x \log(y)}$; the same convention of positivity defined in section 4.3 is used to choose a unique branch for the log. If common factors are discovered, they result in re-representation of the elements. Thus in the presence of $e^{x/2}$, e^x is represented by $(e^{x/2})^2$. In the terminology of section 4.2, we wish to establish a regular field description for the expression in terms of monomials. One consequence of this would be that

$e^x + 2\,e^{x/2} + 1$ simplifies to $e^{x/2} + 1$.

Since the conditions requiring this form of processing are disjoint from those of the radical expressions (i.e. variable exponents are not allowed in), this aspect of simplification does not interfere with the algorithm of section 4.6: This part of the extended algorithm is completely bypassed unless there is a variable exponent.

To outline some details of the algorithm, let p and q be polynomials. Then p^q is written as $e^{q \log(p)}$, then as

$$e^{(q_1 + ... + q_n) \log (p_1 ... p_m)}$$

and then as a product of terms of the form $e^{q_i \log(p_j)}$

If q is a rational (rather than polynomial) expression, it can be written in a partial fraction expansion form (15) as a sum. Some of these terms $e^{q \log(p)}$ will be recognizable as multiples of others, say $e^{q/k \log(p)}$. They are rewritten to reflect this fact, and the expression is rationally simplified with respect to this new set of variables. Further efforts in this direction are described in (15). Exponentials are of particular interest because when extended to complex arguments, they can be used to express sines, cosines, etc.

4.10. Conclusions

A simplification procedure for radical expressions which involves only gcd calculations has been presented. While Caviness' version of a similar algorithm "is so encumbered by combinatorial difficulties that it cannot be considered a practical routine," (p. 72 (5)) the algorithm presented here is easily applied.

This result is an improvement of Caviness' results in several ways, although it is not as general as his approach.

1. Factoring over arbitrary algebraic extensions of the rational field is unnecessary; in fact, only greatest common divisor calculations (of polynomials with integer coefficients) is required.

2. It demonstrates that the only cases requiring Caviness' approach are expressions which contain roots of -1.

3. The inessential restriction to one variable is deleted.

In terms of theoretical advances, the PRI formulation of the interpretation of radicals allows a new and more useful theorem (4.7.2-1) concerned with the canonical properties of this simplified form to be proved.

Extensions of the algorithm of section 4.6 are discussed in terms of their usefulness and practicality: several seem useful enough to suggest that they be made available as programs in MACSYMA or similar systems. A description of several uses of just such a simplification capability have been described in Chapter 3. The RADCAN routine in MACSYMA uses the easily computed form 1' of section 4.5, but does not ordinarily use the denominator rationalization technique. RADCAN also incorporates the additional ideas in section 4.9, except for partial fraction decomposition of exponents.

One of the important consequences of this simplification approach is that new integration methods developed by Risch (40), (41) which rely on the regularity (in the sense of 4.2) of integrands, can be implemented with relative ease.

Chapter 5 - Summary and Prospects for the Future

5.1 Summary

In Chapter 2, The User-level Semantic Matching Facility in MACSYMA, we have described a project both in man-machine communication, and in analyzing the concept of an occurrence of an instance of an algebraic pattern. While this pattern recognition capability has great appeal in the initial approach to solving a new problem, it often yields to more efficient methods when the problem is better formulated. Nevertheless, we feel it is an important tool to place in the hands of a user of an algebraic manipulation system: it gives a handle on problems difficult to formulate in other modes and allows a user to mold a system to conform to his set of expressions and operations. We tried to clarify what we feel to be important in terms of semantic pattern recognition; merely syntactic patterns leave much to be desired in terms of ease of use and flexibility. We have demonstrated several types of applications of pattern matching extensions to MACSYMA: writing programs, writing Markov-algorithm rule-sets, and modifying the simplifer.

In Chapter 3, the rational function representation of MACSYMA serves as a vehicle for several algorithms which, we believe, correspond to notions often used informally and imprecisely in referring to operations on mathematical expressions. For example, the notion of coefficient, as embodied in RATCOEF allows a mathematician to "collect terms" in a single command. This would be quite difficult to phrase in terms of traditional programming languages, or for that matter, formal mathematical definitions. This imprecision is even more apparent in the problem of substitution, where the most obvious aspect of the problem is its inherent ambiguity. By defining the RATSUBST algorithms for substitution, while drawing on the power of the rational function programs to perform global transformations on an expression, several varieties of unambiguous substitutions may be performed. Another use of the rational function representation, in the SOLVE programs, has had consequences in many other parts of MACSYMA. Being able to solve for roots of polynomials (by factoring, or through radicals) while at the same time solving for transcendental roots (by inverting functions), has saved other programs considerable efforts. SOLVE forms an effective tool for finding poles and singularities, an important task for limit calculations, definite integration (using contour integration and residue calculation) and series expansions.

In chapter 4 we have described new simplification algorithms for radical expressions, one of which uses only polynomial greatest-common-divisor operations for its effectiveness. The major consequence is that we can produce a canonical form over the class of radical expressions not requiring roots of -1. The importance of the simplification algorithm lies in two areas: in general, simplification to a canonical form is valuable, especially if it is inexpensive; secondly, the Risch integration algorithm depends strongly on such results.

5.2 Prospects for the future

The research described in this thesis, and similar work is directed toward the goal of making man more creative and computers more useful. No doubt many areas of computer science will benefit from a better understanding of how knowledge can be incorporated into a programming system for mathematics.

The goal of a computer system for the future which 1) is a drudge-work slave capable of doing massive calculations, correctly and rapidly, 2) is a mathematical co-worker, capable of conversing in natural notations about many problems from a wide range of disciplines; and 3) is an encyclopedic mentor, being an organized collection of pertinent facts and algorithms drawn from all of mathematical analysis, is a tempting challenge.

Reference 1, the proceedings of SYMSAM/2, is a detailed description of the present state of the art in algebraic manipulation. Of the papers in that volume, 11 are concerned with applications of computers to group theory, a branch of symbol manipulation which is of limited interest to us here. Some 7 papers deal with applications of computers to problems of interest to workers in celestial mechanics, relativity, high-energy physics, and other areas where massive problems in algebra occur. Some 7 papers discuss the present state of new or revised computer systems for algebraic manipulation. About a dozen are concerned with basically peripheral issues: specifying the syntax for a language for communicating with an on-line system for algebraic manipulation, hand-written input, transportability of such systems, etc. A significant number of papers (8) describe developments which may actually benefit all, or most, algebraic manipulation systems of the future, in that they introduce new algorithms for standard operations of mathematics. Often these algorithms represent improvements, in terms of computation time, of many orders of magnitude. Furthermore, the theoretical tools used for analyzing the times taken for execution of algorithms have been considerably refined. A major factor in the success of these algorithms has been the use of modular arithmetic. Four papers on simplification, and three on limits and integration completed the program.

Many of these techniques and manipulatory algorithms described at SYMSAM/2 are present, or are being implemented in MACSYMA, and for that matter, in several other current algebraic manipulation systems. By joining together as many of these advances as possible, in forms which are easier to approach than the original implementations, we hope to produce a system which is useful both in applications, and in exploring the problems of algebraic manipulation in general.

During the development of MACSYMA, a number of applications came to our attention through colleagues at M.I.T. and Harvard. As a rule, we have avoided the so-called "naive user," but have attempted to attack problems with a combination team of a programmer familiar with MACSYMA (the author) and an application specialist. This approach has sufficed to explore several problems. With Prof. Eytan Barouch of M.I.T's Department of Applied Mathematics, several problems arising in statistical mechanics were attempted, with only moderate success: each problem was reduced to a solvable one whose answer was, in retrospect, not sufficiently accurate to be interesting; the full problem was (at least without new insights) demonstrated to be beyond present machine capacity. With Mr. Francis Heile, an M.I.T physics graduate student, a masters' thesis (25) calculation was completed; the hand calculation of the same result had been abandoned because of its complexity. This involved the computation of traces, utilizing symmetry properties for simplification.

With Mr. Henry Mok, another M.I.T. graduate student, the arithmetic statements in a large computational program for use in plasma physics research, were checked (and an error found in the previous hand calculations).

Other members of the programming group have tried MACSYMA on problems that have come to their attention; that few produce directly publishable results is not surprising -- the ratio of good ideas to bad in mathematics is probably quite small, although the bad ideas often require just as much investigation as the good. It appears that systems such as MACSYMA can perform a useful service if they only weed out untenable computational approaches more rapidly than hand computations. They are only occasionally called upon to solve problems to completion by long and complex computation.

Where can we go from here? In the first place, the study of particular applications is a necessity. In this way, further developments of systems like MACSYMA can proceed in the most useful direction, rather than towards the unnecessarily overgrown syntactical and semantic prospects that seem to threaten developing algebraic manipulation systems.

Of course, since applications are presumably the justification for all this work, efforts in making such systems more readily available for applications may be of major importance. These efforts should include hardware development (especially well-designed terminals, cheaper, faster, list-processing computers, larger memories), and system development (especially systems for management of large programs and data-bases, the sharing of resources, and the facilitation of man-machine interaction).

There is clearly more work to be done in understanding the methods of mathematical analysis, and studying alternative algorithms for common tasks. Large strides in this direction have already been taken, but more is needed in understanding such "simple" questions as "What is the fastest method for raising a polynomial to an integer power?" (recent work by L. Heindel, (22) and separately, W. M. Gentleman (17) draw some conclusions; my own work in this area may be found in (13))

In terms of making a more versatile mathematical system, we might explore the construction of a deductive mathematical system with a more varied data base. We should be able to use such facts as "the second derivative of F with respect to X is 0" or "the absolute value of Y is less than or equal to 5." No current system goes very far toward the difficult task of welding mathematical facts into a coherent data-base.

We should seek to extend the present boundaries of the functions which we can handle through rigorous methods. Zero-equivalence tests for handling large classes of functions appear to be possible. Stronger results concerned with nested radicals may also be possible.

Better algorithms for factoring polynomials, computing solutions to sets of linear equations, may also be valuable goals, but it may be that heuristic approaches will begin to dominate parts of the currently algorithmic problem domain. By rearranging the ordering of variables, enormous speed factors can be gained in, for example, computing polynomial greatest common divisors. Certainly a human mathematician would explore these other methods before embarking on a lengthy calculation -- why not a computer?

Just how realistic these goals are, remains to be seen. The research of the last decade has revealed both unexpected successes, and surprisingly basic difficulties. We expect that, as in the past, research in the field of algebraic manipulation will continue to have important implications in our understanding of mathematical algorithms, the nature of computation and artificial intelligence, programming languages and systems, and problems of man-machine communication.

Appendix I

MACSYMA Users' Manual

Commands to MACSYMA are strings of characters representing mathematical expressions, equations, arrays, functions, and programs. Extra spaces and carriage returns are ignored. Commands are terminated by @ or $. @ causes the command to be evaluated and the result displayed. $ causes the command to be evaluated but the display of the result is suppressed. When typing commands, "rubout" or "delete" deletes (and echoes back at the console) the previous character; ?? deletes the whole command, and causes the line number to be redisplayed.

A potential user who is interested primarily in the "commands" available should peruse the following page, summarizing the input syntax of MACSYMA, and then may skip ahead to section I.7.

Figure A.1 Syntax of MACSYMA Expressions. Examples of the legal input expressions and their meanings are shown below. W, X, Y, and Z stand for any expressions; U and V for variables. Some of these forms can be extended to take an arbitrary number of arguments in the obvious manner.

INPUT	MEANING
AB	variable
'AB	quoted variable
1	integer
1.2	floating point number
F[X,Y]	subscripted variable
F(X,Y)	function invocation
F[X,Y](W,Z)	subscripted function invocation
X!	factorial
X**Y or X↑Y	exponentiation
X/Y	quotient
-X	negation
X+Y	sum
X-Y	difference
X*Y	product
X=Y	equality predicate or equation
X<Y, X>Y, X>=Y, X<=Y	less than, greater than, less than or equal to, greater than or equal to predicate or inequality
X AND Y	logical AND or Boolean operator
X OR Y	logical OR or Boolean operator
'X	quoted expression
[X,Y]	list of expressions
IF X THEN Y	conditional
IF X THEN Y ELSE W	conditional
FOR I:1 STEP 1 UNTIL I>3 DO X	DO loop
A:X	assign the value of X to A
V::X	assign the value of X to the value of V
A(V):=X	define function A(V)
A(V):Y FOR ALL W	define function A(V)
X.Y	non-commutative product
A FOR INTEGER 3<X<10	indexed set
A FOR 3<X<10	real line segment
BLOCK(X,Y,Z)	program block (X,Y,Z are statements)
X,Y,Z	EV(X,Y,Z): X is an expression, Y,Z specify environment

I.1 The Input Stream Editor

At any point while he is inputting a comand, the MACSYMA user can enter the input-stream editor by typing #. The editor is given the string of characters typed so far in the current command. In the case of a detected syntax error, the entire previous command string will be given to the editor.

All the commands to the editor reference a cursor (an underscore or back-arrow, depending on the console) which is displayed within (or at either end of) the string of characters under edit. In the description to follow, n stands for a positive or negative integer. The default value of n is +1. If n is positive, the commands operate toward the right of the cursor; if n is negative, they operate toward the left.

nC	moves the cursor n characters.
nR	moves the cursor n characters in the reverse direction(nR=-nC).
J	(also T) moves the cursor to head (top) of string.
nL	moves the cursor to the right of the nth carriage return (e.g. L moves to the next line)
Sstring#	moves the cursor to the right of the first occurrence of the string of characters "string" searching toward the right. (-S implies left)
nD	deletes n characters.
nK	deletes all the characters through the nth carriage return. (e. g., K deletes the remainder of this line)
Istring#	inserts the characters "string"
#	leaves the editor and returns to inputting from the user's console.

I.2 System Control

Lines are consecutively numbered, except that the input line Ci will be following by an output line (if one is generated) named Di. The next input-output pair will be labelled C(i+1) and D(i+1), respectively. If one command produces several lines of output, the labels will begin with an E, and the line number will be incremented for each additional line. A user can refer to any command or expression by its line label. The most recently computed expression may be referred to as "%".

The system can be set to automatically write old expressions onto secondary storage. The process is controlled by the following variables which can be set by the user. (e.g. FILESIZE:10$ would set FILESIZE, or by the OPTIONS command.)

variable	default value	purpose
DSKUSE	FALSE	If set to T, then expressions are automatically filed away.
FILESIZE	10	Expressions are written out with FILESIZE expressions in each file.
RETAINNUM	8	When the number of expressions in core reaches FILESIZE + RETAINNUM, a file is written.
FILENAME	username	The first name of the file written out. The second names (our filing system requires two names for a file) are 1,2,....
INCHAR	C	The prefix character for inputted line numbers.
OUTCHAR	D	The prefix character for outputted line numbers.
LINECHAR	E	The prefix character for intermediate-output line numbers.

When an expression is written out, the name of the file containing it is attached to the expression name in core. Thus when the expression is referenced in a later step, it can be automatically retrieved from the file.

At the end of the session, the secondary storage files can be deleted by the command FINISH(). The command FINISH(TRUE) allows the user to retain some or all of the expressions on his file. In order to specify the form and contents of the retained file, he must answer a series of questions:

question	response	meaning
OUTPUT DEVICE?	(file spec)	The name of the file on which the output will be saved.
EDIT?	N	Save the files as they are. This response will cut off further questions.
	Y	Read the files back into memory, one expression at a time, so that selected expressions can be saved on the previously specified file.
INTERNAL?	Y	Save the expressions in machine readable form. In this form they may be read back into a fresh system using RESUME.
	N	Save only the two dimensional display forms.
SAVE?	Y	(This is asked for each expression.) Include the expression currently displayed.
	N	Do not include it.

RESUME(file specification) reads a file previously outputted through FINISH, displaying the commands and recomputing the results.

BATCH(file specification) reads an input text from the designated file, command by command. When the end of the file is reached, further commands may be supplied by the user at his console. DEMONSTRATE is like BATCH except it pauses waiting for the user to type a space between commands. Any other character causes a return to the user console for further input. Additional arguments may be supplied to cause some of the input lines to be skipped.

1.3 Rules for Expression Evaluation

We have attempted to define a philosophy of expression evaluation in MACSYMA so as to lead to the most natural mathematical usage. In so doing, we necessarily become involved in a complicated set of rules to (in part) replace explicit quoting mechanisms, such as that in LISP. If we have been successful, a mathematician should rarely, if ever, have to refer to the rules concerned with noun and verb forms, below.

A:X assigns A the value of X. This is the way a user would typically assign a value to a variable. Values are also assigned when the variables are used as labels for expressions on input and output. This assignment is analogous to the FORTRAN "=", or the LISP SETQ.

A variable which does not have a value stands for itself. Numbers always stand for themselves.

A built-in or user-defined function is either a noun-function or a verb-function. A verb-function is one which attempts to effect an applicaton of the function to its arguments and thereby remove itself from the expression. For example, INTEGRATE is a verb-function, and ordinarily will attempt to perform an integration. On the other hand, SIN is a noun-function, and will not attempt to evaluate itself, although it will evaluate its arguments. The EV command can be used to evaluate an expression in a context which says that (for example) all SINs should be numerically evaluated. That is, selected noun forms can be converted to verb forms. Similarly, if a normally verb-type function is desired to operate as a noun-type function, it may be so declared via the function NOUN. Thus INTEGRATE, when declared a noun, would normally return an integral, even if the integration could be performed. If the function F is a verb, 'F can be used as the noun form for F. If F is a noun, ''F (<u>not</u> "F) can be used as the verb form. If a verb-function cannot be evaluated, as, for example, an integral which cannot be computed, it is simply returned as though it were a noun-function. If F is already a noun, 'F is the same as F.

Transcendental functions are nouns. Other defined functions are verbs unless their names are quoted. The arguments of undefined functions are evaluated, but, obviously, the function itself cannot be evaluated, and so is treated as a noun. As an expression is evaluated, it is also simplified.

If a name is subscripted (a subscript is enclosed in square brackets on input), then its value is stored in an array. The size of an array may be declared by the command ARRAYSIZE (<u>name</u>,<u>size</u>)$. An array need not have its dimensions declared, but if it has been declared, it will be permitted to have only numerical subscripts. At the first attempt to store a value in an undeclared array, a mechanism will be set up to describe the entries and their values in terms of a hash-coded list. The hash code can be computed from the subscripts whether or not they are numerical. If an array is subsequently declared, the values in the hash table are transferred to the new (true array) organization. The value of an array entry can be a number, expression, equation (etc.) regardless of whether it is a hash array or a true array. A hashed array is organized as follows: It is initially allocated a hash table with four entries. Each table entry contains a list of subscripts and values which hashes into that entry. Whenever the number of entries with values is equal to the size of the hash table, the size of the hash table is doubled. Whenever the operation ":" is executed, a check is made to see if the name is subscripted. If so, the appropriate array entry is set.

A::X assigns the value of A the value of X. The value of A must be a variable in this situation. This is analogous to a LISP SET.

I.4 Function Definitions and Arrays

MACSYMA incorporates a programming syntax resembling Algol-60 for use on the top (command) level and in function definitions. The parser is entirely syntax directed, so that modifications to the grammar can be easily included; also, an exact definition of the acceptable forms (and their interpretations in terms of LISP and MACSYMA functions) can (but will not) be given. The syntax is illustrated in figure I.3. Each of these constructions has fairly conventional interpretation, except when symbolic and traditional numeric notions conflict. One such instance is in inequalities, and is discussed in the next section in more detail.

The first argument to ":=" (the function definition operator) may take one of three forms: f(x), f[i] or f[i](x). Let the second argument to ":=" (that is, the right hand side) be y. In the first case, the variable f denotes a function, with value lambda(x)y. In the second case, a function definition is being associated with an array. The name f is denoted an AEXPR with value lambda(i)y. An AEXPR is used as follows. If a particular value of an undeclared array (it is an array if the variable is subscripted or if the name has previously been subscripted and assigned a value) is not present in the associated hash table, a check is made to see if the name also denotes an AEXPR. If so, this function is evaluated and the resulting value is stored in the hash table and also returned. If no value is present and no AEXPR is present, the expression is handled as though it were an undefined function.

If the first argument of ":=" is f[i](x), the third case, then f is denoted an AEXPR as above, but this AEXPR evaluates to a function of x. For example, given f[i](x):=x**i, evaluating f[3](5) would cause the AEXPR to be evaluated to lambda(x)x**3 and this value would be stored as the value of f[3] and also applied to 5 to yield 5**3. A subsequent evaluation of f[3](7) would cause the value lambda(x)x**3 of f[3] to be retrieved and applied to 7.

The second argument to := (the right hand side, or procedure body) is ordinarily left unevaluated. This may be altered by the use of the double quote ("), which causes immediate evaluation. Thus F(X):="%@ uses the most recently computed expression as the procedure body.

If several expressions or commands are included in the procedure body, and dummy variables are needed, the correct form is, for example,

```
F(X):=BLOCK([Y] , Y:1, A, IF(Y>X) THEN RETURN(Y) ELSE
        DISPLAY(Y), Y:Y+1, GO(A))
```

This is equivalent to

```
F(X):=FOR Y:1 STEP 1 UNTIL Y>X DO DISPLAY(Y)
```

I.5 Predicates and Conditionals

The comparison operators ">", "<", and "=" are not evaluated in ordinary contexts; that is, they are nouns. However, these operators, along with AND and OR are evaluated when they are in the predicate position of the IF-THEN-ELSE construction; that is, they are transformed into verbs. If the predicate evaluates to FALSE, the ELSE clause is evaluated and returned. Otherwise the

THEN clause is evaluated and returned.

I.6 Special Constants

There are a great many special constants and functions in mathematics which, if given their common names, would pre-empt many of the letters of the alphabet. To avoid various types of misunderstandings, we have chosen the symbol "%" as an "escape" character for special symbols. Thus the base of the natural logarithms, e, is typed into MACSYMA (and displayed by MACSYMA) as %E. Other symbols in this category include %PI (the ratio of the circumference to the diameter of a circle), and %I, the square root of -1.

I.7 General Purpose Commands

INTEGRATE(exp,var) integrates exp with respect to var or returns an integral expression if it cannot perform the integration. INTEGRATE(exp,var,low,high) finds the definite integral of exp with respect to var from low to high. Several methods are used, including direct substitution in the indefinite integral and contour integration. Improper integrals may use the names INF for positive infinity and MINF for negative infinity. If an integral "form" is desired for manipulation (for example, an integral which cannot be computed until some numbers are substituted for some parameters), the noun form 'INTEGRATE may be used.

DIFF(exp,var1,n1,...,vark,nk) differentiates exp with respect to vari ni times. If k=1 and n1=1, n1 may be omitted: DIFF(exp,var). If the derivative "forms" are required (as, for example, when writing a differential equation), 'DIFF should be used.

DEPENDENCIES(f1,...,fn) declares functional dependencies used by DIFF. Each fi (i=1,n) has the format f(v1,...,vm) where each vi (j=1,m) is a variable on which f depends. Thus DIFF(Y,X) is 0, initially. Executing DEPENDENCIES(Y(X))$ causes future differentiations of Y with respect to X to be displayed as

```
DY
--
DX
```

GRADEF(f(x1,...,xn),g1(x1),...,gn(xn)) defines the derivatives of the function f with respect to its n arguments. That is, df/dx1 = g1(x1), etc.

LIMIT(exp,var,val,dir) finds the limit of exp as the real variable var approaches the value val from the direction dir. Dir may have the value PLUS for a limit from above, MINUS for a limit from below, or may be omitted (implying a two-sided limit is to be computed). LIMIT uses the following special symbols: INF (positive infinity) and MINF (negative infinity). On output it may also use UND (undefined) and IND (indefinite but bounded). 'LIMIT may be used to simply create a limit noun form.

RESIDUE(exp,var,val,order) computes the orderth residue in the complex plane of the expression exp when the variable var assumes the value val.

SUBSTITUTE(a,b,c) substitutes a for b in c. b must be an atom or a function with arguments, rather than a function with only some of its arguments. When b does not have these characteristics, one may sometimes use SUBSTPART or RATSUBST. SUBSTITUTE(eq1,exp) or SUBSTITUTE([eq1,...,eqk],exp) are other permissible forms. The eqi are equations indicating substitutions to be made. For each equation, the right side will be substituted for the left in the expression exp (if the left side is non-atomic, and the right side is, the equation will be "flipped")

EXPAND(exp) will cause an expansion of the argument. The MACSYMA variables MAXNEGEX and MAXPOSEX (originally set to 6) control the maximum negative and positive exponents, respectively, which will expand. EXPAND(exp,p,n) expands exp, but uses p for MAXPOSEX and n for MAXNEGEX.

SIMPLIFY(exp) simplifies its argument, thus overriding the value of the MACSYMA variable SIMP which if set to FALSE stops simplification.

PART(exp,n1,...,nk) obtains a subexpression of exp which is specified by the indices ni. The index n1 which (like all the indices is a non-negative integer) selects the argument of the top level operator of exp corresponding to its value. Thus PART(Z+Y,2) yields Y. The index n2 (if specified) picks up an argument of the result of PART(exp,n1). Thus PART(Z+2*Y,2,1) yields 2. The operator is considered to be argument 0.

In exponentiation, the base is considered argument 1, and the exponent argument 2. In a quotient, the numerator is argument 1, and the denominator is argument 2. A minus sign appearing in the display is considered as an operator. For example

```
(C1) X+Y/Z**2@
                           Y
(D1)                       -- + X
                            2
                           Z
(C2) PART(D1,1,2,2)@

(D2)                        2
```

DPART(exp,n1,...,nk) selects the same subexpression as PART, but instead of just returning that subexpression as its value, it returns the whole expression with the selected subexpression displayed inside a box. The variable %PART is given the value of the selected portion. Thus in the example above,

```
(C2) DPART(D1,1,2,1)@
                           Y
(D2)                      ---- + X
                             2
                         *****
                         * Z *
                         *****
```

SUBSTPART(x,exp,n1,...,nk) substitutes x for the subexpression picked out by the rest of the arguments. It returns the new value of exp.

KILL (arg1,...,argn) eliminates its arguments from the MACSYMA system. If argi is a variable, a function name, or an array name, the designated item is removed from core and the storage it occupies is reclaimed. argi = "HISTORY" eliminates all input and output lines to date (but not other named items). argi = a number, n, deletes the last n lines.

STORE(arg1,...,argn) is similar to KILL in that it reclaims core storage (but not quite as much). The values of the arguments to STORE are removed from core and saved on a secondary storage device. Special indicators left in core allow MACSYMA to read back these items whenever referenced. The arguments can be variables, function names, or array references. Numbers or "HISTORY" are not acceptable, since storage of the input and output lines is automatic and controlled by RETAINNUM.

SAVE (arg1,...,argn) simply backs up expressions on disk, but leaves them in core as well.

COEFF(exp,var,n) obtains the coefficient of var**n in exp. For best results, exp should be expanded. N must be an integer or a rational number. Coefficients of var**n which are functions of var are ignored. This command is less powerful than RATCOEF, but is sometimes convenient in interactive situations.

```
(C2) COEFF(Y+X*%E**X+1,X,0)@
(D2)                          Y + 1
```

SUM(exp,ind,lo,hi) performs a summation of the values of exp as the index ind varies from lo to hi. If the summation cannot be performed, or if 'SUM is used, the value is a sum noun form which is a representation of the sigma notation used in mathematics.

```
(C3) SUM(I**2,I,1,4)@
(D3)                          30
```

PRODUCT (analogous to SUM above).

EV(exp,arg1,...,argn) causes the expression exp to be evaluated and simplified with switches set according to the values of the argi.

EVAL reevaluates the expression so that variables in it which have values will be evaluated.

SIMP overrides the setting of the SIMP switch.

EXPAND causes expansion. EXPAND(n,m) set the values of MAXPOSEX and MAXNEGEX.

DIFF causes all differentiations indicated to be performed. DIFF(var1,...,vark) causes only differentiations with respect to the indicated variables.

NUMER causes SIN, COS, LOG, and "**" with numerical arguments to be evaluated.

v=exp causes the substitution of exp for v.

Any other function names (e.g. SUM) cause evaluation of occurrences of those names as though they were verbs.

The arguments following the first (exp) may be given in any order. It should be understood that EV performs a single evaluation and simplification. Thus all of the functions are performed in one scan. This is possible because the simplifier is used to perform expansions, differentiation, and numerical evaluations by the setting of switches. For example:

```
(C4) SIN(X)+COS(Y)+(W+1)**2+'DIFF(SIN(W),W)@
```

```
                                      D                      2
(D4)  COS(Y) + SIN(X) + --SIN(W) + (W + 1)
                                      DW

(C5) EV(%,NUMER,EXPAND,DIFF,X=2,Y=1)@

                   2
(D5)      COS(W) + W  + 2 W + 1.425324
```

An alternate syntax has been provided for EV, whereby one may just type in its arguments, without the EV(). That is, one may write simply exp,arg1,...,argn.

WHEN conditional DO identifier = expression e.g., WHEN I=2 DO K=%@. The value of the identifier is determined by evaluating the conditional. If it evaluates to TRUE, then the expression is evaluated and used for the value of that use of the identifier. If the conditional evaluates to FALSE, then the identifier's value is itself. In effect, the identifier becomes a function of no arguments which evaluates the conditional, and if TRUE, returns the expression as its value. Thus WHEN TRUE DO EXPAND=EXPAND(%)@ makes the atom EXPAND always evaluate to the last computed expression, expanded.

SOLVE(exp,var) solves the algebraic equation exp for the variable var. If exp is not an equation, it is assumed to be an expression to be set equal to zero. Var may be a function (e.g. F(X)), or other non-atomic expression except a sum or product. It may be omitted if exp contains only one variable. Exp may be a rational function, and may contain trigonometric functions, exponentials, etc. Its success may depend partly on switches set by the user. (see OPTIONS)
SOLVE([lhs1,...,lhsn],[v1,...,vn]) solves a system of linear algebraic equations. It takes two lists as arguments. The first list (lhsi, i=1,n) represents the equations to be solved; the second list is a list of the unknowns to be determined. If the total number of variables in the equations is equal to the number of equations, the second argument-list may be omitted. If the given equations are not compatible, the message INCONSISTENT will be displayed. If no unique solution exists, SINGULAR will be displayed. The solutions are exact, asuming the user has not used floating-point numbers in his input, and may involve symbolic variables. The solution set consists of a list of numbered equations and an index to the list.

DISPLAY(exp1,...,expn) prints equations whose left-hand-side is the exp, and whose right-hand-side is the value of the expression. The value of DISPLAY is a list of the labels of the equations displayed.

```
(C7) DISPLAY(D3,I)@
(E7)        D3 = X + Y
(E8)          I = 5
(D9)         [E7,E8]
```

PLOT(*exp*) produces an asterisk-plot of the expression *exp*. *Exp* may be of the form F(X) FOR 1<X<10, or F(X) FOR INTEGER 1<X<10, or [y1,...,yn] or F(X,Y,Z). In this last case, the user will be asked to define the dependent and independent variables, set the extra variables to constants, and provide the domain for the independent variable. If the list of Y-values is provided, the user will be asked for a list of the corresponding X-values.

GRAPH(*xvals*,*yvals*,*xlabel*,*ylabel*) graphs the two sets of data points, and labels the axes as indicated. The data points can be lists or indexed sets. The height and width of the display is affected by the parameters of the user-terminal, or can be altered by the use of OPTIONS.

APPEND(*x*,*y*) appends the two lists *x* and *y* and returns a single list of the elements of *x* followed by the elements of *y*.

CONS(*x*,*y*) returns a new list constructed of the element *x* as its first element, followed by the elements of *y*.

OPTIONS(*arg*) is a tree-structured collection of option-describing and option setting programs. The options concern the setting of switches for disk usage, display, simplification, the SOLVE program, etc. OPTIONS(CATEGORIES) lets the user in at the top level. See figure A.2 below.

Figure A.2 OPTIONS

```
   CATEGORIES
 A LIST OF CATEGORY NAMES EACH FOLLOWED BY THE OPTIONS IN THAT
CATEGORY

    CURRENT VALUE IS

[BOOKKEEPING,
        [DSKUSE,FILESIZE,RETAINNUM,FILENAME,DEV,UNAME,INCHAR,
        OUTCHAR,LINECHAR,TIME,CATEGORIES,RSET,NOUUO],
 SIMPLIFY,
        [SIMP,MAXPOSEX,MAXNEGEX,%EMODE,TRIGSIGN,SUBSTFLAG,%ETOLOGFLAG],
 DISPLAY,
        [NOSTAR,DERIVATIVEABREV,LINEL,SCOPEHEIGHT,SQRTFLAG,
        EXPTDISPFLAG],
 RATIONAL,
        [FULLFLAG,NOREPEAT,INVERTFLAG,FACTORFLAG,RADSUBST,MODULUS,
        BERLEFACT,GCDSWITCH,GCDOFF,RATEPSILON]
 SOLVE,
        [SOLVEFACTORS,SOLVERADCAN,SOLVEHEURS]]

   ***BOOKKEEPING***

   DSKUSE
 IF TRUE CAUSES OUTPUT FILE TO BE OPENED
 CURRENT VALUE IS FALSE

   FILESIZE
 THE NUMBER OF EXPRESSIONS WRITTEN TO SECONDARY STORAGE IN EACH
FILE
 CURRENT VALUE IS 10

   RETAINNUM
 THE NUMBER OF EXPRESSIONS IN MEMORY JUST AFTER A SECONDARY
STORAGE FILE IS WRITTEN
 CURRENT VALUE IS 8

   FILENAME
 THE FIRST NAME OF FILES OF EXPRESSIONS WRITTEN TO SECONDARY
STORAGE
  CURRENT VALUE IS usrxyz
  (where USERNAME begins with usr, and where xyz is a random
number)

   DEV
 THE DEVICE USED FOR FILING
 CURRENT VALUE IS DSK
```

```
   UNAME
 THE USERNAME USED FOR FILING
 CURRENT VALUE IS usr

   INCHAR
 THE FIRST LETTER OF THE NAMES OF EXPRESSIONS TYPED BY THE USER
 CURRENT VALUE IS C

   OUTCHAR
 THE FIRST LETTER OF THE NAMES OF THE VALUES OF OUTPUT
EXPRESSIONS
 CURRENT VALUE IS D

    LINECHAR THE FIRST LETTER OF THE NAMES OF THE VALUES OF
INTERMEDIATE DISPLAY EXPRESSIONS
 CURRENT VALUE IS E

   TIME
 IF TRUE CAUSES THE TIME REQUIRED TO EVALUATE EACH INPUT COMMAND
(EXCLUDING DISPLAY TIME) TO BE PRINTED
 CURRENT VALUE IS FALSE

   RSET
 IF TRUE INTRODUCES A SPECIAL DEBUGGING MODE
 CURRENT VALUE IS FALSE

   NOUUO
 IF TRUE INHIBITS MODIFICATION OF CALL INSTRUCTIONS, A DEBUGGING
AID
 CURRENT VALUE IS FALSE

   ***SIMPLIFY***

   SIMP
 IF TRUE CAUSES AUTOMATIC SIMPLIFICATION OF EVALUATED EXPRESSIONS
 CURRENT VALUE IS TRUE

   ***DISPLAY***

   NOSTAR
 IF TRUE CAUSES MULTIPLICATION TO BE DISPLAYED AS A SPACE
 CURRENT VALUE IS TRUE

   DERIVATIVEABREV
 IF TRUE CAUSES DERIVATIVES TO BE DISPLAYED AS SUBSCRIPTS
 CURRENT VALUE IS FALSE

   LINEL
 THE LINELENGTH USED FOR OUTPUT AND DISPLAY
 CURRENT VALUE IS 68
```

SCOPEHEIGHT
THE NUMBER OF LINES USED FOR PLOTTING
CURRENT VALUE IS 25

SQRTFLAG
IF TRUE, DISPLAYS SQRT AS SQRT. IF FALSE, DISPLAYS SQRT AS EXPONENT 1/2.
CURRENT VALUE IS TRUE

EXPTDISPFLAG
IF TRUE, DISPLAYS EXPRESSIONS WITH NEG. EXPONENTS USING QUOTIENTS.
CURRENT VALUE IS TRUE

RATIONAL

FULLFLAG
IF TRUE CAUSES RATSIMP TO MULTIPLY THROUGH AND REDUCE TO LOWEST TERMS FORMS LIKE (A↑B)↑C
CURRENT VALUE IS FALSE

NOREPEAT
IF TRUE NO GCDS ARE PERFORMED WHEN RE-RATIONALLY REPRESENTING AN EXPRESSION
CURRENT VALUE IS TRUE

INVERTFLAG
IF TRUE CAUSES RATSIMP TO REPRESENT A|(-B) AS (A|B)|(-1) THEREBY FACILITATING SUBSTITUTIONS
CURRENT VALUE IS FALSE

FACTORFLAG
IF TRUE CAUSES INTEGERS TO BE FACTORED BY FACTOR COMMAND
CURRENT VALUE IS TRUE

RADSUBST
IF TRUE ALLOWS RADCAN TO BE CALLED BY RATSUBST
CURRENT VALUE IS FALSE

MODULUS
IF MODULUS IS A POSITIVE INTEGER P ALL ARITHMETIC IN THE RATIONAL FUNCTION SYSTEM WILL BE DONE MOD P
CURRENT VALUE IS FALSE

BERLEFACT
IF TRUE THE BERLEKAMP FACTORING ALGORITHM WILL BE USED OTHERWISE THE KRONECKER ALGORITHM
CURRENT VALUE IS TRUE

GCDSWITCH
IF TRUE THE MODULAR GCD ALGORITHM IS USED OTHERWISE THE COLLINS REDUCED PRS
CURRENT VALUE IS FALSE

GCDOFF
IF TRUE ALL GCDS ARE 1
CURRENT VALUE IS FALSE

RATEPSILON
VALUE OF ACCEPTABLE ERROR IN CONVERTING FLOATING POINT NUMBERS TO RATIONAL NUMBERS IN RAT
CURRENT VALUE IS 1.0E-9

SOLVE

SOLVEFACTORS
IF TRUE SOLVE TRIES TO FACTOR GIVEN EXPRESSIONRUE
CURRENT VALUE IS TRUE

SOLVERADCAN
IF TRUE SOLVE SIMPLIFIES GIVEN EXPRESSION WITH RADCAN
CURRENT VALUE IS FALSE

SOLVEHEURS
IF TRUE SOLVE TRIES VARIOUS HEURISTICS
CURRENT VALUE IS FALSE

Rational Function Commands

RATVARS(<u>var1</u>,...,<u>varn</u>) provides a method for specifying the ordering of variables in CRE form. The most main variable will be <u>varn</u>, the least ("most constant") will be <u>var1</u>.

RAT(<u>x</u>,<u>var1</u>,...,<u>varn</u>) converts the expression <u>x</u> to CRE form. The optional <u>vari</u> serve as an ordering (as in RATVARS) but only within the scope of the single RAT command.

RATDISREP(<u>x</u>) converts a CRE <u>x</u> to a normal prefix expression.

RATSIMP(<u>x</u>,<u>var1</u>,...,<u>varn</u>), FULLRATSIMP(<u>x</u>,<u>var1</u>,...,<u>varn</u>), and RADCAN(<u>x</u>) are simplifiers. Currently, FULLRATSIMP and RATSIMP are identical.

FACTOR(<u>x</u>) factors a polynomial or rational function <u>x</u> (numerator and denominator).

SQFR(<u>x</u>) computes a square-free factorization of the expression <u>x</u>. A number of special checks for content with respect to several main variables occasionally factors polynomials even though the factors occur singly.

PARTFRAC(<u>x</u>,<u>var</u>) expands a rational function <u>x</u> in partial fractions with main variable <u>var</u>.

RATCOEF(<u>exp</u>,<u>x</u>) picks out the coefficient of <u>x</u> (which may be a power, product, sum, quotient, etc.) in <u>exp</u>.

RATSUBST(<u>a</u>,<u>b</u>,<u>c</u>) substitutes <u>a</u> for <u>b</u> in <u>c</u>. <u>B</u> may be a sum, product, power, etc.

GCD(<u>x</u>,<u>y</u>) computes the greatest common divisor of <u>x</u> and <u>y</u>.

DIVIDE(<u>x</u>,<u>y</u>,<u>var</u>) computes the quotient and remainder of <u>x</u> divided by <u>y</u>, as rational functions in a main polynomial variable, <u>var</u>.

RESULTANT(<u>x</u>,<u>y</u>,<u>var</u>) computes the resultant of the two polynomials <u>x</u> and <u>y</u>, and eliminates the variable <u>var</u>.

MOD(<u>x</u>) converts the polynomial <u>x</u> to a modular representation (mod MODULUS). <u>X</u> must be in only one variable.

GFACTOR(<u>x</u>) factors the polynomial <u>x</u> over the Gaussian integers (i. e. with SQRT(-1) = %I adjoined)

The Matching Subsystem

DECLARE(var,pred) declares var to match only expressions satisfying the predicate pred, when var is used in a pattern.

DEFMATCH(name,exp,var1,...,vark) defines a pattern matching program with name name.

DEFRULE(name,exp,repl) defines a transformation rule with name name which matches the pattern exp and transforms it to the replacement repl.

APPLY1(exp,r1,...,rk) (and similarly for APPLY2) applies the rules r1,...,rk to the expression exp, and returns the transformed expression. The difference between APPLY1 and APPLY2 is in the sequencing through the expression and rules.

TELLSIMP (pat,repl) (and similarly for TELLSIMPAFTER) changes the simplifier, so that in all subsequently simplified expressions, an occurrence of the pattern pat will be replaced by the expression repl.

Several additional predicates and testing programs are provided for use in constructing patterns and their predicates. SIGNUM(x) returns -1,0, or +1, depending on whether the sign of x is negative, zero, or positive. If x is a number, this question is simple. If x is not a number, its signum is computed from the coefficient of the leading term in a rationally simplified expression equivalent to x. FREEOF(x,y) returns TRUE if y does not depend explicitly on x. This is accomplished by searching through y for an occurrence of x, and assumes that x is not, for example, used as a dummy variable of integration. INT(x) returns TRUE if x is an integer. CONSTANT(x) returns TRUE if x is a constant. REALNUM(x) returns TRUE if x is a floating point number. RATNUM(x) returns TRUE if x is a rational number or an integer. NUMBER(x) returns TRUE if x is an integer or a floating point (real) number.

The Matrix Subsystem

The matrix subsystem currently is not completely integrated into MACSYMA, since matrix data types are not automatically handled by the simplifier. Furthermore, the ARRAY facility, which allows more general data types (e.g. hash-coded indices need not be integers), is a separate facility. This will, we hope, be remedied shortly.

For the moment, however, the following commands provide a fairly thorough set of primitive operations on matrices.

MATRIX (row1,...,rown) defines a rectangular matrix with the indicated rows. Each row has the form of a list of expressions, e.g. [a, x**2, y, 0] is a list of 4 elements.

ENTERMATRIX(m,n) allows one to enter a matrix element by element with the computer asking for values for each of the m by n entries.

EMATRIX(m,n,x,i,j) returns an m by n matrix with all entries zero except for the (i,j) entry, which is x.

DIAGMATRIX(n,x) returns a diagonal matrix of size n by n with the diagonal elements all x. An identity matrix is created by DIAGMATRIX(n,1), or one may use the next command.

IDENT(n) produces an n by n identity matrix.

SETELMX(x,i,j,m) creates a new matrix which is identical to the matrix m except that its (i,j) element is x.

ROWX(m,i) creates a new matrix which is the ith row of the matrix m.

COLX(m,i) creates a new matrix which is the ith column of the matrix m.

TIMEX(m1,...,mn) multiplies two or more matrices (or scalars and matrices).

ADDX(m1,...,mn) adds two or more matrices.

DIFFERENCEX(m1,m2) computes m1 - m2.

POWERX(m,i) computes the ith power of the matrix m

INVERX(m) inverts the matrix m.

TRANSX(m) produces the transpose of m.

ECHELON(m) produces the echelon form of m.

MINORX(m,i,j) computes the i,j minor of the matrix m

DETERMINANT(m) computes the determinant of m.

CHARPOLX(m,var) computes the characteristic polynomial for m. That is, DETERMINANT(DIFFERENCEX(m,DIAGMATRIX(var,size of m)).

SUBMATRIX(m1,...,mn, M, n1,...,nn) creates a new matrix composed of the matrix M with its mi rows deleted, and its ni columns deleted.

The Power Series Subsystem

The power series subsystem is divided into two parts. The first handles truncated power series, and the second manipulates summations in their general form. The code is still being changed to accomodate different needs as they appear in applications, so that the following description is liable to need revision. This subsystem is not, at the moment, particularly well integrated into the MACSYMA system, in the sense that there are no calls from the simplifier to these programs.

TAYLOR(exp,var,pt,pow) expands the expression exp in a truncated Taylor series in the variable var around the point pt. The terms through (var-pt)**pow are generated.

PS(exp,var,pt,pow) resembles TAYLOR, except that the internal form of the expression is a special form especially suitable for manipulation as a truncated expression. Such expressions can be manipulated with the programs PSPLUS, PSMINUS, PSTIMES, PSEXPT, and PSDERIV (for adding, negating, multiplying, raising to a power, and differentiating, respectively).

POWERSERIES(exp,var,pt) attempts to generate the general form of the power series expansion for exp in the variable var about the point pt (which may be INF, for infinity).

A large table of general expansions is now on a MACSYM disk file, including hyberbolic, hypergeometric, and various other special functions.

Miscellaneous Utility Commands

This section includes a few miscellaneous controls that the user has over the MACSYMA system. Some of these are not commands in the strict sense, since they do not require a "@" to take effect.

Control-B (written B̃) "seizes" the lineprinter, if it is available, and outputs future lines (in tandem with the console) on the lineprinter. Ẽ releases the lineprinter. K̃ prints on the next line the contents of the input buffer. This is useful when several characters have been deleted, to clarify current input line. $\tilde{L}$, on Cathode Ray Tube consoles, clears the screen and resumes printing on the first line (as in $\tilde{K}$).

MACSYMA is loaded into a LISP system. In the course of running programs, it is occassionally useful to call some LISP program directly. In order to read and evaluate one LISP s-expression, one has the command EVAL(). One may "quit" out of a loop and return to the supervisor by typing $\tilde{X}$. H has the effect of "breaking" at a point in execution, and allowing a user to examine the depth of program nesting, the values of internal variables, etc. $P <space> resumes execution.

$\tilde{G}$ causes an exit from MACSYMA to LISP. The LISP expression (CONTINUE) returns to the MACSYMA supervisor.

When within MACSYMA, it is sometimes handy to read in files of LISP programs. (Note that BATCH handles only MACSYMA input, not LISP files) The following command help one to do this.

LOADFILE(*fn1*,*fn2*,*device*,*username*) loads a file as described by its arguments.

The next few commands help file away MACSYMA results.

WRITEFILE(*device*,*username*) opens a file for writing.

CLOSEFILE(*fn1*,*fn2*) closes the file.

PLAYBACK() "plays back" all the input and output lines since (C1). PLAYBACK(*n*) plays back the last *n* expressions (Ci, Di, and Ei count as 1 each). Ordinarily this would be done between a WRITEFILE and CLOSEFILE to store a neat set of commands and responses or to refresh one's memory as to what he has already done (especially at a CRT console).

Comments are welcome, and should be directed to Richard Fateman.

Appendix II

The Polynomial and Rational Function Package in MACSYMA

II.1. Introduction

This appendix describes a series of LISP routines for manipulating sparse polynomials and rational functions in several variables. This description is intended as a detailed guide to the implementation and assumes some familiarity with LISP and the concept of rational expressions. A user-oriented view of these facilities may be found in the previous appendix.

The polynomial programs were originally written by W. A. Martin. They were debugged, and in some cases, rewritten, by R. Fateman; the rational function and conversion programs, the modular greatest-common-divisor (gcd) algorithm (4), and the lower level modular arithmetic routines were programmed by R. Fateman. The Berlekamp factoring algorithm (2) was implemented by L. Rothschild.

Because these routines are written entirely in LISP they can be exported to other LISP systems very simply. If they are to interface with another LISP-based algebraic manipulation system, the only programs that need to be altered are those which convert to and from rational expression form. The programs are written so as to allow complete symbolic algorithms to be composed entirely within the rational function package.

It is well known that any rational function can be written as the ratio of two polynomials with integer coefficients and with no common polynomial divisors. With the added provisions that the denominator be positive and that the polynomials be represented in a canonical form (such as recursive in the variables in some fixed order), we can produce a canonical form for any rational expression. MACSYMA has a special internal representation for canonical rational expressions (CREs) which has many useful properties, the most important of which is that this representation will map any set of functionally equivalent rational expressions into a unique (canonical) representation.

II.2. Representation of Canonical Rational Expressions

A rational function in canonical rational expression (CRE) form is represented at the top ("MEVAL") level of MACSYMA by the form

```
((MRAT SIMP varlist assoclist) polyform1 . polyform2).
```

Each polyform is a list of the form (mainvariable, highest-exponent, coefficient, next-highest exponent, coefficient, ...). Polyform1 is the numerator, polyform2 is the denominator.

When the coefficient of a term is zero, the exponent-coefficient pair is omitted. This makes the representation attractive for sparse polynomials. The coefficients may themselves be polyforms in variables with a lower order, or may be numbers. This recursive property makes this representation suitable for polynomials in any number of variables, and makes programming particularly simple in LISP. The leading coefficient in polyform2 is positive, and the greatest common divisor of polyform1 and polyform2 is 1. The ordering of variables on VARLIST determines which is the main variable, and which (recursively) are main variables of the coefficients of the main variable.

By altering the definitions of CPLUS, CMINUS, CTIMES, CEXPT, CQUOTIENT, CDIFFERENCE, CFACTOR, CDERIVATIVE, CGCD, and PCOEFP, the non-polyform coefficient arithmetic can be reimplemented with a number of different domains. Currently, the coefficient arithmetic is the domain of arbitrary precision integers.

If the global variable MODULUS is set to a positive prime number R, rather than its default value of NIL, all coefficient arithmetic is performed modulo R. In such a case, the representatives of the field have values between -R/2 and R/2.

Alternate coefficient routines have been implemented by R. Zippel which (along with minor changes in the rest of the programs) allow the coefficients to be rational numbers, or rational functions. Additionally, he has written programs which automatically truncate their results.

A set of special coefficient routines which have "counters" in them is also available. This is convenient for examining the number of coefficient operations required for execution of a program.

Among these coefficient routines, the only one whose purpose is not obvious is PCOEFP. PCOEFP is a predicate which returns T when applied to a member of the coefficient domain (presently, LISP integers), NIL for a member of the polynomial domain (i.e. polyforms).

Zero could consistently be treated as an empty polynomial or as a zero coefficient. For our purposes it is most convenient to express zero by 0.

Returning to the generalized form for a CRE, we see that it is to our advantage to have a rapid method for testing whether "x" is more of a main-variable than "y" (etc.). In order to compare two variables on the VARLIST quickly, each variable is associated with a generated symbol (they look like G001, G002, ...) and the generated symbols are ordered by their values. POINTERGP is used as an order-testing predicate. The generated symbols are used in the body of the polyforms, and the varlist and assoclist are used only in conversion to and from CRE form.

Examples

Assume the POINTERGP ordering of 3 generated symbols G001, G002, and G003 is in that order.

```
3/4 = ((MRAT SIMP NIL NIL) 3 . 4)

X/3 = ((MRAT SIMP (X) (G001)) (G001 1 1) . 3)

3/X = ((MRAT SIMP (X) (G001)) 3 G001 1 1)
```

$(X^2+XY)/Z$ = ((MRAT SIMP (Z Y X)(G003 G002 G001))
(G001 2 1 1 (G002 1 1))
G003 1 1)

Polynomials are written with polyform2 = 1

```
3 = ((MRAT SIMP NIL NIL) (3 . 1))

7X = ((MRAT SIMP (X)(G001)) ((G001 1 7) . 1))

0 = ((MRAT SIMP NIL NIL) (0 . 1))
```

Note that both the assoclist and varlist may contain variables not actually in the expression. Thus

```
 3 = ((MRAT SIMP (X Y Z)(G003 G002 G001) (3 . 1))
```

is valid.

II.3. Polynomial Functions

In this section, X and Y are expressions with the same ordering of variables.

PCOEFP(X) returns T if X is a member of the coefficient domain, i.e. a number or NIL.

PPLUS(X,Y) adds two polyforms to yield a polyform.

PTIMES(X,Y) multiples two polyforms to yield a polyform.

PQUOTIENT(X,Y) divides X by Y to yield a polyform; signals an ERROR if the remainder is not zero.

PDIVIDE(X,Y) yields a list of two RATforms: the quotient and the remainder, with respect to the main variable of X, of X divided by Y.

PDIFFERENCE(X,Y) yields the value of X-Y.

PDERIVATIVE(X,VAR) yields the polyform equal to the formal derivative of X with respect to the variable VAR, which need not be the main variable of X.

PEXPT(X,N) raises the polyform X to the power N, which must be a non-negative LISP integer. It uses a modified multinomial expansion technique which is generally superior to other methods (13).

OLDGCD(X,Y) yields the polynomial greatest common divisor of X and Y. Collins' reduced PRS algorithm, described in (27), p. 372 is used.

OLDCONTENT(X) yields a list of the (positive) content of X and the primitive part of X. X is considered to be a polynomial in one variable with coefficients that are polynomials in the other variables. Thus the content of xyz+xy with x the main variable is gcd(yz,y) or y, while the primitive part is xz+x. This definition is used for the Collins algorithm.

PGCD(X,Y) yields the polynomial greatest common divisor of X and Y. The modular algorithm described by Brown in (4) is used. It calls PGCDM, PGCDP, and PGCDU corresponding to algorithms M, P, and U in (4).

PCONTENT(X) yields a list of the (positive) content of X and the primitive part of X. X is considered to be a polynomial in many variables with integer coefficients. The content is always an integer. Thus the content of xyz+xy is 1, with primitive part xyz+xy. This definition is used for the modular GCD algorithm (PGCD).

PMODCONTENT(X) is a peculiar type of content calculation required by the multvariate modular gcd. X is considered to be a polynomial whose coefficients are polynomials with modular coefficients in one variable (the main variable in X). Thus the content of xyz+xy for main variable x, is x, with primitive part yz+y.

PMINUS(X) yields -X, 0 if X is 0.

PMINUSP(X) yields T if the leading coefficient of X is negative, otherwise, NIL.

PINTERPOLATE(L,VAR) finds an interpolation polynomial given a set of points. L is a list of n values (integers or polynomials) of a polynomial P to be found by interpolation at the points 0, 1, ..., n. VAR is the main variables of P. PINTERPOLATE returns P only if P has integer coefficients, and signals an ERROR otherwise. PINTERPOLATE is used by PFACTOR.

PCSUBST(X,VAL,VAR) substitutes the number VAL for the variable VAR in the polyform X.

PFACTOR(X) returns a list of items consisting of positive (except possibly -1) primitive, irreducible (over integers) factors of X, followed in each case by the degree (multiplicity) of that factor. Berlekamp's algorithm (2) is used. PFACTOR will not factor with respect to variables whose generated symbol is on the list $DONTFACTOR. If $FACTORFLAG is set to NIL, the integer part will not be factored. For example, with $FACTORFLAG set to T:

PFACTOR ((X 4 2)) = (2 1 (X 1 1) 4)

PFACTOR (0) = (0 1)

PFACTOR (1) = (1 1)

PFACTOR (-3) = (-1 1 3 1)

PFACTOR (-1) = (-1 1)

PFACTOR ((X 4 (Y 4 6))) = (3 1 2 1 (X 1 1) 4 (Y 1 1) 4).

PSQFR(X) is the same as PFACTOR except the polynomials are not necessarily irreducible, just squarefree (22, p. 381).

PMOD(X) returns the polynomial X with its coefficients reduced mod MODULUS. If MODULUS is NIL, X is returned unchanged.

II.4. Rational Functions

Here X and Y denote <u>ratforms</u>. A ratform is a dotted pair of polyforms, that is, the CDR of an MEVAL-level CRE. All results are in lowest terms. The denominator is always positive.

Division by zero will cause a LISP ERROR.

RATPLUS(X,Y) performs addition: X + Y.

RATTIMES(X,Y,SW) performs multiplication: X*Y. If SW is NIL, it will be assumed that if X=a/b and Y=c/d, that the greatest common divisor of a*c and b*d is 1. In situations where an expression is repeatedly converted to CRE form, a considerable amount of time can be saved by not repeating the g.c.d. calculations.

RATQUOTIENT(X,Y) performs division: X/Y.

RATINVERT(X) inverts X: 1/X.

RATMINUS(X) performs negation: -X.

RATDIF(X,Y) performs subtraction: X-Y.

RATEXPT(X,N) raises X to the Nth power, where N is a (possibly negative) LISP integer.

RATREDUCE(P,Q) takes two <u>POLY</u>forms P and Q and reduces them to lowest terms. RATREDUCE is used when needed by the other rational functions, except as noted in RATTIMES. RATREDUCE returns P/Q as a ratform.

RATFACT(X) factors the numerator and denominator of the ratform X, and returns a list similar to that returned by PFACTOR, except that the factors of the denominator will have negative multiplicities.

RATABS(X) returns the absolute value of the ratform X.

RATDERIV(X,VAR) returns the formal derivative of the ratform X with respect to VAR.

RATGCM(X,Y) returns the greatest common multiple of X and Y. If X = a/b and Y = c/d, then gcm(X,Y) = gcd(a,c) * gcd(b,d)/(b*d).

II.5. Conversion Functions

$RAT(M) uses NEWVAR, described below, to put non-rational subepressions of M on a variable-list (VARLIST), and then calls RATREP(M,VARLIST) as described below. Floating point numbers are converted to rational numbers (within a relative error of \$RATEPSILON, set by the user).

RATREP(M,VARLIST) given a non-CRE MACSYMA expression M creates the appropriate CRE form, including the (MRAT SIMP varlist assoclist) prefix. All the variables and non-rational expressions on VARLIST will be associated with generated symbols, listed on assoclist. All the non-rational elements and variables in M must be on the list VARLIST given to RATREP. See the discussion of NEWVAR for how this may be done.

\$RATDISREP(X) converts the CRE X into a standard MACSYMA prefix expression. A RATSIMP flag is put on each of the operators. Extraneous parts of the expression (such as multiplication by one, nested MPLUSs or MTIMESs, exponentiation by 0 or 1) are edited out automatically.

Example:

$2X^2+3XY+1$ = ((MRAT SIMP (Z Y X)(G003 G002 G001))((G001 2 2 0 (G001 2 2 0 (G002 1 (G003 1 3 0 1)))) . 1)) is \$RATDISREP'd to

((MPLUS RATSIMP)((MTIMES RATSIMP) 2 ((MEXPT RATSIMP) X 2)) ((MTIMES RATSIMP) 3 Z Y) 1)

NEWVAR(X) examines an expression M (a MACSYMA prefix form or a CRE, or a sum, product, difference, or quotient of the two), and constructs a list L of non-rational components and variables which are not already on the global VARLIST. That part of L collected from CREs is placed on the VARLIST in as nearly the same order as in the CREs as possible. The rest of L is sorted on the function GREAT (by the function SORT) and NCONC'd to the front of VARLIST. A subsequent call to RATREP with arguments M and the global VARLIST will then produce the CRE corresponding to M.

ORDERPOINTER(L) sets up the correspondence between generated symbols and the expressions on the list L, generating new symbols if L is longer than any previous VARLIST. These generated symbols, placed on the CDR of the value of the variable GENVAR, also form the assoclist for a newly RATREP'd expression.

II.G. Command-level Programs

\$RATSIMP(X) converts the expression X and all its non-rational subexpressions into CRE form, and then back again. This has the effect of rationally simplifying the expression X and all its non-rational expressions.

\$RESULTANT(X) computes the resultant of the polynomial X, using a polynomial remainder sequence technique.

\$GCD(X,Y) computes the greatest common divisor of the two polynomials X and Y.

\$DIVIDE(X,Y,V) calls PDIVIDE on the two polynomials X and Y, whose main variable is V.

\$FACTOR(X) factors the numerator and denominator of the rational function X over the integers.

\$GFACTOR(X) factors the polynomial X over the Gaussian integers (with sqrt(-1) = %I adjoined).

\$MOD(X) converts the single-variable polynomial X to modular representation, accessing the value of MODULUS set globally.

II.7. Other Notes

Efficiency can be considerably improved if the knowledge that a quotient is in lowest terms can be preserved. If the global flag NOREPEAT is set to T, then any MQUOTIENT or MTIMES with a RATSIMP flag (placed there by \$RATDISREP) will not be RATREDUCE'd as it is being converted to CRE form by RATREP. It may occasionally be useful to set NOREPEAT to NIL, since E = (x-1)/(z+1) is reduced, but if $z = x^{1/2}$, E can be reduced further on a second pass by treating it as $(z^2-1)/(z+1)$.

Setting \$GCDSWITCH to NIL replaces the modular GCD algorithm with the reduced PRS algorithm. Setting \$GCDOFF disables the gcd routines entirely (all gcds are 1). Setting \$BERLEFACT to NIL replaces the Berlekamp factoring algorithm with the less efficient Kronecker algorithm.

Listings of these functions and their subroutines are are available from the author.

Bibliography*

1. ACM. Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, Calif., March, 1971. (This volume will be referred to as SYMSAM II.)

2. Berlekamp, E. R. "Factoring Polynomials over Large Finite Fields," Math. Comp. 24, no. 111, July, 1971 (713-736).

3. Brown, W. S. "Rational Exponential Expressions and a Conjecture Concerning pi and e," Amer. Math. Monthly 76, Jan., 1969, (28-34).

4. --. "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors," SYMSAM II, also J. ACM 18, no. 4, Oct., 1971, (478-504).

5. Caviness, B. F. "On Canonical Forms and Simplification," doctoral dissertation, Carnegie-Mellon University, 1967.

6. --. "On Canonical Forms and Simplification," J. ACM 17, April, 1970, (385-396).

----

7. Carraciolo di Forino, A. _et al_, "PANON-IB -- A Programming Language for Symbol Manipulation," University of Pisa, Italy, 1966.

8. Christensen, C. "Examples of Symbol Manipulation in the AMBIT Programming Language," _Proc. ACM 20th National Conference_, Cleveland, Ohio, 1965, 247-261.

9. Collins, G. E. "SAC-I System: An Introduction and Survey," SYMSAM II.

10. Eisenpress, H. and Bomberault, A. "Efficient Symbolic Differentiation Using PL/I FORMAC," IBM New York Scientific Center Technical Report No. 320-2956, New York, Sept., 1968.

11. Engeli, M. "Achievements and Problems in Formula Manipulation," _Proc. IFIP Cong. 68_, Invited Papers, North-Holland Publ. Co., Amsterdam, 1968, (79-84).

12. Farber, D. _et al_, SNOBOL, A String Manipulation Language, _J. ACM 11_, no. 1, Jan., 1964, (21-30).

13. Fateman, R. "On the Computation of Powers of Polynomials," _SIGSAM Bulletin_ (to appear).

14. --. "The User-Level Semantic Matching Capability in MACSYMA," SYMSAM II (311-323).

15. --, and Moses, J. "Canonical Forms for First Order Exponential expressions," in preparation.

16. 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; now available from the Clearinghouse, document AD-657-282.)

17. Gentleman, W. M., "Optimal Multiplication Chains for Computing a Power of a Symbolic Polynomial," Dep't of Applied Analysis and Computer Science Research Report CSRR 2035, University of Waterloo, Waterloo, Ontario, March, 1971, also SIGSAM Bulletin no. 18, April, 1971 (23-30), also Math. Comp. (to appear).

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

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

20. --. "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.)

21. Heile, Francis B. "Photon Propagation Through Strong Uniform Magnetic Fields," Master's Thesis, MIT, 1971.

22. Heindel, Lee E., "Computation of Powers of Multivariate Polynomials over the Integers," Bell Telephone Laboratories, Holmdel, N.J., Feb., 1971.

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

24. Johnson, S. C. "On the Problem of Recognizing Zero" SYMSAM II, also J. ACM 18, no. 4, Oct., 1971, (559-565).

25. Kleiman, S. L. "Computing with Rational Expressions in Several Algebraically Dependent Variables," Bell Telephone Laboratories, Murray Hill, N. J., 1965.

26. Knuth, D. E. The Art of Computer Programming, vol 1, "Fundamental Algorithms," Addison-Wesley Publ. Co., 1968.

27. --. The Art of Computer Programming, vol 2, "Seminumerical Algorithms," Addison-Wesley Publ. Co., 1969.

28. London, R. "Proving Programs Correct: Some Techniques and Examples," BIT 10, no. 2, 1970, (168-182).

29. Manove, M., Bloom, S. and Engelman, C. "Rational Functions in MATHLAB," Memo MTP-35, The MITRE Corp., Bedford, Mass., August, 1966.

30. Martin, W. A. "Symbolic Mathematical Laboratory," doctoral dissertation, M.I.T., 1967. (also appeared as report MAC-TR-36, Project MAC, M.I.T., Cambridge Mass., 1967; now available from the Clearinghouse, document AD-657-283.)

31. --, and Fateman, R. "The MACSYMA System," SYMSAM II (59-75).

32. McBride, F. V., Morrison, D. J. T., and Penguelly, R. M. "A Symbol Manipulation System," in Machine Intelligence 5, B. Meltzer and D. Michie, editors, American Elsevier Publ. Co., N. Y., 1970 (337-347).

33. McCarthy, J., et al. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass., 1965.

34. Moses, J. "Algebraic Simplification, a Guide for the Perplexed," SYMSAM II, also Comm. ACM 14, no. 8, Aug., 1971, (527-538).

35. --. "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," SYMSAM II, also Comm. ACM 14, no. 8, Aug., 1971, (548-560).

36. --, Rothschild, L. P., and Schroeppel, R. "A Zero-Equivalence Algorithm for Expressions Formed by Functions Definable by First Order Differential Equations," in preparation.

37. 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.

38. Ralston, A. A First Course in Numerical Analysis, McGraw-Hill, N. Y., 1965.

39. Richardson, D. "Some Unsolvable Problems Involving Elementary Functions of a Real Variable," J. Symb. Logic 33 1968, (514-520).

40. Risch, R. "The Problem of Integration in Finite Terms," Trans AMS 139, May, 1969, (167-189).

41. --. "The Solution of the Problem of Integration in Finite Terms," IBM Watson Research Center, N.Y., (submitted to Bull. AMS.)

42. Sammet, J. "Revised Annotated Descriptor-Based Bibliography on the Use of Computers for Non-numerical Mathematics,"in Proc. of the IFIP Working Conference on Symbol Manipulation Languages, North Holland Publ., 1968, pp. 358-484. This bibliography first appeared in Comp. Rev. 7, (1966), (B1-B31). Updates are published periodically in the Bulletin of the Special Interest Group on Symbolic and Algebraic Manipulation (SIGSAM) of the ACM; the bibliography is presently being maintained by John C. Wyman of Syracuse University.

43. 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.)

44. 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.)

45. Tobey, R. "Experience with Formac Algorithm Design," Comm. ACM 9, no. 8, Aug., 1966, (589-597).

46. Tschebetarow, N. Grundzuge der Galois'chen Theorie, P. Noordhoff, N. V., 1950.

47. van der Waerden, B. L. Modern Algebra, vol. 1, Frederick Ungar Publ. Co. N. Y., 1953.

48. Wilson, L. R. "Hypergeometric Functions in MATHLAB," M.I.T. Artificial Intelligence Group Memo 196, June, 1970.