A Review of Mathematica

RICHARD J. FATEMAN*
(*fateman@cs.Berkeley.EDU*)
Computer Science Division, University of California, Berkeley, CA 94720, USA

(*Received: 16 November 1990*)
(*Revised: 16 September 1991*)

The Mathematica computer system is reviewed from the perspective of its contributions to symbolic and algebraic computation, as well as its stated goals. Design and implementation issues are discussed.

1 Introduction

The Mathematica[1] computer program is a general system for doing mathematical computation described in Wolfram (1988, 1991). It includes a command language, a programming language, and a calculation environment that is oriented toward symbolic as well as numeric mathematics.

The back cover of the manual (Wolfram (1991)) provides excerpts from rave notices like "The importance of [Mathematica] cannot be overlooked ... it so fundamentally alters the mechanics of mathematics." —The New York Times. *Fortune* says "... it will do, instantaneously, virtually all of applied mathematics ... " Taubes (1988).

Hype aside, the program is without question interesting to mathematicians, computer scientists, and engineers because of its combination of a number of technologies that have arisen in initially separate contexts—numerical and symbolic mathematics, graphics, and modern user interfaces. The exploitation of the PostScript language for plotting contributed to its natural fit into the package of programs initially released for the NeXT workstation.

Not all commentary on Mathematica has been uncritical. For example, reviews in *Science* (Foster & Bau (1989)) and *Notices of the AMS* (Herman (1988), Simon (1990)) compare Mathematica's features, reliability and efficiency to similar programs for algebraic and/or numerical interactive manipulation. Additional commentary on the program includes Hoenig (1990), Vogel (1989). A perceptive book review of the reference manual (first edition) has also appeared (McCurley (1990)). Electronic-mail messages on various semi-public bulletin boards (in particular `netnews: sci.math.symbolic`) have discussed features and bugs of Mathematica as well as similar programs. There is also an active mailing list specifically for Mathematica users (`mathgroup@yoda.ncsa.uiuc.edu`). Such forums provide opportunities for valuable exchanges but, especially as "subscribers" become more numerous, the continuing unedited message streams overwhelm the large picture. Therefore there appears to be value in a more widely available and more detailed commentary on Mathematica, specifically from the perspective of its context and contribution to technology.

*This work has been supported in part by the following: the National Science Foundation under grant numbers CCR-8812843 and CDS-8922788, through the Center for Pure and Applied Mathematics and the Electronics Research Laboratory (ERL) at the University of California at Berkeley; the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through ERL; and grants from the IBM Corporation, the State of California MICRO program, and Sun Microsystems.

[1] Mathematica is a trademark of Wolfram Research Inc. (WRI).

In order not to keep the reader in suspense, my major conclusion is that Mathematica has many flaws. Some of them are substantial and are unlikely to be repaired because they reflect decisions rather than oversights. The gaps between claims and actuality are substantial. These gaps are not all inherent in the nature of mathematical algorithms or representations since competing commercial programs often provide correct answers where Mathematica fails.

One way of improving the state of the art in automating mathematics is to examine current programs critically; our purpose in this review is, in part, to direct attention to shortcomings and suggest improvements.

2 Preliminaries

This review occasionally assumes the reader has a more-than-casual familiarity with Mathematica, and is certainly no substitute for a primer on the subject. Careful study of the major reference (Wolfram (1988, 1991)) may be an adequate substitute for experience in using the program.

Four themes permeate this review corresponding to areas of technology to which Mathematica potentially could make a contribution:

- the field of symbolic and algebraic computation,
- the field of numerical computing,
- programming language and human-computer interface design, and
- the organization of mathematical information.

These should be kept in mind as the discussion proceeds, but first a brief historical perspective seems in order.

3 Prior Art in Mathematica

The idea, of using computers for symbolic rather than numerical (arithmetical) computation, actually predates the electro-mechanical computer. Ada, Countess of Lovelace and patron of Charles Babbage, inventor of the "Analytical Engine", suggested such usage in 1844 (see Knuth (1969)). It took over a century for the first actual symbolic computation to be cited in the literature. (see van Hulzen & Calmet (1983) or Barton and Fitch (1972) for a survey.)

The more recent tradition of the field of Symbolic and Algebraic Manipulation (SAM) by computer has had a small but loyal following at least since the early 1960's. Members of the Association for Computing Machinery (ACM) joined together to form a Special Interest Group (SIGSAM) in 1965.

Chronologically, Mathematica probably should be considered as about "third-generation" among algebra systems, placing it among the (more-or-less) contemporary general-purpose systems such as Derive (The Soft Warehouse (1991)), Maple V (Symbolic Computation Group 1990), and AXIOM, (previously referred to as Scratchpad II (Computer Algebra Group (1988))). These are not necessarily better than the second-generation hold-overs — in particular, Macsyma (Moses (1979), Mathlab Group (1983), Pavelle (1985), Fateman (1989)), provides answers when Mathematica sometimes does not; another second-generation system with a wide following is Reduce 3 (Hearn (1984)). These recent systems were built upon research results plus practical experience in using second-generation systems of the late 1960's, including ALTRAN, CAMAL, PL/I-FORMAC, Mathlab, Scratchpad I, Symbolic Mathematical Laboratory, SAC-1, SIN, and others. An early comparison

of some of these systems is still worth reading for background (Barton and Fitch (1972)). Stepping back further in time, the first generation systems of the early-to-mid 1960's included ALPAK, FORMAC, Formula Algol, PM, SAINT, SNOBOL and LISP. There are also numerous other "special purpose" systems such as Schoonschip, Sheep, Trigman and Cayley referenced in Buchberger (1983).

Although there are ample historical precedents for Mathematica's symbolic facilities, and a clear intellectual debt, there is virtually no acknowledgment of prior software or algorithms in Wolfram's reference (Wolfram (1988, 1991)). Indeed, the first edition (p. xvvii) indicates in passing that Mathematica "represents a synthesis of several different kinds of software" including some 16 systems. These are not mentioned in the second edition at all. Algorithm documentation receives similar treatment. If you want to know about the "Risch algorithm" (mentioned on page 528 of the first edition, expurgated from all but the index for the second edition) or how factoring is done, you won't find any information or references.

Although space limitations prevent us from providing details and full references, it is clear that recent developments in other areas have either inspired or paralleled the facilities in Mathematica. These include user interfaces such as the Macintosh environment or other window systems; alternative mathematics display and manipulation systems such as Mathscribe (Soiffer & Smith (1986)), CaminoReal (Arnon *et al.* (1988)), Theorist (Bonadio (1990)), Milo and FrameMaker (Avitzur (1988)); operating systems and languages with interprocess communication, display technology (PostScript in particular); and programming language ideas (object-oriented programming, pattern matching, functional programming).

An important point here is that Mathematica arrived riding the crest of a wave: a mass market newly-formed from the appearance of vastly improved low-cost computing hardware. That mass market was a large part of what Mathematica's precursors (Macsyma, Reduce, Maple, etc.) lacked. Mathematica became news partly because it was new, and not because it was that much better than its predecessors.

In summary, Mathematica is more evolutionary than revolutionary. Not only does it depend heavily on unacknowleged prior art and technology, it is in many respects not so advanced as older systems.

4 Examination of the Objectives

Stephen Wolfram, the principal designer of the system and author of the user documentation (Wolfram (1988, 1991)) for the system, specifies several objectives for Mathematica which are summarized or paraphrased below. This review addresses each of the objectives. Occasionally there will be a comparison to other systems; however, the claims of Mathematica are made in relation to mathematics, and rarely with reference to other computer programs. It seems appropriate to try to review the system in that light.

Objectives of Mathematica

- To provide a system for doing interactive symbolic mathematical calculations (interactive Mathematica); (§5)
- To provide a repository for mathematical exposition and education (Notebooks); (§6)

- To provide a programming language which unifies ideas from procedural programming, functional programming, rule-based programming, object-oriented programming and constraint-based programming; (§§7, 8)
- To provide facilities for exact symbolic computation and arbitrary-precision numerical computation; (§§9, 10)
- To provide a repository for information on the simplification and manipulation of mathematical functions, polyhedral objects, etc. (Libraries); (§11)
- To provide high-quality plotting from algebraic or discrete computations in a format that can be further manipulated (PostScript); (§12)
- To provide built-in functionality (data types) for algebraic manipulation of formulas comprising polynomials, rational functions, the usual functions of elementary calculus, advanced functions of physics ("special functions"), functions of number theory, combinatorics, as well as composite data structures (lists, matrices, literal functions and arguments, etc.) and debugging; (§§13, 14)

In the final sections we discuss some general issues about the relationship of Mathematica to mathematics.

5 The Interactive System

Mathematica is intended to be used primarily as an interactive program supporting the day-to-day computational needs of a mathematician or scientist. For the most part it fits into a traditional model typical of the last 25 years of time-sharing, or the last 10 years of standard data entry into personal computers. The user types a line or more, and after some computation a display is produced. Given the possibilities that have developed recently, it is somewhat surprising that Mathematica hasn't progressed much beyond line-at-a-time input; for an example of what more could be done, see Milo (Avitzur (1988)) or Theorist (Bonadio (1990)), or MathScribe (Soiffer & Smith (1986)) or even the considerably older research system DREAMS (Foster (1984)). Each of these systems allows some use of pointing devices for selection of mathematical expressions. (By contrast, experimental input systems based on tablets and character recognition have just recently resurfaced as "palm-top" computer systems. The intuitive attractiveness of hand-writing of mathematics for raw input—rather than keying in new text and selecting already-displayed material — has never been exploited successfully in past experimental systems. Experience indicates that a combination of typing and pointing seems to work better.) In editing of commands, Mathematica allows selections only of linear text-strings. In its parser as well as its display of equations (limited to fixed-width character-grid typewriter-font "2-D" expressions) Mathematica seems more like Macsyma (circa 1968) than a system taking advantage of bit-mapped workstations.

Because of the interactive nature of most uses of Mathematica, the intuitiveness of the system and the user-visible programming language is very important. The two components are discussed in separate sections on the display (in the next section) and a more extensive subsequent section on the programming language.

An advertised novelty in Mathematica is its separation of a front-end "interactive" component from a back-end "computational" component. In practice both parts of the program often run side-by-side in the same computer, but in principle, they could be running on distinct machines. Mathematica was not the first system to try this separation since there were prior experiments with the Maple system, and indeed the current commercial version (Maple V) has such a separation. Even so, the separation in Mathematica has

still-unrealized potential. Feedback from mouse-input on plots has been demonstrated on some platforms, but only in recent versions of the program, or on advanced workstations (e.g. Silicon Graphics' systems).

Overall, the notion of (say) a Macintosh front-end attached to a supercomputer back-end sounds better than it really is in practice. Supercomputers generally do not run symbolic mathematics programs particularly well. Running both front- and back-end parts of Mathematica on a fast remote computer works well, assuming there is a "front-front-end" such as an X11 window system to display the graphics. Since such a window system is probably in use anyway, and it can be totally ignorant of and unlicensed for Mathematica, it may be preferable.

Mathematica is potentially appropriate for use as an interactive front-end to other programs written in C or other languages. Through a "foreign function" interface, it is possible to link to other systems. It appears to be non-trivial to get this to work, however.

6 Notebooks and the Display

Mathematica runs on a variety of machines, and the quality of the user interface varies across a spectrum from the universal but workable ASCII-terminal mode to specially-tuned versions for the Apple Macintosh and NeXT lines of computers.

The most sophisticated interface model provided is called a Notebook. The user types commands as text into an outline processor. That is, there is an option of suppressing details of display of material at lower levels. The computer generates additional text and displays into the outline. The graphics sections can be re-displayed as PostScript and edited. Since the Notebooks can be exchanged between computers, there is a simple technique for reproducing results and building up a library, at least if the Notebooks are properly constructed so as not to conflict in the user's space of objects. Although this is undoubtedly a useful approach, in some respects it is not as advanced for modeling of mathematical problem solving as some other programs such as Milo (Avitzur (1988)) or Theorist (Bonadio (1990)). These programs offer facilities missing in Mathematica: they allow the "text" parts of the mathematics to be data objects in a system where algebraic expressions can be selected, manipulated, linked to other expressions, etc. They provide interactive type-setting of mathematical expressions. In addition, Theorist supports animation and rotation of surfaces. Although Mathematica provides these latter facilities on a subset of its platforms, it lacks the pencil-and-paper quality of interaction that these other products offer.

Although it is possible to import descriptive material from other programs into a Notebook, including digitized pictures or typeset formulas, the linkage is rather roundabout. Perhaps in the future it will be possible to run a `TeXForm` version of an equation through TEX and put it directly into the notebook. This would certainly create a better environment for Notebooks as an alternative mode of publication of mathematics. In fact, there are nine books cited by Wolfram (Wolfram (1991) page xviii) which describe the use of Mathematica in scientific or educational contexts, and some of them are clearly dependent upon Notebooks as an interface. The design of Notebooks seems more supportive of presentation of information than interaction, and this may be just fine. One colleague finds the Notebooks to be the best new feature of Mathematica compared to other symbolic mathematics systems. On the other hand, another serious Mathematica user indicated to me that he found the Notebook front-end to be a hindrance. It may very well be a matter of previous experience and developed preferences.

Incidentally, a Notebook will likely contain only a fragmentary record of computations so it departs somewhat from the tradition of the "laboratory notebook" containing all raw data, experimental results etc. It is more like a showcase.

7 Programming Language

7.1 Overview

The challenge of making computers truly useful (and perhaps making programmers obsolete) is often couched in terms that make it sound like a programming language issue: All one would need is to create the right syntax and semantics to banish all the problems of applications programming.

That solution is not here yet, but the approaches to the challenge through the years can be characterized as mixtures from three streams:

- The elaborate all-inclusive language (like PL/I, Ada, Common Lisp)
- The extensible language (C, Algol-68, Common Lisp)
- The language oriented to a specific application (PostScript, JCL).

Mathematica falls mostly in the first camp in that it has adapted in some way nearly every construction appearing in some general language, and it certainly is elaborate. But it includes some extension techniques, and has certainly packaged together some application-specific subroutines. The evidence to date is that, superficially at least, Mathematica is actually fairly comforting to an experienced programmer—most of the familiar tools, as well as some others, are there. There are generally several ways of writing "equivalent" programs using different programming paradigms. More so than in other languages, different paradigms may differ by orders of magnitude in resource consumption. Since no particular programming style is imposed on the programmer, some programmers will use the numerous syntactic shortcuts to produce "write-only" programs (so called because they become incomprehensible shortly after being written). Indeed, the fact that there are so many variations possible is especially discomforting for a programmer concerned about efficiency. Because there is such limited information available on the internal algorithms and data structures in Mathematica, there is sometimes no alternative to trying various versions of an algorithm and timing them. The fact that details might change is not a good excuse—such information could be part of release notes, for example.

Mathematica does not have an extensible syntax. It uses all the non-alphabetic symbols of the ASCII character set, and quite a few multi-character symbols. Mathematica's designers did not choose the more conventional wisdom that it may be advisable to leave some characters "for the users" for possible syntax extension. Techniques for such extension are fairly easy to adopt using the parser model used by both Macsyma and Reduce (see Pratt (1973)), and is advocated in Common Lisp, as well. Mathematica builds a complete syntactic box, for good or ill. You can't tinker with it unless you write a new front-end processor.

From a mathematician's point of view, J. R. Kudera (Kudera (1988)) comments "... computer mathematics *languages* are ghastly to use" and that "Problems that lend themselves to this kind of computation [user-written programs] simply do not occur often enough to allow users to develop proficiency." Thus intuitiveness is important: when a system deviates substantially from common mathematical notation and semantics as well as from conventional programming, it becomes positively hazardous.

Nevertheless, programming seems inevitable since the system constructors simply cannot anticipate all needs. In the next several sections we look in more detail at various aspects of Mathematica's approach to the support of programming.

7.2 Object-Oriented Programming

The programming language specialist may observe that Mathematica's version of this popular supports neither hierarchies nor inheritance. This omission considerably weakens the faithfulness to the notion of object-orientedness (for those who care). Type-based dispatch of operations is nicely integrated linguistically with the pattern matcher, however. What the programmer may think of as a function definition is, in some senses, equivalent to a pattern-match and replacement rule associated with an object, usually the main operator (or `Head`) of the function, but alternatively one of its operands. Thus instead of re-programming some central simplification routine to handle a new user-defined symbol `f[...]`,and its arguments it is possible to associate, in some piecewise fashion, simplification rules with `f` itself. For example `f[x_Integer]:= 0 /;x<0` defines simplification of `f` at negative integer values. The approach of using some kind of "local" control based on the operator (e.g. `f`) for simplification is actually fairly common and appears in one of the first algebraic simplification programs, Korsvold (1965).

7.3 Contexts and Information Hiding

Any modern programming language intended for building large systems must provide information-hiding capabilities. Mathematica uses its notion of Contexts for this purpose. The major construction for modular system building appears to resemble packages in Common Lisp, and even uses the delimiters `BeginPackage` and `EndPackage`. In Mathematica it is possible to delimit sections of code by `Begin` and `End` brackets, identifying a context in which public names (those exported to external or Global contexts) and private names (those local to this context) are separated. Unfortunately, this is less effective than one might wish, because entering and exiting a Context (by setting the `$ContextPath`) does not have the effect one might expect.

For example, one might wish to assert the rule that logs of products should be re-written as sums of logs.

```
Log[x_*y_] := Log[x] + Log[y]
```

But just saying this in the Global context is dangerous. In particular, system programs that rely on a certain behavior from `Log` may be damaged. Therefore one might wish to contain it in a context, for example:

```
Begin["logsimp`"]
Log[x_*y_] := Log[x] + Log[y]
End[]
```

However, this rule is placed on the Global `Log` symbol, rather than the `logsimp`Log` symbol, and the system does not distinguish between a rule that rewrites `Log[x_*y_]` and one which re-writes `Log[logsimp`x_*logsimp`y_]`. Thus this packaging does not limit the effect of the Log rule, and one must presumably explicitly delete and re-assert such rules when needed (or explicitly apply them when appropriate). Merely asserting them once for all time leads to generally unforeseeable consequences. We found, for example, such a rule had the effect of breaking the `Integrate` command.

This appears to limit severely the utility of rule-based programming as a technique for adding general information to the system. The programmer has three choices:

- Avoid the use of any of the same function symbols as the system;
- Use patterns only as a front-end to the Mathematica system;

- (and/or) Use patterns only as a back-end to the real Mathematica system.

  This front or back-end usage might include converting all `Log`s to `log`s for special simplification, and then converting back to `Log`s.

Perhaps another example will illustrate the difficulty of this approach: If you teach the system that $x + 1 > x$, then the system does not use this information to compute $\max(x, x+1)$. How can one fix this short of reprogramming the system function `Max` as well as all the proprietary system functions that use some internal version of comparison? By the way, this obvious inequality is not true for all possible x representable in Mathematica; Consider $x = \infty$.

A subtle point, perhaps not intended to be noticed by the casual fan of Mathematica, is that the rule on the back cover of Wolfram (1988) defines simplification for `log`, not `Log`, thereby not interfering with built-in rules. This trick of using lower-case names does not work very well if one wishes to alter, by rules, any built-in functions, or functions like `Factorial` or `Plus`, which do not have lower-case equivalents in their usual representation as `!` and `+` respectively.

It is possible to group rules and apply them together, as illustrated by programs in the on-line library as well as in Maeder (1988). Such grouping of rules as in the trig-simplification routines eliminates "cross-talk" by limiting scope. Unfortunately, such tricks weaken the possible synergy of rules. If the idea behind rules is to have them take effect when appropriate, without specific attention by the programmer, the necessity for grouping vitiates the concept.

One experienced Mathematica hand advised me not to modify any built-in functions. This simple piece of advice carries with it implicitly the idea that if you don't like what Mathematica does with the `Log` function, you are free to program anything and everything you want about the `log` function. But then you'll have to change `Integrate` which returns answers in terms of `Log` to (say) `integrate` which returns answers in terms of `log`. If you choose to differentiate the result, you have a choice of changing `log` to `Log` temporarily, or propagating your changes throughout your program: defining rules for the differentiation and numerical evaluation (etc.) of `log` – in effect writing a "shadow" Mathematica. This is made rather difficult because the system's internal functioning is hidden for proprietary reasons. Even if you are willing to pay the substantial penalty in performance, you cannot tell how much functionality must be recreated.

An attempt to make extensive use of Mathematica rules in defining a system using abstract data types is reported by Buchberger (1991) who found it resulted in frustratingly slow computation.

Two other shortcomings in Contexts are worth noting: Mentioning a name before reading in the package defining it shields the name in the package from the global environment, effectively disabling the package. Debugging, never easy in Mathematica, becomes even harder when packages and contexts are involved. Also, printing out a program defined in another context (see, for example, the data associated with the `Bessel` functions in version 1.2) involves the repeated display of fully-qualified and rather lengthy context names.

7.4 Spaces mean Multiplication

This is perhaps a minor point, but annoying in its own way. In the Mathematica programming language, one can use spaces or even adjacency to signal multiplication. This idea, used by Wolfram in his earlier SMP system, is initially appealing—that one can simply write `2 x` or even `2x` instead of `2*x`. It looks like the traditional mathematical convention. But is it? Consider that it implies that `sin x` or even `sin(x)` is a product, equal to `x * sin`. The Scratchpad II system (Computer Algebra Group (1988)), following

another mathematical convention, interprets `sin x`, `sin(x)` and `sin.x` as function applications. Mathematica requires the syntactic construction `sin[x]` or `sin@x` or `x//sin` for function application, and just to make sure you use the built-in function, you must capitalize the first letter: `Sin[x]`. This departure from conventional notation may be of special concern to a teacher whose students may be struggling with notation in the first place. There is a weak argument that most potential users have not had experience "typing" conventional mathematics in *any* notation, and therefore requiring square-brackets may not be objectionable.

What else goes wrong, though?

A few things. For example, `a++ * ++b`, which to those familiar with the C programming language appears to be the computation of `a*(b+1)` leaving `a` set to `a+1`, as well as `b` to `b+1`, cannot be written in Mathematica as `a++ ++b`, since *that* space does not stand for multiplication. Rather it stands for nothing: Mathematica parses this expression as the necessarily meaningless `(a++)++ * b`. Even the author of the output-printing program was confused on this, since echoing back the first expression (by typing `Hold[a++ * ++b]`) displays the non-equivalent `Hold[a++ ++b]`. This particular inconsistency of the display with the internal form was reported as a bug some time ago, but its existence suggests a design error in that the parser and display should not have access to inconsistent precedences. In a well-designed system the two closely related subsystems would use the same precedence data, stored in one place.

The precedence of the space as multiplication is not adhered to. (As another example, `3! ++a` results in an error: `Factorial is write protected`).

As a matter of clarity, I suspect the physicist who is used to writing $1/2\pi$ may be lured into writing `1/ 2Pi` which is actually the rather different $\pi/2$.

Finally, the accidental omission of a semicolon or comma will not be caught by a syntax check. The forms `f[a b]`, `f[a,b]` and even `f[a;b]` are all unexceptional syntactically.

My conclusion is that insisting on the use of an asterisk is preferable, because leaving it out brings up too many problems. One reader suggests as an alternative inserting parentheses when there is any doubt. This advice is hardly likely to be followed by those who need it.

7.5 Other Syntactic oddities

The version 1.2 valid input form `(a,b)` is not documented. It is not a `List`, but `Sequence[a,b]`. Sequences are most naturally produced by pattern matches involving a collection of arguments. (If `f[x__]` is matched to `f[a,b,c]` then `x` is `Sequence[a,b,c]`) A Sequence has the remarkable property that `f[1,2,Sequence[a,b]]` means `f[1,2,a,b]`. One consequence is that `f@(a,b)` is mapped to the same form as `f[a,b]`, but the perhaps easily mistyped `f(a,b)` is somewhat unexpectedly mapped into `a*b*f`. (Explanation: `f(a,b) = f*(a,b) = Times[f, Sequence [a,b]] = Times[f,a,b]`). If you type `f(0,x)` you get `0`. A simple fix for this problem is to be less clever and forbid the alternative input-syntax of `(a,b)` for `Sequence[a,b]`. Version 2.0 has adopted this fix. You still should beware that `f(0)` is 0, regardless of the definition of `f`. Note that the `Head` function does not work on a `Sequence` since `Head[Sequence[a,b]]` is equivalent to `Head[a,b]`, which is meaningless.

As another example, if you want to redefine factorial for certain values, you might think to do

```
Unprotect [Factorial]; big!=bigfact
```

but that won't work. Here you need the space:

```
big! =bigfact
```

because the language includes a not-equals operator (`!=`). Finally, the expression

```
4/. 4->5
```

which means "substitute 5 for 4 in the expression 4," returns `5`. It has a rather different meaning from the expression without the space in it;

```
4/.4->5
```

which returns the rule `10.->4` because `4/.4` is `10`. What's the point in all this? Simply that it is potentially quite confusing to see a modern programming language design in which the meaning of "white space" is not only significant but has puzzlingly different meanings depending on context.

7.6 Procedures, Functions, Patterns, Efficiency

There are attractive aspects to the method used to define procedures. Were it not for the difficulties caused by errors in semantics, and the almost inevitable inefficiency which results from the reliance on matching, it would be even more attractive. In a nutshell, the approach is to

- Incorporate control structures from all of C, Lisp, APL and functional programming.
- Unify the three notions: a mathematical function $f(x)$; a pattern `f[x_]`; and a procedure invocation. Thus $f(3)$ is "implemented" by a pattern match of `f[3]` which results in a temporary binding of x to 3. The evaluation of the right-side of a rule provides the semantics for the function. (Mathematica has other ways of viewing function objects meant as programs, based on the lambda calculus and Lisp. You can express $\lambda(x, y).(x + y)$ as `Function[Plus[Slot[1], Slot[2]]]` or, in the runic syntax `#1+#2&`.)

Unfortunately many difficulties with details crop up.

7.6.1 Rules and Patterns

Basically, any algebraic system that tries to implement mathematics by transformations on (mostly) uninterpreted trees, is going to fall into pits. Consider the plausible rule `x_ - x_ := 0`.
This might be considered universally true, regardless of the pattern which `x` matches. Yet it is not true where each apparently identical syntactic expression can really be different. For instance, it would be an error to simplify `Infinity - Infinity` to 0. Indeed, Mathematica returns `Indeterminate`. And yet to Mathematica, the expressions `f[Infinity]-f[Infinity]`, `f[RealInterval[{0,1}]]-f[RealInterval[{0,1}]]` and `Random[Integer,n]-Random[Integer,n]` are each 0 (although the latter provokes an error message).

Similarly, two occurrences of $O(x^2)$ are not semantically identical because each may imply a different asymptotic constant. In particular, $O(x^2) - O(x^2) = O(x^2)$ would seem most plausible. In fact, the use of the equality for that expression is an unfortunate convention; a more formalistic approach would use a notation perhaps reminiscent of set inclusion (Graham *et al.* (1989)).

The problem here is fairly deep, and quite important. How could one modify the rule that `x_ - x_ :=0` to make it correct? Perhaps by saying that `x` must satisfy some

predicate? What should that predicate be? Mathematica supports examination of the `Head` of the expression tree that is denoted by `x`, and this will sometimes work. If `x` is an integer, one might agree the rule applies. But if `x` denotes (say) an expression headed by `Plus`, one must examine in principle *all* components to see if any of them are among the "dangerous" kind indicated above. Mathematica has no handle on this problem, and the kind of handle that is necessary is probably based on global information regarding the type of an expression. This problem is eliminated by systems which assign types to expressions, such as Scratchpad II (see Computer Algebra Group (1988)) or Newspeak (Foderaro (1983)).

Mathematica encourages the user to define functions by writing collections of transformation rules, each of which has a pattern, a replacement, and optionally, some conditions. This mechanism has a long history as a model of computation, and each formalism in the past has had to define the order in which rules are applied. This order has a strong bearing on the real semantics of a rule set, since different orderings can change the answers or even cause an infinite loop.

How are Mathematica's rules ordered?

The manual explains that the most explicit rules are used in preference to the most general. Yet it is clear that except for very simple cases, Mathematica has not got the slightest clue as to which rules are more specific. Maeder (1988) p. 59 points out "...Mathematica cannot always find out which of the rules is more special than the other and it might fail to reorder them accordingly." If the rules are not ordered correctly, it may be fairly painful to write patterns to make sure that the pre-conditions for matching do not overlap. Furthermore, if you "cover up" a built-in operator with rules, and then you wish to refer to the built-in routine, you have no direct access to the system's prior definition. In version 2.0, WRI apparently gave up on getting the ordering right and the user is given a chance to rearrange the rules by setting (for example) `DownValues` explicitly (see p. 266 of the 2nd edition of the manual). It is implausible that users would find this a convenient way of "correcting" a rule set.

Mathematica's patterns provide the mechanism for the "left-hand sides" of rules. The notation for 1-, 2- and 3-blank "match variables" with optional attached predicates is compact and relatively easy to read. The handling of defaults appears more general than other computer algebra systems; the handling of commutative operators is probably no messier than necessary. One can hope that the implementation is no costlier than necessary. The integration of the pattern matcher into the language extension facilities is superficially neat. The idea of so-called "up-rules" to avoid clogging common operators (especially `Plus` and `Times`) with rules, is a clever heuristic. This allows, for example, rules that pertain to the addition or multiplication of special functions to reside with the special functions, not with the common operators. This means that in principle the simplification of sums is not slowed down unnecessarily by having to look at inapplicable rules, even if they are nominally rules affecting sums. These problems have been addressed, for the most part less effectively, by numerous earlier programs. (A sample would include Cooperman (1986), Fenichel (1966), Greif (1985), Hearn (1976), Jenks (1976), Mathlab Group (1983), McIsaac (1985)). Whether the melding of rules and conditions that have side-effects can actually be put together in an efficient full-evaluation mechanism appears to be an open question. Mathematica doesn't quite match its specifications in this regard (see §8.3).

Yet even viewed as transformations of trees, there are tricky issues in pattern matching.

Consider, for example, the transformations to contract certain expressions involving factorials: Simplify $n(n-1)!$ to $n!$. A rather economical and readable version of this transformation (given that one must first understand that `n_` is a match variable that matches anything, and is referred to as `n` on the right-hand side) is

```
n_ * (n_ -1)! := n!
```
Placing this rule on `Factorial` rather than `Times` is a good idea:
```
Factorial/: n_ * (n_ -1)! := n!
```
since this would only affect the speed of simplification of products involving `Factorial`. A version that is more generally applicable, and works on $(h-2)(h-3)!$ also, uses a condition.
```
Factorial/: n_ * m_! := n! /; n==m+1
```
Indeed, there is a rule equivalent to this latter piece of code in Mathematica's combinatorial simplification library, along with some slightly more ambitious rules:

```
Factorial/:
  (n_)!/(m_)! := Product[i, {i, m+1, n}] /; n - m > 0 && IntegerQ[n-m]
Factorial/:
  (n_)!/(m_)! := 1/Product[i, {i, n+1, m}] /; m - n > 0 && IntegerQ[m-n]
Factorial/:
  (k_)! k1_ := (k1)! /; k1 - k == 1  (*equivalent to our rule *)
```

Consider now the problem of reducing $n/(n!)$ to $1/(n-1)!$. It would seem that a rule
```
Factorial /: n_/(n_!) := 1/(n-1)!
```
would do the trick. Unfortunately, one gets the error message:

```
                                                    n_
TagSetDelayed::notag: Tag Factorial not found in -----.
                                                  (n_)!
```

This means that the rule cannot be placed on `Factorial` because it is not among the top two levels of Heads in the left-hand-side. Indeed, only `Times` and `Power` qualify, and neither one of those is an attractive choice, being too common. (The expression looks like `Times[n_,Power[Factorial[n_],-1]]`). The point of this illustration is that the up-rule idea only delays by one level the potential exponential growth of rule-sets to implement complex simplifications. Delaying the problem by one level is a heuristic that may turn out to be window-dressing if programmers make heavy use of the facility. How much degradation would be caused by implementing it for deeper nesting? What would be the benefit?

Consider reducing to zero the expression

$$m!^2(m+1)^2 - (m+1)!^2.$$

The rules do not work, and one presumably has to try for a new rule involving powers, perhaps of the form
```
n_^e_ * m_!^f_ := n^f1[e,f]*m!^f2[e,f] /; pred(e,f,n,m)
```
where the details of the predicate to be applied, as well as the functions `f1` and `f2`, are, for the moment, unimportant. (Properly formulated, this rule covers one of the earlier rules.) Again what is significant here is that the "up-rule" technique won't work two levels down: this rule must either be placed on `Times`, or on `Power`, and neither option is attractive. Thus, while the Mathematica pattern matcher appears to be handy for patching the simplification rules, it is probably unwise to expect this technique to provide simple and efficient implementation of all new code. There is continuing active research into understanding the limitations and techniques for term-rewriting systems. The *ad hoc* melding of rules as supported in Mathematica, regardless of the sophistication of pattern matching, is certainly not going to guarantee important properties (including termination). The double issue of *J. Symb. Comp.* **3**, 1 and 2 (1987) discusses rewriting techniques and

applications. Yet the evidence of the past several decades casts strong doubt on the idea that an efficient version of mathematical knowledge can be imparted to a symbolic system *primarily* by rule-transformations on trees. The more fundamental approach of transforming expressions into canonical forms when possible has been shown to be quite effective. (For factorials this is probably best done by a calculus of difference and shift operators.)

7.6.2 Efficiency

There are a number of factors contributing to inefficiency in any computer algebra system. Perhaps the major factor is that most systems, including Mathematica, are interpreter based. Consequently, a simple program that can just as easily be expressed in a compiled language (e.g. C), runs orders of magnitude slower than if programmed in C. (see, for example, Buchberger (1991), where basic list structure operations are shown to execute $3000-8000$ times slower). One trick is to write such programs in C, and call them from the interpreter.

There are other factors:

- Generality. Even with a compiler (introduced in version 2.0) it is unreasonable to expect compilation, considering Mathematica's elaborate semantics, to reach the same kinds of speeds as compiled C or FORTRAN. Consequently, although it may be convenient to express a simple "do" loop in Mathematica, it will be slow in execution. The version 2.0 compiler is useful only for speeding up the numerical machine-precision real evaluation of expressions in plotting, numerical integration, or similar functions.

- Evaluation Model. Mathematica claims to implement infinite evaluation all the time, even for local variables. This means that when a variable is evaluated, a bit-vector is checked (see §8.4) and possibly the entire structure may have to be traversed even if nothing happens. Data structure operations that would normally be expected to be constant-time depend on the size of the expression. For example, `Part[stuff,1]` or `stuff[[1]]` evaluates all of `stuff` even though all that is needed is the first sub-part. This can be very expensive, as illustrated in the next section.

- Data Structures. Mathematica implements `List` objects as arrays. Consequently, extracting an item from a list based on its index has a cost that is independent of the index. Yet adding an element to the front or back of a list (`Prepend` or `Append`) depends on the length of the list, since the old array must be copied to a new location. Counter to the intuition one might have from other uses of the word `List`, adding to the front or the back appears to be equally expensive. Furthermore, picking out the ith element of a list requires that all the elements be *re-evaluated*, so that indexing is itself not constant time, but a function of the length of the list and the complexity of the elements in it. Consider, for example, a table of 10^6 zeros set up by `h=Table[0,{10^6}]`. The statement `h[[5]]=h[[5]]+1` takes 4.3 seconds on a Sun SPARC 1+, because all the elements of `h` are re-evaluated. Computing a histogram this way would be painfully slow. (see also §8.4).

  It is formally possible to concoct an abstraction of a linked list by "function calls" in a Prolog-like form. That is,

```
list[a_,b__]:=cons[a,list[b]]
list[a_]    :=cons[a,nil]
list[]      := nil
car[cons[a_,b_]]:= a
cdr[cons[a_,b_]]:= b (*etc. *)
```

but this is quite inefficient, and few of the built-in features would work on this data.

You do not have much choice about the substantial overhead caused by the generality of the mechanism for function calls, each of which constitutes a pattern match. (Mathematica's `Function` notation may perhaps be a faster choice when anonymous functions are adequate to some purpose).

- Size. Within fixed memory resources, the more space taken by system code the less is available for user programs or data. Mathematica has gotten substantially larger from its first version to version 2.0 and, if for no other reason, it may be slower as a consequence of paging or other memory-based activities.

On the other hand, there are several sources of efficiency in a high-level language.

- The built-in algorithms and appropriately chosen data-structures may provide an advantage over naive programs. For example, an arbitrarily clever test for primality, perhaps even coded in assembly language, could be included as a built-in command.
- High-level programming constructs provided to anticipate common sequences of low-level operations can eliminate repetitive interpretation. For example, a single command to map the operation of addition over a list would be faster than an iterative program indexing through the same list.
- Some expressions known to be evaluable to numbers can be compiled in a conventional fashion (especially in contexts such as plotting or numerical root-finding).
- Results can be "cached" for re-use, making it possible to reduce the complexity of some styles of heavily-recursive programming.

Mathematica uses each of these tactics. Other computer algebra programs make different cuts through the morass of decisions, and it is certainly possible to find significantly faster, as well as significantly slower programs for comparable computations. An easily accessible paper (Simon (1990)) compares four systems, including Mathematica, and may be of interest as a view of their effectiveness in solving problems on some concrete examples.

8 Major Semantic Problems

8.1 Canonicality

Wolfram (1988, p. 212-213) (1990, p. 269-270) seems to argue that, because it is theoretically impossible to have a program that reduces *all* expressions to canonical form, that the set of transformations provided (basically, `Expand`, `Factor`, `Simplify`), are sufficient.

Even within the class of expressions with decidable zero-equivalence procedures, the user is left somewhat at loose ends trying to figure out which combination of commands might be effective in mapping the difference of two possibly-equal expressions to zero, and thereby directing the result of an `If` expression the right way. The description of `Simplify` in Mathematica ("`Simplify[expr]` performs a sequence of algebraic transformations on `expr`, and returns the simplest form it finds.") fails to inspire confidence. Especially in the realm of rational functions (since in this case the answers are easily computed), the failure to use canonical simplifications can be a source of wrong answers: programs which depend on zero-equivalence being decided correctly can be misdirected. Any routine that does divisions can fall prey to this problem. While commands such as `Factor` and `Simplify` provide a nominal zero-equivalence decision-making capability, they must be called explicitly, and they do not, in general, take advantage of the possibility of the far more compact

representation and manipulation that limited domains afford. Particular problems involving complicated expressions and their negatives inside radicals are repeatedly cited as difficulties in public "bug reports" in the network newsgroup `sci.math.symbolic`. This is discussed further in section 15.1.

8.2 Dummy arguments in patterns are global

As we have discussed earlier, Mathematica combines function definitions and pattern matching with an interesting technique. A simple program definition `f[x_]:= ...` looks like a pattern-match and replacement for the expression `f[...]`. This simplifies the definition of a function over distinct structural cases (or types of arguments) by allowing more elaborate "dummy argument" specifications. Thus one can define `f` on a special case expression where `x_` would match `Sin[...]` as `f[Sin[y_]]:= ....` One can define `f` for integer arguments by `f[x_Integer]:=...`, etc.

Among algebraic manipulation systems perhaps Scratchpad I (Jenks (1984)) is a precursor, although other programming languages including ML, Prolog, and SNOBOL have similar notions. Another place the merging of the concept of matching and parameter passage appears is in the destructuring of lambda-list arguments to `defmacro` in Lisp (Steele (1984, 1990)). Unfortunately, the analogy, or the implementation of the analogy, leaves a lot to be desired. The principal objection is that a change to the dummy variable `x` on the right side of the rule `f[x_]:=(x=3)` followed by `f[y]` changes the value of the global variable `y`. A cleaner treatment would be to make `x` serve as a "bound variable" within the pattern.

Problems with evaluation may also be related to the design error in binding explained in the next section, or might be independent.

8.3 Binding of variables

Here is a simple program:

```
g[x_]:=Block[{a},a=1+x]
```

This is a program much simplified from what one might write in the course of computing an elaborate function, and it seems to be very straightforward. The result of `g[3]` is `4` and of `g[s]` is `1+s`. But why then is `g[a]` not `a+1`? It is `251+Hold[1+a]`, with a warning message about "Recursion depth exceeded". This is one result of a Mathematica "feature" which can have drastic destructive effects:[2] Any time you compose a (presumably more serious) program with local variables, you may have a conflict with a name that occurs within the actual parameters. In the Mathematica manual, section 2.5.10, Advanced Topic: Function Arguments and Local Variables, Wolfram admits, "Sometimes it can be very confusing to have the name of a local variable conflict with a value you give for a function argument. The best way to solve this problem is somehow to have the names of the local variables that appear in your functions be such that they will never appear in the values of function arguments." That is, you could try to avoid this conflict by programming, for example:

```
g[x_]:=Block[{afunnyvar},afunnyvar=1+x]
```

But this can rapidly get tiresome; potentially recursive functions lead to constructions like `Block[{v=Unique["s"]},...` in version 1.2's Laplace transform package, and even this is not necessarily effective.

[2] To quote from Maeder (1988) p. 8 "The more subtle ones [problems] become only apparent in the context of a longer session with Mathematica and are then normally very hard to find." Actually, it appears that his proposed solution is a work-around for a bug, corrected by the introduction of `Module`.

Maeder suggests avoiding the accidental "capture" of variable bindings by using the construction

```
Begin["`Private`"]
f[x_] := Block[{`a},a=x+1]
End[]
```

The user's variable `a`, or to be more verbose, `Global`a` is then never the same as the identifier `Global`Private`a` that is used inside the body of `f`. However, typing `f[`Private`a]` still causes an error, so the problem is not really solved. You can elaborate on the Context names to reduce the likelihood that names will conflict (you then have to worry about two programmers accidentally chosing the same Context name), but you cannot really eliminate the problem as is done by, for example, Pascal. By the way, if you make the mistake of trying to look at a program in some Context, on-line, you will be deluged by the repetition of Context qualifications on every name. This is quite painful.

Apparently this scoping was finally recognized as wrong, and a real solution appears in version 2.0. (also described by Maeder (1991) p. 8.) Using `Module` instead of `Block` one defines

```
g[x_]:=Module[{a},a=1+x]
```

and this works as expected: `g[a] = 1+a`. The new problem is that every time the `Module` is entered, new names are produced, presumably taking up system resources of time and perhaps space. This can be seen by trying the program

```
h[x_]:=Module[{a},Print[a]]
```

which prints a different value (looking like `a$12`) each time it is invoked. Even if they are removed eventually as `Temporary` objects, the overhead is certainly much higher than stack-allocation of variables in a classical Algol-like language.

Consider another example, also fixed in version 2.0, based on the definition of the function `h[j_] := Sum[h[k],{k,0,j-1}]`. This has problems with the local variable `k`. Mathematically speaking it is reasonable that `h[0]` returns 0, and it would seem that `h[1]` should return `Sum[h[k],{k,0,0}]` or just `h[0]`, which is 0. However Mathematica 1.2 wrote

```
General::itervar: In iterator {k,0,0-1}, variable k already has a value
```

followed by a 230-line message related to infinite recursion.

Although these *specific* examples, all reported as bugs, have been fixed, and `Module` repairs the defective `Block`, the misunderstanding persists.

Even though the role of the variable `k` in `Limit[f[k],k->b]` is very much like the role of `k` above, in this case the global value of the symbol (try `k=3`) interferes with its use as a bound variable in the limit. Furthermore, `D[Sum[a[i],{i,1,n}],a[i]]` returns `Sum[1,{i,1,n}]` while `D[Sum[a[i],{i,1,n}],a[j]]` returns `0`. The bound variable `i` is somehow capturing an instance of `i` from outside the summation.

As a sidelight, might one really wish to write a program `g[x_,y_]:=(x=y)` which when invoked as `g[r,3]` actually sets the value of `r` to 3? Attitudes toward programming by side-effects on parameters vary. Purists condemn it and point out it is not necessary. Others have grown up in a culture that permits or even encourages it. FORTRAN, and among computer algebra systems, Maple, make use of it. Common Lisp provides multiple-value returns to allow a somewhat disciplined approach to returning several values from a function, reducing the need for side-effects (or returning temporary structures of values requiring destructuring) merely to return extra results. In my view, Mathematica

should strongly discourage the use of side-effects, given that it cannot keep track of them effectively. The next section explains the problem.

8.4 Infinite evaluation

The evaluation process in Mathematica has several difficulties. Although it would be hard to discern from the documentation, the design (a) uses fallible heuristics and (b) injects some non-determinism into its results. As a consequence, Mathematica's evaluation techniques can be problematical. This section reviews the process and its consequences to the system semantics.

When presented with an expression $h[a_1, a_2, \cdots]$ the system (recursively) evaluates the head of the expression h to f, then the arguments[3] a_1 to b_1, a_2 to b_2, etc., in turn. Next, attributes and transformation rules as appropriate for application to $f[b_1, b_2, \cdots]$ are examined and additional changes may be made to the expression, perhaps through more evaluation of the right-hand sides of rules.

If anything has changed from the original expression, $h[a_1, a_2, \cdots]$ it is claimed that the system "effectively starts the evaluation sequence over again."

In fact, it often doesn't re-evaluate, because that would be quite costly. Instead the system predicts whether an additional evaluation will change anything. Sometimes the prediction fails. (I am grateful to David Jacobson, as well as public discussion by S. Wolfram for illuminating these points).

The prediction is apparently not even entirely deterministic. The only user-visible hint about this problem is in the description of the `Update` command which can be called to re-evaluate symbols "under special circumstances that rarely occur in practice." The only specific special circumstance mentioned in the manual (Wolfram (1991)) is the case of change to a global value changing the outcome of a `Condition` defining a symbol's value. The simplest example of this is `f[x_] := x /; globflag`. If the global flag `globflag` is `False` and you evaluate `f[3]`, it evaluates to itself. A subsequent change to `globflag` should cause an instance of `f[3]` to be re-evaluated. It won't.

The basic problem is that the presence of side-effects makes it virtually impossible to tell if evaluation is really complete. Consider

```
count = 0
g[x_] := x /; ++count > 100
```

In an actual infinite evaluation system, `g[3]` would evaluate to `3`. In Mathematica it evaluates to `g[3]`. Only after 100 forced re-evaluations will it return `3`.

The notion of "effective re-evaluation" is based on placing a time-stamp on each expression which evaluates to itself. "Time" in this case is simply a function that increases monotonically as computing proceeds. Note that if enough objects are produced, the system might over-run a fixed-width time-stamp field. On a MIPS M/120 processor using version 1.2, executing `Do[k=1,{12200}]` takes about 1 second. If the time-stamp is stored in an unsigned 32-bit word (as we speculate is done here), then executing `Do[k=1,{2^32}]` would over-run the time-stamp in about 4 days. As explained below, an inappropriate time-stamp might provoke an error, thus causing a long-running computation to fail mysteriously.

Assuming we do not encounter such an unlikely problem, the result of evaluating an expression E is simply E if E previously evaluated to itself and if E's time-stamp is sufficiently up-to-date.

[3] unless they are "held."

The notion of "sufficiently up-to-date" appears to be implemented approximately as follows: Every symbol in the system is stored in a hashed symbol table, using buckets for chaining. Each symbol-table bucket has a field which is updated to the current time when any symbol in the bucket is changed in any way. If the time-stamp on an expression E is at least as recent as all the symbol-table buckets of the symbols which occur in E, then that is deemed recent enough. This is clearly wrong. Even symbols *not* occurring explicitly in an expression can, by changing, cause an expression to no longer self-evaluate (e.g. `count` above).

Also, since any change to a symbol will update the time-stamp on *all* the symbol-table bucket occupants, coincidences of shared hash-buckets affect evaluation. Such coincidences cause re-evaluation of expressions (and consequent side effects) in a manner not predictable by the user, who has no way of determining which symbols share a hash-table bucket.

An aside: Why use this per-hash-bucket time-stamp? Presumably to keep the checking cost down. If there are b hash-buckets, then instead of descending through the tree of an expression to identify and check all its leaves, the evaluation mechanism can be based on a vector of b bits, associated with each expression. Then if the vector v associated with E has a "1" in position p, there is some dependence on some symbol in hash-bucket p. If the time-stamp on hash-bucket p is more recent than E, a full evaluation must be done, along with a recomputation of v.

Note that if the user or the system suspects that something "unusual" has happened that affects expressions involving `x`, then a call to `Update[x]` sets the time-stamp on the symbol-table bucket associated with `x` to "now." This will cause any expression E using `x` *or any other symbol in the same bucket* to be fully evaluated when E is next evaluated.

There are a number of tactics that can be used to make checking each of the bits' hash-buckets faster, although they may translate into (much) increased time when changing values.

Removing the infinite evaluation heuristics (especially the assumption on failed rules) may force the cost of evaluation to escalate substantially. Unfortunately this may be the cost of getting an infinite-evaluation answer right. An alternative might be to *forbid* certain kinds of "state alterations" or side-effects in the condition part of a rule—directly or indirectly, rather than merely advising against them. Note that the side effects might occur during the evaluation of the parameters, and hence in the evaluation of an entirely different function. Identifying such side-effects "automatically" would ordinarily not be computationally feasible, unfortunately. Another choice would be to use a better understood evaluation scheme that has stood the test of time in some other language. At a minimum, the documentation should clearly identify the pitfalls.

9 Numerical calculation

We start with a brief example (suggested by W. Kahan):

```
In[1]:= p=314159265358979323;
        q=314159265358979323.;
        r=314159265358979323.00000000000000000000;
        s=p+0.00000000000000000000;

In[2]:= {Tan[s],N[Tan[p]],Tan[q],Tan[r]}

Out[2]= {1.59981, 1.59981, ComplexInfinity, -1.1297926523089085443}

In[3]:= {p==q, q==r, r==s, r==p}
```

```
Out[3]= {True, True, True, True}
```

At first sight, the values of p, q and r all look the same. Adding zero to p should not alter its value much. However, Mathematica can see differences here, even though it claims all the values are equal. (Incidentally, the last value in `Out[2]` is correct.)

To continue the example, recalling that p and q are equal,

```
In[4]:= {N[Tanh[p]],Tanh[q]}

General::ovfl: Overflow occurred in computation.

Out[4]= {1., 0.}
```

Since $\tanh x \to 1$ as $x \to \infty$, the answer 1 is correct; the overflow error message suggests that a poor method is being used for computing tanh of large arguments. Although the number 0. is the result of the overflow, the answer does not indicate it. Only by computing its `Accuracy`, can you discover that "no digits are correct". (The comparable results from version 1.2 are even worse, incidentally.)

Should we care about such discrepancies? Yes! Numerical calculation in the context of a symbolic system should be done in a manner that places special importance on making decisions correctly. A system which cannot deal correctly with numerical constants will ultimately have difficulty with symbolic computation. You need not explicitly use its numerical subsystem to run into problems.

As part of a *symbolic* calculation, a subroutine may determine that an expression is a constant, but it is generally through careful *numerical* calculation that it can tell if (for example) the constant is positive. Such a determination is often quite important. Mathematica version 2.0 seems to distrust its own numerical system to such an extent that it refuses to determine the sign of $e - \pi$. (This is an improvement over version 1.2 which apparently gets the sign wrong since it simplifies $\sqrt{(e-\pi)^2}$ to $e - \pi$; in version 2.0 you must use `PowerExpand` to force this error.) What does it mean for a system to "know" all about constants, if it cannot answer simple questions about them?

9.1 Accuracy and precision

Mathematica uses the terms accuracy and precision in non-standard ways. By convention, as well as by informal usage, accuracy indicates the accomplishment of a goal of "closeness to the truth." If v is the correct value and v' is the computed value, then two measures for accuracy are absolute error: $v - v'$ and relative error: $(v - v')/v$. Precision is the exactitude of an assertion, even though the assertion might be false. For example, the statement that $\pi = 22/7$ is quite precise, although it is not very accurate. Precision is generally discussed in terms of binary or decimal digits provided in a representation of a number; it refers to the effort used to carry out detailed operations, but not whether some goal of correctness is accomplished.

Mathematica associates attributes of `Precision` and `Accuracy` with each number, unlike FORTRAN or C in which a *variable* has a precision attribute. Mathematica defines `Accuracy` as the number of decimal digits to the right of the decimal point, and `Precision` as the total number of significant decimal digits. The term "significant" is not defined, however. In Mathematica's terminology, 3.01 meters is more Accurate but less Precise than 301.0 centimeters[4]. In Mathematica, the number written as 3.0 is a representation of

[4]In this discussion we ignore the fact that Mathematica generally uses machine "hardware" floating-

any number greater than 295/100 and less than 305/100. That is, 3.0 is like an interval—a number which rounds to 3.0 when printed with two decimal digits. The numbers which you might type in as 3.0 and 3.00 are different, since they represent different intervals—although as we have seen, Mathematica does not always respect this distinction since they are equal under comparison..

Mathematica computes using arbitrary-precision arithmetic, presumably when necessary. Its decision mechanism for determining when this is necessary is not always obvious or correct. For example, what is a good approximation to sin of a large integer apparently close to a multiple of π? `Sin[3141592653589793238.]` yields 0, to `Accuracy` 0; `Sin[31415926535897932 38.00]` yields -0.45 to `Accuracy` 2; but the expression `N[Sin[3141592653589793238]]` yields -0.641653 to `Accuracy` 16. The most `Accurate` is the least accurate. In discussions with Wolfram Research employees, they defend their usage of the terms by reiterating "Accuracy is defined as the number of significant digits to the right of the decimal place." and that "Your paradoxes result from associating the everyday meaning of accurate with the well-defined concept of Accuracy." This is reminiscent of Lewis Carroll: "'When *I* use a word,' Humpty Dumpty said in rather a scornful tone, 'it means just what I choose it to mean—neither more nor less' " [*Through the Looking Glass,* 1872]

It is not difficult to compute these functions correctly: there are several packages for computing arbitrary-precision transcendental functions in the open literature (see for example Brent (1978), Wyatt *et al.* (1976), Fateman (1976), Bailey (1991)).

As another example, this one illustrating a debatable choice for the use of `Precision`, consider the function `f[h_]:=N[N[Pi,2 h]-N[Pi,h],3*h]` which one might use to look at digits of π as computed by Mathematica. The function `f` plausibly would give the difference between a $2h$-digit approximate value for π and an h-digit value, computed to $3h$ digits. Computing `f[50]` returns the value 0. (`Accuracy` 49, `Precision` 0). Clearly only about h-digit computation is being done. What one might hope would work better (in version 2.0 and later using `SetPrecision`) is

```
g[h_]:=SetPrecision[N[Pi,2 h],3 h]-SetPrecision[N[Pi,h],3 h]
```

but oddly enough, $g[50]$ returns 0, to `Accuracy` 149. The explanation for this is presumably that the value of π is apparently being "cached" to $2h$ digits, and the subsequent "lower precision" version of `Pi` actually has hidden digits that correspond to more places than deserved when it is re-precisioned to $3h$ digits. As a consequence, the value of `N[Pi,50]` is not a constant, but depends on what has been computed before.

Before discussing one alternative model, we present some remarkable puzzles—consequences of Mathematica's arithmetic—based on examples discovered by David Jacobson of Hewlett-Packard Laboratories.

Consider the iteration $s_i := 2s_{i-1} - 3{s_{i-1}}^2$. This iteration converges from a starting value of $s_0 = 0.3$ rather rapidly toward $s_n = 1/3$. Indeed, with any number x replacing 3 in that formula, the iteration proceeds to compute an approximation to $1/x$ from a suitable starting value. See what happens in Mathematica (2.0).

```
In[1]:= s[i_]:=s[i]=2 s[i-1] - 3 s[i-1]^2;

In[2]:= s[0]= SetAccuracy[3/10,18]  (* Use "bigfloats" *)
```

point arithmetic for "small" numbers such as 3.0 and 3.00; that is, all numbers 3.0...0 with fewer than 16 zeros are the same, but each is subtly different from 3.0...0 with 16 zeros. We assume that rules for calculation are those specified for the software extension to arbitrary precision in Mathematica.

```
Out[2]= 0.3

In[3]:= {s[1],s[2],s[20],s[30],s[40]}

Out[3]= {0.33, 0.3333, 0.333333, 0.3, 0.}
```

In other words, given a convergent iteration that mathematically gains accuracy at each step as it approaches 1/3, the computation begins to veer away to 0. It ends up at a stationary point at 0. Furthermore, even though they print differently, all the iterates beyond `s[4]` are pairwise `Equal` in Mathematica.

This is apparently not a bug, but a feature. WRI argues that the `Accuracy` of the results is decreased at each iteration. One way of thinking of this is as though the arithmetic were in some sense subject to experimental error. This is supposed to protect the user from believing inaccurate results.

Such a scheme might be valuable if it worked. On the other hand its utility would seem to be rather low if one could increase the `Accuracy` of a number by averaging it with itself 100 times. But this is just what happens, as seen below.

```
In[1]:= r=q=1.00000000001000000;

In[2]:= Do[q=(q+q)/2,{100}];

In[3]:= q

Out[3]= 1.000000000010000000003422632249182644607332086 74

In[4]:= {Accuracy[r],Accuracy[q]}

Out[4]= {17, 47}
```

This "bug" was fixed in version 2.0, so that now to exhibit the aberrant behavior, line 2 should be `Do[q=((q+q)+(q+q))/4,{100}];` instead. Making such errors more obscure in succeeding versions of the system will not make them go away, unfortunately. Incidentally, `((q+q)+(q+q))` is always re-written as `q+q+q+q` whether you like it or not.

In either version of the system, the value of `q` has increased in `Accuracy` by 30 decimal digits.

This general approach, sometimes called significance arithmetic, attempts to propagate "fuzz" in operations on numbers like 1.23?...? and 3.45?...? where ? indicates an unknown digit. Unfortunately, making this system secure under operations requires rules that are quite pessimistic, and generally results in too large a loss of "significant digits" to sustain a chain of numerical computation. Less pessimistic rules as implemented in Mathematica not only provide rather poor control of uncertainty, but can lead to nonsensical results. According to W. Kahan, it is a "folk theorem" that under any implementation of significance arithmetic there must exist chains of operations that lead to either an unwarranted loss of significance, or to a gain in significance, or both. The real problem is then that users will be lulled into a false sense of security. Somewhat less damaging, but certainly disconcerting, users may also be given answers much worse than they deserve. The rate of gain plus the rate of loss in bits per operation must exceed one. David Jacobson's examples above illustrate Mathematica's susceptibility to the consequences of this theorem.

Is there a way out of this? Certainly one can attempt to figure out by some independent calculation, the actual accuracy of every variable at the end of each iteration, and replace it by an equal, but more or possibly less "Accurate" value by using `SetAccuracy` and/or

`SetPrecision`. But then not only has the Mathematica design failed to perform its intended automatic error analysis, it has in fact made it advisable to perform such error analysis where previously it was not needed! Indeed, in a tutorial on Mathematica's arithmetic, Keiper (1990) illustrates the necessity, in computing the value of a Chebyshev polynomial, to repeatedly compute the appropriate precision in intermediate calculations. Keiper also explicitly sets the accuracy of the final result. Each of these resettings requires copying over the value into a newly created number, and so is not a trivial operation.

Consider for all computational purposes that each number representable in a floating-point hardware format in the computer means an exact rational number. That's it. No special hidden "precision" or "accuracy". The number 3.0 has the same *numerical value* as the integer 3.

For very long strings of digits, the hardware is inadequate, so we are inevitably forced to a software "arbitrary-precision" system to extend the model of numbers. Such a system allows a number of bits in the "fraction" or "mantissa" of the representation beyond almost any expected useful range, and generally allows a very much larger, and perhaps arbitrary-precision, exponent range. Usually such a system controls the fraction length for all calculations by some global setting, but other techniques can be used to combine numbers with different fraction lengths. Some well-known systems (Fortran) associate precision with the name of a variable. The precision of operations can be controlled by observing the precision of the destination variable, as well as explicit requested precisions along the way.

An straight-forward extension of numbers to arbitrary precision does not address *accuracy* explicitly in the number system, but (if it is to be dealt with at all) requires auxiliary information to be computed. There is no illusion that the confidence in numbers is related to their representation. The typical technique for increasing certainty *in a sequence of calculations* is to increase the number of bits in the fraction as represented, improve the input data if possible, and repeat the computation.

Arbitrary-precision floating-point arithmetic along this model is implemented in Macsyma, Maple, and Reduce. It is used in packages by Brent (1978) and Bailey (1991). One of the advantages of this system is that it can actually be used for the treatment of inexact values better than in the default system in Mathematica. For example, if x is in the interval [2.94,3.01], then x lies in the closed interval between the two exact (*not fuzzy*) numbers 294/100 and 301/100. Interval arithmetic has a substantial literature and is used in a number of computer packages (see Moore (1979)). Iterative algorithms and notions of convergence generally must be recast for interval computation. Although Mathematica version 2.0 introduces a notation for intervals, namely `RealInterval`, a critique of this aspect of the system will have to wait until the bugs are fixed and only the features remain.

On a related topic, since Mathematica allows machine floating-point numbers, it must deal with the IEEE binary floating-point standard Infinity and Not-a-Number (NaN) representations. One can obtain such an expression by computing 0/0, which Mathematica identifies as `Indeterminate`. Combining two of these quantities by addition, multiplication, etc., generally yields `Indeterminate`. So far, so good. Unfortunately, this value has Precision and Accuracy of `Infinity`, and any two `Indeterminates` test equal (`==`). Consequently, regardless of what you might wish, 0^0 is equal to 0/0. The IEEE 754 specification requires that NaN symbols compare as UNEQUAL to everything, including themselves. There is also, in Mathematica, a notion of `ComplexInfinity`, which is returned by the Limit program, even when `RealInfinity` or, (to use a notation included in the system) `DirectedInfinity[1]` would be more appropriate.

9.2 Numerics—A summary

In summary, although the immediate symptoms arise from the way Mathematica computes the `Precision` and `Accuracy` of the result of additions and multiplications, the actual problem is *its version of arithmetic turns out to be a bad idea.* Making the model more useful would require an admission of the incorrectness of the model being promoted. Wolfram Research is aware of the anomalies above (as illustrated by the somewhat ineffective changes in version 2.0) but appears to be unmotivated to change to a model that has fewer problems, such as the one above. Mathematica *fails to satisfy a reasonable set of criteria for floating-point arithmetic and dealing with uncertainty in the context of symbolic and arbitrary-precision computations.* Certainly identity operations should not ratchet up or down in precision.

In a well-designed system, exact numbers should not cause difficulties. Systems should not make misleading representations about floating-point numeric data any more than for other data. It should be possible to independently certify numbers to be of some particular accuracy or precision, regardless of the method of their computation. Presumably "machine-precision" floating-point numbers should be incorporated in a manner that will be at least as good as numerical subroutine libraries, and allow for fast computation. And special values like ∞ should be handled with a view toward preservation of correct computations.

Finally, lulling the user into a false sense of security is a far greater defect in a symbolic-numerical computing system than in a purely numerical system. A symbolic system has (at least in principle) the tools to provide correct answers.

10 Integration

Symbolic integration has always seemed of particular interest in computer algebra systems. Integration can show systems in a good light because it is now easy to compute solutions to nearly all problems encountered in a first calculus course. Yet, success with such problems does not mean the domain is "solved." On the contrary, many important problems cannot be handled either by methods taught to freshman calculus students, or by the more powerful but still only partly effective algorithms implemented by systems.

In an attempt to correct some of the flaws in earlier versions, Mathematica 1.2 includes a package for definite integrate (`DefiniteIntegr`) which, when read in, uses piecewise integration and correctly provides the answer to the problem `Integrate[1/x^2,{x,-1,1}]`, namely ∞, rather than the totally incorrect answer, -2, it provided previously. But it takes special effort to get the right answer (reading in the library), which is most unfortunate.

Also unfortunate is the very large number of known instances in which Mathematica produces incorrect results for purely symbolic (rather than numerical) integrals. One example is all that space here allows: A computation which incorrectly returns 0 in version 1.2, is the following:

$$I = \int_{-\pi}^{\pi} \frac{1 - x\cos t}{1 + x^2 - 2x\cos t}\, dt$$

Actually, the value of this integral depends on the value of the parameter x. For example, if x is set to $1/2$, Mathematica 1.2 correctly computes the answer as 2π rather than 0.

In a later version of Mathematica, the errors caused by too quickly simplifying the square-root of perfect-squares that are generated in this problem are avoided by not sim-

plifying roots. The answer is given as

$$\frac{\pi\left(1-x^2+\sqrt{(-1+x^2)^2}\right)}{\sqrt{(-1+x^2)^2}}.$$

This solution is too conservative in simplifying; the remedy is apparently to advise the user to apply `PowerExpand` to simplify and/or to commit errors. This has two benefits for the Mathematica authors: (a) The analysis necessary to see if the simplification can be done is now "avoided." For example, the system transforms $\sqrt{3^2} \rightarrow 3$, but $\sqrt{\pi^2}$ is unchanged. (b) Any errors in expanding powers are now committed explicitly by the user.

Neither of these benefits is of great use to the customer. Indeed, the most likely situation is that the user is less well equipped than the system to determine the validity of the transformations in `PowerExpand`. The choice of branches can sometimes be made from the context of the computation, but such context is not generally available in Mathematica.

Incidentally, the unsimplified integration answer above is still arguably wrong for $x = 1$. Substituting 1 for x in the answer gives `Indeterminate` as a result of computing a $1/0$ expression. Using the `Limit` program gives 0 even though the directional limits from above and below give different answers of 0 and 2π at $x = 1$; and finally, if you compute the integral with $x = 1$ from first principles the answer is neither 0 nor 2π but π.

11 Libraries

The packaging mechanism seems to be an unhappily complex one, but perhaps no worse than that available in most other languages. Common Lisp however seems to have a more effective technique for organization of modules in its Object System (CLOS, Steele (1990)). Mathematica is not unique among interactive programs intending to provide access to large libraries of scientific programs. It does, however, fail to address adequately three issues:

1. Quality of numerical routines:

   It would be preferable to have only the highest quality numerical routines. Often well-known mathematical formulas suffer from disasterous numerical instability, or are unreliable with respect to choice of complex branches.

   Because Mathematica has arbitrary-precision floating-point numbers, the design of a correct routine for some of its more esoteric functions may have to break new ground (or alternatively, be unnecessarily slow or inaccurate). On the other hand, the availability of extra-precise arithmetic may ease the implementor's task.

   Lacking any documentation, and faced with the evidence of past bug reports, it is difficult to be confident in the quality of the routines. Occasionally the correctness of their intentions is hard to assess.

   It would raise the quality of the routines if Mathematica were to adopt *en masse* the product of several decades of scientific computing analysis from one of the reputable software libraries.

2. Accessibility of symbolic or other algorithms that might be in disk library files:

   The situation in version 2.0 has improved somewhat from the past, when programs basically had to be loaded manually before the system had any information about them. Version 2.0 introduces "stubs" as a partial remedy. A stub attribute for a name causes (at parse time) the loading of an associated file defining that name. Still, the system presupposes that the user knows more about names of appropriate

functions and packages than is likely to be the case. A better solution may be difficult, but the problem remains.

3. Efficiency in retrieval of the most appropriate data:

   Mathematica seems to have barely scratched the surface in these matters. For example, the incorporation of large tables of integrals using Mathematica's pattern-matcher is problematical in terms of effectiveness and speed. Although it is plausible to add a handful of special integration rules by patterns, experiments have suggested that adding a large table would require very general patterns whose appropriateness can be eliminated only after the application of side-conditions. The look-up process in such a table is extremely costly and cannot easily be optimized. There are undoubtedly other approaches to incorporating information, but pattern-matching is the technique Mathematica promotes most extensively. The effectiveness of this "in the large" remains to be demonstrated.

There may, of course, be alternative approaches developed for library building but Mathematica's current framework for the integration of large amounts of mathematical knowledge is rather disappointing.

12 Plotting

Mathematica's adaptive plotting of curves in a plane is somewhat effective, choosing more points at areas of high curvative than other places, and not rendering exceptionally distant points. This is not in itself a novelty since, for example, Maple (version 4.1) already had such a feature. Undoubtedly any plotting program can be fooled by some functions. Defining functions by rules, logical statements or other discontinuous expressions provides enormous opportunity for generating difficult problems. Even relatively routine-looking expressions can provide difficulties. Consider the curve (taken from a problem set by W. Kahan to stymie plotting functions), $y(x) := 1+x^2+0.0125\log|1-3(x-1)|$ for $0 < x < 2$. Mathematica does no better than other programs in finding the very narrow slot, in an otherwise parabolic function, that at $x = 4/3$ dives to $-\infty$. The slot's width is on the order of 10^{-4} units. Although it is discovered in plotting from 0.0 to 2.0, it is entirely missed if the plot range is 0.1 to 2.0.

Adaptive plotting for surfaces would presumably be even more useful, but this is not provided.

The surface rendering algorithms seem to be highly effective, and the chosen default settings often work. When they do not, some study of the options is necessary; substantial flexibility is provided. There is a high level of interest in Mathematica simply for its graphics facilities, and the ease with which curves and surfaces can be specified. It is not unique in these capabilities since numerous other (purely numerical) systems such as Matlab (Mathworks (1988)) have high-quality plotting routines. Using PostScript as a device-independent intermediate form for plots is somewhat novel; other programs (including Maple and Macsyma) tend to use PostScript only for communicating with hard-copy devices.

A feature that is missing from Mathematica, but is present in Milo (Avitzur (1988)) and Theorist (Bonadio (1990)) is interactive re-plotting, where one can specify a "rubber-band" rectangular region of interest and have that section blown up (or replotted) for more detail. Re-displaying a graph in Mathematica with a different `PlotRange` can re-scale and clip out a section, but does not provide new intermediate points to justify the detail. Indeed providing such an interactive facility might be a challenge to Mathematica implementation, given the separation of front- and back-end processing in the design. The linkage, which might in the most general case require figuring out what kinds of

modifications the user has inserted in a PostScript text, or where new points must be calculated relative to mouse-positions, may be daunting, and would probably require a rather different representation. The interface in Theorist is simple, provides menu-driven or mouse-directed changes such as zooming in, changing the viewpoint on 3-d graphs, or coloration. Maple version V provides a less elaborate setting but also allows for the very useful real-time rotation of surface plots—a feature lacking in Mathematica.

13 Data types and operations

Mathematica is not set up to deal with data types except in a rather superficial way. It has been found quite useful in other computer algebra systems, and indeed in programming languages generally, to have a deeper notion of types. Mathematica supports "surface" type checks only, and does not understand their relationships. For example, although an integer is (mathematically speaking) a special case of a rational number, Mathematica does not recognize this. A pattern for `x_Rational` will not match 3. Although Mathematica has a program for testing for the superficial appearance of a polynomial `PolynomialQ`, it does not know that $(x^2-1)/(x+1)$ is a polynomial in x. It does not have a polynomial "data type" at all. Contrast this to other systems—Maple has a structured type `polynom(R,V)` which corresponds to the class of expressions in the variables `V` whose coefficients are of type `R`. That is, `polynom(integer,x)` specifies univariate polynomials in `x` whose coefficients are integers.

Scratchpad handles types in a different way, being strongly typed. A value belongs to only one type in which a clearly defined set of operations are available. A type is assigned to each expression by the system top-level interpreter if none is supplied by the user; types are an essential part of every manipulation.

There is a real dilemma presented by Mathematica's design: Consider what would happen if you were to try to incorporate a data type for compactness of representation or speed. Let us say you wished to write a much faster `PolynomialExpand` command.

On the one hand, the system could be changed to include a polynomial data type. It could be included the way `Complex` is a data type, or it could be superficially simulated by construction of a new "function." That is, one could use `Poly[x,4,5,6]` to represent $4x^2+5*x+6$ (for a discussion of such representations and speed, see Fateman (1991)).

But then the semantics of the Mathematica system require that a user be able to alter the simplification of such components as `Plus` and `Times` — how could you assure that user patterns, as well as the built-in evaluation mechanism would always match forms including `Poly` as appropriate? Since patterns are based on syntactic forms, and the new data type proposal has removed or changed those forms, the pattern mechanism in place will not work. Perhaps a whole new pattern matcher could be built to match the "abstraction" of a polynomial?

It appears that the complexity of such changes to the matcher and other parts of the system are hidden from the user, who is consequently prohibited from making such important changes.

Further subsections here elaborate on the consequences of the design.

13.1 Expressions

Mathematica makes no use of the well-defined type structure that is easily imposed on a major part of the mathematical domain of the system. The manual claims that every object is an expression which can be used anywhere in the system. This is a considerable oversimplification of what in fact is provided, but this kind of belief has probably led the implementors to neglect checking for appropriate classes of data when needed. Such checking is,

in many cases, rather difficult because of the system design. Sometimes expressions either make no sense, or have to be treated rather differently depending upon details which may be unexamined in general. For example, consider a function $f(x)$ with a discontinuity at 0. Take the limit of $f(x)$ as x approaches 0. Version 1.1 of Mathematica was clearly unprepared for the task since it proceeded to compute what was reasonable for analytic functions, but unreasonable here. That is, it computed a Taylor series. `Limit[If[x==0,1,0],x->0]` yielded `If[0[x]^4+False*x+0[x]^3+Equal^(0,0),[0,0],1,0]`. This kind of error can be fixed on a case by case basis, but the notion that the `Limit` evaluation program should be constructed as a collection of manipulations of data-representations, without mathematical semantics, was, and still is, fundamentally inadequate. The next version of the system fixed the particular example of the limit of `If`, but without addressing the general problem. As a somewhat random example, but anticipating the next section, consider `Limit[0[x]^4,x->3]`. Probably this should be undefined. Mathematica 1.1 gave 0. Version 1.2 gives `Indeterminate`. Version 2.0 leaves the expression unevaluated.

Resolving some of these questions "once and for all" is hindered by the fact that Mathematica does not have a canonical simplification policy (see §8.1, also Moses (1971)). The `Simplify` command initially worked at most for algebraic (not transcendental, exponential, or log) expressions. Even so, `-(-1+q)/(1+q)` withstood `Simplify`, and needed `ExpandNumerator` to reduce it to `(1-q)/(1+q)`. In version 2.0, `Simplify` is more powerful. Yet, how many *ad hoc* simplify-like commands are needed? In some other systems, the appropriate use of data structures would produce simplified expressions as a consequence of the abstraction and representation in use. Operations on these forms would be well defined.

13.2 Series

Although Mathematica uses a single format for expressions based on prefix trees, at least one sub-area is specialized, and uses the standard form to encode data more like a list of terms. These are expressions with the head `Series`. Although it is my view that special mathematical notions can well deserve special forms, this one was integrated into the system in a hazardous way.

Given two `Series` that agree to a certain order, their difference should be equal to the next (omitted) term. Yet in version 1.2 the expression

```
Series[Sin[x],{x,0,2}] - Series[Tan[x],{x,0,2}]
```

comes out 0, rather than `0[x]^3`. Correcting version 1.2 by putting a rule on `Subtract` produced errors in other parts of the system. `N[Series[...]]` fails. This bug was fixed in version 2.0, but its presence is indicative of the kinds of patches that will haunt the system unless there is a more formal approach to special data types. A new data type in version 2.0, `RealInterval` promises to be a major problem in this regard, since all the operations of the system must be extended to a new set of objects for which numerous built-in assumptions fail.

Even the latest version I've tried, which says correctly that `Sin[0[x]^4] = 0[x^4]` and `Sin[Pi/2+0[x]^4] = 1+0[x^4]`, erroneously believes that `Cos[0[x]^4]=1`. Certainly these are bugs to be ironed out, but is there an end in sight?

The semantics for big-O notation (see, for example, Graham *et al.* (1989)) can be dealt with formally, but such a formal treatment necessarily points out that the use of the big-O notation is inconsistent with the usual meaning of "=". Given $f(x) = O(x)$ and $g(x) = O(x)$, Mathematica correctly says that `f[x]-g[x]` is $O(x)$ but mistakenly believes that `f[x]==g[x]`.

13.3 Other Datatypes

If another type of data structure such as Poisson series—a structure used in celestial mechanics that can be thought of as similar to Fourier series—were to be introduced, how would the addition of two different series be handled? Or even the addition of two series in different variables or about different expansion points? These are touchy areas and cannot be dismissed by allowing the user to guess which of several possible mathematical conventions might hold. The informal use of rules to define combinations is quite error prone and inefficient. A further difficulty is that any operations on novel data-types left undefined may fall into the pit of Mathematica's notion of generic operations. Leaving the current set of generic operations unaltered is risky; on the other hand, modifying the generic operations requires skill and may slow the system down substantially. To be practical, it may also require access to proprietary code.

Is there a good solution? The correspondence between mathematical concepts and data abstractions, and then between abstractions and representations, has a rather substantial literature, especially among the Scratchpad implementors. Formal systems take this seriously, and there are successes to be observed. Multiple representations in Macsyma add to its power (and complexity). While each of these approaches has problems, Mathematica seems to have taken a very superficial step toward integrating the creation of new data structures into the system. It could do better.

14 Debugging

While it is not our primary objective to complain about particular hardware/software implementations of Mathematica, when a system design itself may be interfering with the ability to debug programs, it rises to the level where it should be noted in a general review.

In the past, our experience in debugging user programs with the standard kernel on several early versions of the system were quite negative. If you succeeded in interrupting the computation from the keyboard, something that was not always possible, you were thrown into a break in which you could not examine values, compute values, or do anything but print the name of the (usually internal) program that was interrupted, or the sequence of programs (without arguments), being executed. One had the choice sometimes of continuing, aborting the computation or exiting from Mathematica. These options and these pieces of information were meager.

Beginning in version 1.2 and continuing in 2.0, a redesign of debugging has changed the situation. Now the `On` tracing command ordinarily produces excessively verbose data. By using `Trace` rather than `On`, a user may filter this flood: it is possible to provide a pattern to compare with the forms so that only matching expressions will be printed. This requires a sensitive touch and some prescience about what will be relevant to see.

It is apparently possible to still find non-interruptible loops in Mathematica, in which case no debugging is really possible, and a "crash with loss of data" is about the only way to halt a computation.

Finally, the spelling correction facility can painfully slow, especially when first inadvertently invoked and especially on smaller-memory systems. Mercifully it is possible to disable it.

15 Abuse of Mathematics

While many of Mathematica's intellectual ancestors make logical hash of mathematical ambiguities or boundary values, probably no other system has documentation so bold as to assert that the system, rather than mathematical tradition, should determine the

meaning of (for example) multiple-valued inverses (Wolfram (1988) page 358) or that it is the human user of the system who has primary responsibility to check the input (and perhaps the output) of the system (Wolfram (1988) page 425: "You have to be careful, however, when the integration region contains a singularity.").

While one can be philosophical about this and (to quote W. N. Venables), "Like cars and knives and most other useful things, symbolic manipulation systems in general, and Mathematica in particular, are inherently dangerous and not for the reckless." it is certainly possible to include some safety measures. The competitors to Mathematica are not without fault; they tend, however, to be more cautious. For example, Macsyma checks to see if $n = -1$ before returning the value $x^{n+1}/(n+1)$ for $\int x^n dx$. Mathematica does not. DERIVE finesses this problem by returning $(x^{n+1} - 1)/(n+1)$ which yields the correct limit as $n \to -1$.

16 Which weaknesses are easily fixed? Which are permanent?

Certainly one can find errors of implementation, or inefficiencies, or situations in which programs in Mathematica were written to be efficient but insufficiently general. Improving Mathematica's algorithms may fix certain problems, and if past experience is any guide, occasionally will insert new ones.

In well-defined areas such as the division of arbitrary-precision integers, the factorization of polynomials over the integers, and so on, one assumes that past identified bugs in Mathematica either have been or will eventually be repaired. Thus, other than reporting and classifying known problems of this kind to some maintenance person, and perhaps devising work-arounds, the customer has little choice but to wait for the fix to arrive. This is, of course, the situation with almost any piece of software that is not available in source code; and even with source code, most people would not be well equipped to find and repair bugs.

The more difficult problems have to do with errors of design, implementation errors that are particularly widespread (in the code) and whose fixes have major impact on the speed of the system, or areas where retraction of claims in the documentation are needed.

Such areas are reviewed below.

16.1 Errors of Design

These areas have been mentioned previously.

- Scope of names: blocks, packages, rules

  In addition to problems already indicated with local programming variables in `Block` which is fixed by using `Module`, Mathematica seems to have a problem dealing with quantified mathematical variables. Most mathematicians would agree that the use of the literal x in $\lim_{x \to a} f(x)$ is locally bound, and that this expression is entirely equivalent to $\lim_{z \to a} f(z)$. The puzzle is how to specify the result of `z = f[x]` followed by `Limit[z,x->a]`. Is the `x` inside `f[x]` the same as the `x` inside the `Limit`?

  Although each of several built-in constructions using bound variables may be patched (eventually), there is still confusion inherent in Mathematica's language concerning local and global programming variables and mathematical indeterminates with the same printed name. The definition of `Hold` and related commands (now including `Evaluate, ReleaseHold,` and `Unevaluated`) suggests more than anything else that the system is still out of control.

- Infinite Evaluation

  This technique is too expensive to apply correctly, and somewhat haphazard when implemented heuristically. It appears to be one of the more negative and unforeseen consequences of relying heavily on rules as though they were procedure calls.

- Model of numerical calculation

  It is possible to perform arithmetic on approximate quantities of different precision by using the wider precision, and padding the lower precision with zeros (of whatever radix is being used in the representation). It is also possible to truncate or round the wider precision to be cruder, and perform only the crude arithmetic. As we have illustrated, Mathematica's attempt to predict numerical errors is flawed.

  Furthermore, without interval arithmetic implemented in a sensible (and subtle) way, the "problem of constants" becomes intractable instead of merely occasionally unsolvable. Such questions as the resolution of $\sqrt{x^2} \rightarrow x \cdot \mathrm{sign}(\mathrm{Real}(x))$ cannot be handled until this problem is solved together with at least linear inequalities.

- Stark data model

  The data model is that of a basically uninterpreted tree. Mathematica does not support representing and computing with an expression which is to be treated as (for example) real-valued. It has no support for "declaring" that x can assume only positive integer values, and that for example $x > 0$ or $x \cdot (x + 1)$ is an even integer. If it is told that $x > 0$ it does not know that $x < 0$ is false, nor will it object to setting x to a complex number. At least primitive versions of such representations are necessary if the system is to proceed automatically in simplifying expressions such as `Sqrt[x^2]`.

  Even though piecewise-defined functions are a basic representation scheme for introducing new functions, there is no useful way of differentiating such functions, even if they are piecewise differentiable.

  Since only surface types are handled, Mathematica often does not hesitate to respond with meaningless results when given unexpected, but not necessarily incorrect, inputs. Because features which are accidents of the implementation are mostly undocumented and are not necessarily mathematically consistent they are presumably subject to undocumented revision in the future.

  It would appear possible in principle to add piecemeal all the rules that might be needed to provide the equivalent of the Macsyma "assume" facility (Mathlab Group (1983)). A careful examination of such a programme quickly leads to the conclusion that much of the built-in functionality of the simplifier component of Mathematica would have to be papered over, and the efficiency would be very poor.

- Lack of canonical forms

  While we have earlier indicated problems with the decidability issues in Mathematica's simplifier, there is another consideration worth mentioning under the general topic of "canonical forms."

  In fact, for very significant types of calculations, especially those with more structure than polynomials, it is sometimes imperative to provide special canonical compact encodings for efficient processing. A prime example of this is Poisson series, used for computations in celestial mechanics and areas where computations with similarly large (many thousand terms) sine-cosine series are important. Of the "general purpose" computer algebra systems, Macsyma is apparently still unique in incorporating alternative encodings comparable to those used in special-purpose systems.

Other systems including Reduce and Maple, as well as Mathematica can "simulate" the form of a Poisson series using a general tree-like representation similar to that used for every other object, but this simulation is orders of magnitude slower and uses many times more storage than necessary (special purpose systems such as Schoonschip are described in Buchberger (1983)). In Mathematica there are no canonical compressed data forms comparable to Macsyma's CRE (canonical rational expression) form, or Poisson series. The only case where a somewhat compressed form is used is Mathematica's `Series` form. This is built out of the usual list structure, but is printed as a sum with a "big-O" term. This form is a mixed blessing because it is poorly integrated into the system; its problems suggest that integration of any additional special forms will also cause design difficulties.

- Reliance on the user

  The user is cautioned, "You have to be careful, however, when the integration region contains a singularity." (Wolfram (1988, 1991)). Actually, it would be helpful if the *system* were more careful, and not only for integration.

  In version 2.0, the `PowerExpand` function is provided so that the user can commit errors explicitly that in version 1.2 were committed automatically. Unfortunately the user may be quite unprepared to determine the necessary information (about signs of subexpressions in ranges) that determine the legality of the transformations.

  M. Monagan points out that for some functions, e.g. `LinearSolve`, one can pass a `ZeroTest` to the function. It doesn't stop Mathematica from returning wrong answers, but again merely "passes the bug" on to the user.

- Defective model of equality

  Often identities, invariants, or axioms don't hold in Mathematica. If `a==b` or the even stronger assertion that `a===b` is `True`, then for any deterministic, side-effect-free function `f`, once should expect that `f[a]==f[b]`. Mathematica does not match this expectation.

- Documentation

  Although it may seem that the documentation (Wolfram (1988, 1991)) is quite a superior part of the Mathematica package, there are subtle omissions. For example, the precise algorithms used for most of the routines are not described or even hinted at. The rationale given by Wolfram (March, 1991, in a lecture at the University of California at Berkeley) was that such lack of specification was a positive feature. Specificity would hinder the implementors—they would not be able to replace an old routine with a new one with different characteristics (e.g. time complexity). Wolfram asserted that it was more important to implement new features than document the old. These arguments are not especially convincing: the same arguments were offered in the context of Macsyma's reference manual twenty years ago, and the documentation is still lacking. By contrast, Maple system documentation of algorithms is extensive and on-line.

16.2 Errors of Implementation

- Non-uniformity of approach.

  Although much of Mathematica could in principle be written in its "own" language, a barrier has been placed in the implementation for reasons of efficiency and security. The user has little chance of altering the behavior of the system in fundamentally efficient ways. The only handles available are those provided by using flags already

anticipated by the developers. By contrast, even though Maple is also a proprietary system, most of the source code is available for examination, and is in a language the user can write in.

- Full-evaluation and `Update`.

  As previously noted, it appears that this is implemented with heuristics that are sometimes wrong in order to be fast. Even so, it is probably slower and certainly harder to explain than single evaluation.

- Bugs and Debugging.

  There seem be a large number of barriers to effective debugging of user-written code and rules because of the many ways rules can interact. A programmer is well-advised to make frequent use of the `Clear` command to remove the effects of partially debugged code during development.

- Complexity of code.

  It may be difficult and expensive for WRI to repair design defects that are only now being recognized. There are some 180,000 lines of C code in version 1.2 and more than 350,000 in version 2.0. If each programmer must become familiar with a substantial portion of this corpus before contributing to it, the prospect of real improvement will decline, and the prospect of introducing bugs will increase. A complex environment for developers cannot be healthy.

  By contrast, the comparable system Maple has a kernel of about 20,000 lines of C, with most of the higher-level command capability defined through programs which can be easily printed in source-code form as part of the on-line documentation.

  In the course of evaluating Mathematica, we at Berkeley have generated voluminous correspondence (nearly one megabyte—450 text pages) reporting bugs in version 1.1, 1.2 and (beta) releases of 2.0. Many bugs survive in 2.0, and additional ones have appeared. Although we have had version 2.0 (beta) only a few months, about 1/3 of the bug reports are on this (beta) system. Some bugs are, of course, simple to repair or inconsequential. Others may be quite critical to applications.

16.3 Unjustified Claims

From the academic point of view, this last type of error has great potential for damage: there are many problems for which Mathematica appears to claim complete solutions, but is not even as good as other programs (for example, in symbolic integration, solution of equations, simplification, numerical evaluation). Mathematica certainly does not "know" all of mathematics, nor is it apparent that it could form the basis for a program that approached such a lofty goal.

A user of this system might erroneously believe that if Mathematica cannot solve a problem, no other program (or human, perhaps) can do so. Users of Mathematica may prematurely abandon the technology because the first system they tried was insufficient. (This phenomenon was prevalent among early FORMAC users (see Tobey *et al.* (1965)), who typically ran out of memory on the 32K word IBM 7094 rather rapidly, and assumed that if this quite powerful system (for that time) couldn't solve their problem, it was not solvable.)

In brief, there is a risk that the audience for symbolic mathematical computation will believe that "whatever Mathematica has done, has been 'done' as best as possible" and thereby believe that any shortfall is inherent in the technology, and not the particular program.

On the other hand, the general awareness of symbolic computing has been vastly improved, and this may be more than fair compensation to academics or others interested in the field.

17 Conclusions

This review is hardly the final word on Mathematica, a program we expect to continue to change even as it has changed substantially during the writing of this commentary. In fact, we hope that some improvements in the program were prompted by earlier drafts of this paper. In the interests of brevity, almost all mentions of earlier bugs now fixed have been dropped. In a few places some discussion of features of "obsolete" versions of the system remain for the following reasons:

- In some sense, every user has an "old" version. The as-yet-unreleased version (as we write, 2.0 is not yet generally available) cannot be reviewed satisfactorily.
- Looking at the flaws of the past gives some insight into the flaws of the future.

It is possible that having raised the consciousness of the public to symbolic mathematics, the Mathematica program will then also evolve to satisfy all the various criticisms indicated here, as well as other criticisms. Alternatively, or in addition, commercial or academic "rivals" may provide new or better solutions to these problems.

For those persons waiting eagerly for mathematicians to be replaced by a universal computer program that "does mathematics", it is our opinion that this will require the development of technology that does not yet exist. Continuing research should learn from the Mathematica experience in combining symbolic mathematics in a general scientific information and programming environment with applications for research and development, teaching, and even entertainment.

18 Acknowledgments

Opinions expressed in this paper are the author's and do not necessarily represent the views of government sponsors or others mentioned below. The author wishes to thank numerous persons for enlightening discussions, comments on earlier drafts of this review, and access to preprints or technical reports. These include Paul Abbott, Bruno Buchberger, Robert Campbell, Bruce Char, Steven Christensen, Gene Cooperman, James H. Davenport, Sam Dooley, David Jacobson, G. H. Gonnet, R. W. Gosper, Dan Grayson, W. Kahan, Silvio Levy, Roman Maeder, Kevin McCurley, Kevin McIsaac, Michael Monagan, Steven Omohundro, Malcolm Slaney, Neil Soiffer, Ilan Vardi, William N. Venables and Stephen Wolfram. Also thanks to the four anonymous referees and the editors of JSC for extremely helpful comments, as well as patience through a number of intermediate revisions of sections.

A preliminary copy of this paper was provided to Wolfram Research Incorporated (WRI), and the resulting comments were used in refining this version. The author would also like to thank WRI for providing access to beta-test versions of Mathematica 1.2 and 2.0.

19 References

Arnon, D., Beach, R., McIsaac, K. and Waldspurger, C. (1988) CaminoReal: An interactive mathematical notebook, in *Document Manipulation and Typography: Proc. Intl. Conf. on Electronic Publishing, Document Manipulation, and Typography* (J.C. van Vliet, ed.), Nice, France, April 20–22, 1988, Cambridge University Press. 1–18.

Avitzur, R. (1988). *Milo* (a Macintosh computer program) Paracomp Inc. San Francisco, CA. Milo has been incorporated in FrameMaker, Frame Technology Corp. San Jose CA. 1990.

Bailey, D. H. (1991). MPFUN: A portable high performance multiprecision package. NAS Applied Research Office, NASA Ames Research Ctr. Moffet Field, CA.

Barton, D. and Fitch, J. P. (1972). A review of algebraic manipulation programs and their application. *Comput. J.* **15,** 362–381. This is an extended abstract of: Applications of Algebraic Manipulative Programs in Physics, in Rep. on Prog. in Phys.**35,** no. 3 235–314.

Bonadio, A. (1990). Allan Bonadio, *Theorist* (a Macintosh computer program), Prescience Corp. 939 Howard St. San Francisco CA. 94103. (1990, 1991).

Buchberger, B., Collins, G. E., Loos, R. (eds). (1983). *Computer Algebra: Symbolic and Algebraic Computation,* Springer-Verlag.

Buchberger, B. (1991). Gröbner bases in Mathematica: Enthusiasm and Frustration. RISC-LINZ Report 3-3, J. Kepler Univ., A-4040 Linz, Austria.

Brent, R. P. (1978). A FORTRAN multiple-precision arithmetic package, *ACM Trans. on Math. Softw.* **4** no. 1, 57–70.

Computer Algebra Group. (1988). The Scratchpad II computer algebra system interactive environment users guide, (Draft 1.1 July 19, 1988) Mathematical Sciences Dep't, IBM Research Division, T. J. Watson Res. Ctr. Yorktown Hts, NY. (382 pages) see also Jenks, R. D., Trager, B. M. (1984). A primer: 11 keys to new Scratchpad. *Proc. Eurosam 84, Lecture Notes in Computer Science* **174**, Springer-Verlag. Cooperman, G. (1986). Semantic matcher for Macsyma. *Proc. 1986 ACM Symp. on Symbolic and Algeb. Comp.*, 132–134.

Fateman, R. J. (1976). The MACSYMA Big-Floating-Point arithmetic system. *Proc. of the 1976 ACM Symp. on Symbolic and Algebraic Computation*, 209–213.

Fateman, R. J. (1989). A review of Macsyma. *IEEE Trans. on Knowledge and Data Eng.* **1,** no. 1. 133–145.

Fateman,R. J. (1991). FRPOLY: A benchmark revisited. *Lisp and Symbolic Programming,* **4** 153–162.

Fenichel, R. (1966). An on-line system for algebraic manipulation. Ph.D. dissertation, Harvard Univ., also Report MAC-TR-35, Project MAC, M.I.T., available from the Clearinghouse, document AD-657-282.

Foderaro, J. K. (1983). The design of a language for algebraic computation systems. Ph.D. dissertation, EECS Dep't., Univ. Calif., Berkeley.

Foster, G. DREAMS: Display REpresentation for Algebraic Manipulation Systems. Rpt. UCB/CSD 84/193, Computer Science Div. Univ. of Calif, Berkeley.

Foster, K. R., Bau, H. H. (1989). Symbolic Manipulation Programs for the Personal Computer. (Software review) *Science* **243,** 679–684.

Golden, J. P. (1977). The evaluation of atomic variables in Macsyma. *Proc. 1977 Macsyma Users' Conf.* Univ. of Calif, Berkeley. 109–122.

Graham, R. L., Knuth, D. E., Patashnik, O. (1989). *Concrete Mathematics*, Addison-Wesley Publ. Co.

Greif, J. M. (1985). The SMP pattern matcher. *Proc. Eurocal '85, vol. 2, Lecture Notes in Computer Science* **204,** Springer-Verlag. 303–314.

Hearn, A. C. (1984). *Reduce 3 User's Manual,* The RAND Corp. P.O. Box 2138, Santa Monica CA 90406.

Hearn, A. C. (1976). A new REDUCE model for algebraic simplification. *Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation*, 46–50.

Herman, E. A. (1988). Review of Mathematica. (also, discussion by Barwise, J., Uhl, J. Jr., Zorn, P. *Notices of the AMS* **35,** no. 9, 1334–1349.)

Hoenig, A. (1990). Mathematica, a program for various work stations and personal computers. (Review) *Math. Intell.* **12,** no. 2, 69–74.

Itturiaga, R. (1967). Contributions to mechanical mathematics. Ph.D. dissertation, Comptr. Sci. Dep't., Carnegie-Mellon Univ., Pittsburgh, Pa.

Jenks, R. D. (1976). A pattern compiler. *Proc. 1976 ACM Symp. on Symbolic and Algebraic Computation*, 60–65.

Keiper, J. (1990). Numerical Computation. Tutorial Notes. Mathematica User Conference, Redwood City, CA.

Knuth, D. E. (1969). *The Art of Computer Programming, vol 1. Fundamental Algorithms*, Addison-Wesley Publ. Co.

Korsvold, K. (1965). On-Line algebraic simplify program. Stanford A.I. Project Memo 37.

Kudera, J. R. (1988). Physics made easy. letter to the editor, *Fortune* May 25, 1988.

Maeder, R. (1988), (1991) *Programming in Mathematica.* 1st, 2nd edition. (Corresponding to Mathematica versions 1.2 and 2.0 resp.) Addison Wesley.

Mathlab Group. (1983). *Macsyma Reference Manual*, Lab. for Comp. Sci, MIT, Jan, 1983 (2 volumes: version 10), available also from the National Energy Software Center (NESC), Argonne, IL. Similar manuals are available from Symbolics, Inc., for example, version 11 (Symbolics, Inc.) Oct. 1985.

McCurley, K. S. (1988). Book review (Wolfram, Stephen (1988) Mathematica: A System for Doing Mathematics by Computer.) *ORSA J. on Computing* **2,** no. 4. 366–368.

McIsaac, K. (1985). Pattern matching algebraic identities. *SIGSAM Bull.* **19,** no. 2. 4–13.

Mathworks Inc. (1988). *Matlab* (a computer program). S. Natick, MA.

Moore, R. E. (1979). *Methods and Applications of Interval Analysis.* SIAM, Philadephia, PA.

Moses, J. (1971). Algebraic simplification, a guide for the perplexed. *Comm. ACM.* **14,** no. 8. 527–538.

Moses, J. (1974) Macsyma: the fifth year. *Proc. Eurosam 74*, Stockholm, Sweden, *ACM SIGSAM Bull.* **8,** no. 3. 105–110.

Moses, J. (1977). The variety of variables in mathematical expressions. *Proc. 1977 Macsyma Users' Conf.* Univ. of Calif, Berkeley.

Pavelle, R., Wang. P. S. (1985). Macsyma from F to G. *J. Symbolic. Comp.* **1,** no. 1, 69–100.

Pratt, V. R. (1973). Top down operator precedence. *1973 ACM Symposium on Principles of Prog. Lang.*, Boston, MA. See also, a detailed memo (1977) CGOL—An algebraic notation for MACLISP users, distributed with the CGOL source code.

Simon, B. (1990). Four computer mathematical environments. *Notices AMS* **37,** no. 7. 861–868.

The Soft Warehouse (1991). *Derive*, (a computer program). version 2.03 The Soft Warehouse, 3615 Harding Av., Honolul HI 96816.

Soiffer, N., Smith, C. J. (1986). MathScribe: A user interface for computer algebra systems. *Proc. 1986 ACM Symp. on Symbolic and Algeb. Comp.*, 16–23.

Steele, G. L. Jr. (1984, 1990). *Common LISP the Language*, Digital Press. 1st ed., 2nd ed.

Symbolic Computation Group. (1990). *Maple* (a computer program) version V. Computer Science Dep't., Univ. of Waterloo, Waterloo, Ontario, Canada. (This program is sold by Waterloo Maple Software for most computers except for the Apple Macintosh. The distributor for Macintosh is Brooks/Cole Publishing Co., Pacific Grove, CA 93950)

Taubes, G. A. (1988). Physics whiz goes into biz. *Fortune*, April 11, 1988. 90–93.

Tobey, R. G., Bobrow, R. J., Zilles, S. N. (1965). R. G. Tobey, R. J. Bobrow, and S. N. Zilles. Automatic simplification in Formac. *Proc. AFIPS 1965 Fall Joint Comput. Conf.,* 37–52.

van Hulzen, J. A., Calmet, J. (1983). Computer algebra systems. in: Buchberger *et al.*, (1983) 221–224.

Vogel, W. K. (1989). Mathematica 1.1. *Biotechnology Software.* (Mary Ann Liebert Inc. Publ. NY) July–August. 2–7.

Wolfram, Stephen. (1988, 1991). *Mathematica—A system for doing mathematics by computer*, Addison-Wesley, 1st ed., 2nd ed.

Wyatt, W. T. Jr., Lozier, D. W., and Orser, D. (1976). A portable extended precision arithmetic package. *ACM Trans. on Math. Softw.* **2,** no. 3. 209–229.