Syntax Extension as a Tool for Application Programming

Richard Fateman
Computer Science Division, EECS
University of California, Berkeley, 94720-1776

ABSTRACT

Notation is a powerful tool for leveraging our thinking. As an example, most people believe we can conceive of and express algorithms more effectively when we use a suitable computer programming language. It is clear that we lose some capabilities when inappropriately constrained by a language. It possible to accomodate different "paradigms" of thinking by choosing alternative notations or programming languages: this is often the essence of specially designed computer *application languages* or so-called "methodologies".

The largely unexploited issue we address here is the notion of having one programming language but with a *run-time mutable syntax.* It seems especially plausible to provide such a feature when the language must address an application whose own notational syntax can mutate, even over a short textual span. This is likely in mathematics and in other fields using formulas in which "customized" new scientific notations are sometimes invented.

Preface and Personal Disclaimer

I am not keen on extra notational constructs for usual *programming.* I usually write programs in Lisp, a language notorious for its near-absence of specially distinguished characters and its total lack of precedence rules. All non-atomic Lisp expressions have explicit boundaries marked by parentheses. An expression that appears in a more conventional "infix" syntax as `a+b*c+d` is written as `(+ a (* b c) d)`, and $\sin^2 x+\cos^2 x$ is `(+ (^ (sin x) 2)(^ (cos x) 2))`. To persons raised on infix programming languages such as C, Fortran, Java, this latter form may seem ugly and unfamiliar, but then the conventional expression $\sin^2 x + \cos^2 x$ is quite unacceptable as input to C, Fortran, Java or Lisp. Observe that this "natural" formula consists of a 2-dimensional variable-font utterance. What is familiar to a mathematician (or even a high school student) is ugly to a computer.

It is a conceit of computer scientists that mathematics shares a significant common notation at its core with higher-level "algebraic" programming languages. Admittedly Fortran (whose name comes from Formula Translation) is far closer to formulas than assembly language, yet its notation (and that of various "improvements" on Fortran) merely overlaps a few aspects high-school algebra, constituting a very short step into mathematics. Languages descended from Algol such as Pascal and C are not much better, and perhaps worse in some respects.

What is natural? The current Mathematica reference manual displays most of its operators in a table that has about 67 levels of precedence, with some levels containing over a dozen operations. I would be willing to guess at the precedence the expression $a\vee b\wedge c$, having seen these in logical discussions, but I would not trust my intuition to order the different precedence classes containing the symbols `CircleDot, CircleTimes, SmallCircle, Diamond` etc. This is an interesting problem: It is certainly plausible that a predefined unified notation for "all the mathematics that you or anyone else can think of" would somehow be useful, and that a sufficiently devoted technician can put it into a computer. But it is also inherently complicated and probably not intuitively clear as a single work.[1] One way we have attempted to conquer this kind of complexity is to simply not mention much of it[2].

Where are we headed? What can we say about the uses of notation in programming languages?

Syntax and Manipulating Expressions

When it comes time to manipulate expressions, additional criteria for notation must enter into the comparison. Consider the displays below with the question in mind: which you would prefer to manipulate by hand or computer? (each is an abbreviation of $(1+x+y+z)^4$):

```
z^4+(4*y+4*x+4)*z^3
    +(6*y^2+(12*x+12)*y+6*x^2+12*x+6)*z^2+
```

[1]Again, in Mathematica, `x y` means the product of x and y. The expressions `a!` and `!b` mean, respectively, the factorial of a and the logical negation of b. Thus `a! !b` would seem to be a product of $a!$ and $!b$ but is, instead `(a!)! b`. This was pointed out first in 1989 and is still in the language in 2002.

[2]As in the design of PL/I — but there it seems to have failed: the parts of the language you did not know about could still affect your program.

To be submitted ISSAC, 2003

```
.... terms omitted here
+(6*x^2+12*x+6)*y^2+(4*x^3+12*x^2+12*x+4)*y
+x^4+4*x^3+6*x^2+4*x+1
```

or the Lisp version, which may take a minute or two to understand.

```
(+
 (^ z 4)
 (* (^ z 3)
    (+ (* 4 y)
       (* 4 x)
       4))
 (* (^ z 2)
 ....  many terms omitted here
))
```

Note that a good editing program routinely would allow you to mark, copy, or move any balanced parenthesis group, to any depth, in either representation, but Lisp uniformly groups all terms down to the lowest level: `(* 4 x)`.

Life without syntax: Lisp is nearly syntax free

The typical Lisp program consists of defining new functions to be invoked with the same delimited operations using the same parenthetical delimiters (). Thus syntax is not an issue for the general Lisp programmer. There are other paradigms for display of large expressions. A spreadsheet arrangement is also possible, and suitably enhanced for computer algebra, this presents a number of advantages over the typeset version. The pleasant prospect of the lisp data above is that the machine or human parsing of "*" is no different from the parsing of "+" or the parsing of "$\hat{}$". In fact, "parsing" the printed (lisp-style) form of that expression into lisp is done by executing the program `(read)`. If the data is in a file `data`, a conventional way of parsing would be `(with-open-file (stream "data")(read stream))`.

Inventing Syntax

Lisp's regularity provides a solution to several problems for which many subsequent inferior proposals exist. We are not arguing just about certain programming languages whose syntax is so complex that many students are overwhelmed to the point of not ever really understanding the semantics. We also note that Lisp puts markup languages (html, sgml, xml, mathml) to shame. The reasons for them are reminiscent of the parental rhetorical question, "If your friend Johnny jumped off a cliff, would you jump too?" Apparently the answer for markup languages is "`<response>yes</response>`."

How can we help those misguided(?) persons who, for reasons of history or mysterious preference regard such additional syntax as useful?

The Lisp view is simple: the prospect for elaboration of notation *by writing parsers or interpreters in Lisp* opens up wide-ranging prospects, and indeed Lisp programmers can easily define new languages. The notion of a read-eval-print loop (REPL) essential to Lisp and other interactive languages can be easily changed to define different language syntax by replacing the components. The Lisp reader can be replaced by a parser for another language, and the eval/print parts can be changed to correspond.

Anticipating interest in changing the input language, the Common Lisp ANSI standard specifies a programmable "readtable" which is a built-in lexical analyzer. This model of REPL computing is, in large part, the point of several weeks of our introductory programming course entitled "Structure and Interpretation of Computer Programs."

The REPL is also one of the fundamental building blocks of any other interactive computer systems, including various UNIX command shells, languages like Basic, APL, Matlab.

Computer Algebra Systems

If we pursue our notation interest, we would prefer to aim our remarks at more significant interactive systems, namely those that are aimed at a more sophisticated mathematical audience. These systems are principally computer algebra systems (CAS) like Maple, Mathematica, Macsyma, Reduce, MuPad, Axiom, Derive, and some others, where the command syntax is some approximation of infix "conventional mathematics" through which a user is supposed to utter formulas and commands. The output can be nearly anything from a plot to a table or some approximation of two-dimensional typesetting comparable to (or even including) Knuth's TEX system. In between the input and the output the mechanics of a computer algebra system consist of some melding of interactive programming with mathematical operations. The often-cited examples of operations include symbolic differentiation, integration, and simplification.

Returning to our theme of syntax extension, let us turn to the two, we think separable, questions at issue here:

- Should programmers generally be allowed to make up their own syntax?
- Should CAS users (including users of systems which incidentally allow programming) make up their own syntax?

So as to not keep you in suspense, here are our answers:

- Most programmers should be discouraged, but (programmers being independent souls) may do so anyway.
- People building serious CAS application packages should be aware of this prospect and use it to present a more application-oriented view of a system to their users. As always it helps to design such a system through understanding constraints of technology, good user-interface design, and common-sense tenets of good notation.

In the rest of this paper we will discuss the latter issue, since notation for symbolic mathematics and languages to manipulate mathematics are our primary application. Secondarily, we consider the role of mathematical notation in documents (scientific word processing) and digital information.

Hasn't syntax extension already been done by C++?

The C++ language (c. 1985) has operator overloading allowing programmers to define classes that re-use familiar operators to apply different semantics. Overloading can be done for most of the existing operators, but requires that at least one operand be a user-defined type. Used judiciously it allows the definer of a class to provide a more intuitive notational hook into a class.

How far does this get us?

One traditional flaw in C of not providing a "power" operator for A^B means that you cannot make one for any of your defined classes. Unless of course you wish to call your operation * or + and use the same precedence as those operators. This does not seem like a good idea.

And you cannot change the meaning of any built-in operator when its arguments are built-in types.

Liberal use of overloading is hazardous since examining a program section taken out of context is can be misinterpreted: the operations could mean almost anything and the syntax would not give you much of a hint. Lisp too, through its Object System (CLOS) allows for generic functions, and so the potential problem exists there too, but you are not likely to be misled by the *syntax* in Lisp. It all looks about the same. You could be misled by the names, if a programmer were disposed to redefine PLUS and TIMES[3]

Among computer algebra systems, MuPad offers both types and operator overloading. The late lamented Axiom system had an elaborate system with similar but more ambitious goals. At runtime, Mathematica, through pattern matching, offers a kind of overloading, whereas programmers for other computer algebra systems such as Macsyma and Maple generally simulate overloading by (in effect) inserting a case-statement for run-time checking of arguments' types or other attributes.

How did we get so far off track?

From a mathematics notation design perspective, any developer of "new" syntax should consider the positive effects of a design to resemble the common "non-computer" practice *in the domain of the application.* The negative effects may be substantial in a poor design, making programming by "programmers" more difficult. Sometimes the computer language design differs from the non-computer notation in subtle and harmful ways. *de gustibus non est disputandum.*

The computer has no taste and it is just as possible to clarify via notation as it is to obfuscate. Imprudent use of TEX may clothe a document in obscurity.

[3]Clearly any programming language supports the possibility of writing confusing programs. It is considered bad taste to overload operators without some clear documentation and underlying plausible analogies governing the multiple definitions of symbols. Extending the definition of printing routines to print new objects is a plausible addition. Extending the definition of printing routines to do matrix inversion is implausible.

Sometimes new syntax makes programs exceedingly compact and (perhaps through analogies with other notations from mathematics, physics, etc) quite perspicuous. In such cases the argument can be made that some pain in absorbing new syntax is worth the gain.

A further rather strong argument in favor of our considering syntax extension in mathematics is that it is *inevitable*: the contrary view requires that all syntax has already been invented. This can be contradicted on request by any mathematician[4].

The *most obvious* difference between computer languages and mathematics is that symbols in most mathematical formulas are single glyphs from a large alphabet including Greek, special symbols, and diacritical marks like primes, hats, overbars, and various vertical arrangements[5] This may seem like a silly point, but in fact mathematicians using such discrete symbols then have the ability to use the empty space or just the appearance of adjacency between those symbols in many different ways. Contrary to computer science (CS) common wisdom, `3.0d0*ALPHA**4*SIN(X)**2` is not normal mathematics. Compare it to $3\alpha^4 \sin^2 x$ is closer to conventional appearance.

The *major* difference is not the better symbol set. A more significant difference, and one reason that programming language parsers have such problems with "real" mathematics is the ambiguity that is often inherent in interpreting the absence of symbols! Consider the spaces in that last formula[6]. Two of the spaces mean multiplication, one means function application. Also the two superscripts must be interpreted differently. The form $s^2(x + y)$ could mean `(s(x+y))^2` or `s^2*(x+y)` or perhaps even `s(s(x+y))` and you don't know which until we tell you if s is a function or a constant, and what our convention might be for "powers" of a function.

This gap between CS conventional notation and mathematics is traditionally bridged by (in effect) telling the human user: if you want to use the computer, then you have to forego the freedom and familiarity of your own notation and learn some programming language. While this seems like a perfectly fine solution from the CS side, traditional mathematicians may acquiese or rebel. But must the CS side be so inflexible?

Arguments in favor of sticking to the programming language and forcing users to learn it.

> The scientist who wishes to use a computer for numerical experiments must learn a computer language like C or Fortran, or (in rare cases where lower-level acess is required for speed) assem-

[4]a visitor to Tilu seemed to think `v sin2(x)+k` was appropriate to express $\sqrt{\sin^2 x + k}$. How can you battle that?
[5]Fortunately we are no longer restricted to encoding such notations into all-capitals Fortran names limited to 6 or 8 characters.
[6]Some of these issues are addressed in attempts to parse images of typeset mathematics. This domain of inquiry is part of a field named Document Image Analysis (DIA). There is an overlap with formal language theory, but not as much as you might think.

bler. Using the rigid syntax of the programming language imposes constraints and prevents ambiguity. It might even promote a good style of thought. Only by learning the CS discipline can the scientist appreciate the good and bad aspects of the programming language implementation and perhaps even computer architectures, and bring the full power of the computer to bear on the computation. The higher-level languages are not only unambiguous and carefully documented, programs written in them may be subject to optimizations and transformations, tracking the advances in computers(parallel, distributed, pipelined, etc.) allowing the programs to run faster.

Arguments in favor of providing a language as close to the format understood by the application scientist.

The most important part of the problem statement is that it be correct. To facilitate this the scientist should state the problem to be solved in a declarative form as close to the mathematical model as possible. Systems such as $\TeX$ are popular, but not easy to use (or even unambiguous) as a language. Good translation mechanisms can and should be built directly from the problem statement to the level of execution, taking full advantage of the higher-level information. It is easier to optimize an utterance like "integrate(f(x),x,a,b)" by finding an appropriate integration routine than it is to try to deduce the programmer's intent in some naive attempt to implement Simpson's rule, and produce an "optimal" result, whatever that might actually be.

The programs and the documentation also look better if the notations are concise and conventional in science terms. Expert mathematicians (see for example the draft of the Digital Library of Mathematical Functions citedlmf) do not want to be told to use different and clearly inferior notations. After all the computer should be the servant, not the master.

For any computational purpose it should be plausible to have a computer map conventional notations in any given domain into the necessary semantics.

Some proposed solutions

0.1 Fortran, C, bad; New Language, good?

We could propose that the existing traditional languages are inadequate for the task, but a new language we have invented will make it possible to bridge the gap. See for example MAS, ALDES, SAC-2, Aldor, Newspeak, Weyl, Common Lisp, Java, C++, Modula-2, ML, OCAML, or any of the computer algebra systems with their fixed syntax.

Yes, we've seen them. They are not natural mathematics. They attempt to take applications programmers and (in some cases) force them to learn Other Mathematics (abstract algebra, category theory), and then Some Unfamiliar Notation. Sadly, these techniques often appear to be a poor match to the original task. Perhaps because the language was designed first, and was not influenced by the contemporary notation of the application.

0.2 Syntax Extension

Can we retroactively fix a language? What has been done in the past? How has syntax extension been incorporated in computer algebra systems? Probably the oldest syntax extension system is in Macsyma (circa 1970). The syntax extension is implemented even in the free version (circa 1982, now updated on sourceforge under the name Maxima.) The syntax extension features of Mathematica are more recent (1996?)

Examples from Macsyma

The vertical bar is sometimes used as a marker to evaluate. For example, $\sin x|_{x=a}$ means evaluate $\sin(a)$. Macsyma allows this

```
prefix(sin);
infix("|",40,40);
(r | s) := ev(r,s);
sin x | x=a;  /*evaluates to sin(a),
                displayed as sin a */
```

The mysterious number 40 was chosen to specify left and right binding power for the operator "`|`". This is legal Macsyma[7].

If we wish to state that `A` and `B` are the names for lines that are parallel, we could use some programming construct like `IsParallel(A,B)`, or if we were using Lisp, `(IsParallel A B)`. But why not

```
infix("||")
a||b;
```

and similarly, for perpendicular:

```
infix("_|_");
```

Thus one can convey a condition as `A||B and C _|_ D`

There are other notational extensions in Macsyma, for example so-called "matchfix" operations. As an example, we could choose `{a,b}` to be a simpler form for `a.b-b.a`. Ordinarily this notation would not be expanded explicitly, but left implicit, subject to simplification rules such as observing that for any x, the expression `{x,x}` should be replaced by zero.

[7]It may be that the battle for the simplest user interface is one that pits such knowledge of precedence versus the use of patterns or templates in Mathematica. It appears that the order in which templates are introduced is effectively setting precedence and may not really provide a solution.

```
matchfix("{","}");
matchdeclare(r,true);
tellsimp({r,r},0);
```

Composition of such simplification rules, using the natural notation of the physics problem, is an essential part of symbolic computation. The flexibility of a notation can make it far more pleasant, and the conciseness may make some calculations feasible where expansion into some other forms would lead to exponential space consumption.

As another example, one which has occurred in my own notes for computer algebra, consider difference and shift operators. Normally $\Delta f(x)$ is a shorthand for $f(x+1) - f(x)$, and the shift operator E can be defined as $Ef(x) = f(x+1)$. Without much effort we can declare Δ to be a prefix operator with precedence (binding power) between that of "+" and "*", Then the rule for difference of a quotient could be expressed as

$\Delta f(x)/g(x) = -(f(x)*\Delta g(x)+g(x)*\Delta f(x))/(g(x)*Eg(x))$

or perhaps, leaving off the x dependency,

$$\Delta f/g = -(f * \Delta g + g * \Delta f)/(g * Eg).$$

The display program will reflect this statement by typesetting this:

$$\Delta \frac{f}{g} = -\frac{f\,\Delta g + (\Delta f)\ g}{g\,Eg}.$$

By comparison if we use a strictly functional form, the corresponding expressions would look like this:

$\Delta(f(x)/g(x)) = -(f(x)*\Delta(g(x))+g(x)*\Delta(f(x)))/(g(x)*E(g(x)))$

or

$$\Delta(f/g) = -(f * \Delta(g) + g * \Delta(f))/(g * E(g))$$

or, in displayed format,

$$\Delta\left(\frac{f}{g}\right) = -\frac{f\,\Delta\,(g) + \Delta\,(f)\ g}{g\,E\,(g)}$$

Compare the size and complexity of the display on these small examples. Convention would have the smaller expression clearer, assuming you understood that $g\,Eg$ meant the product of g and Eg, where this latter expression is g shifted by one. The differences may not be enormous, but they get far more substantial on larger expressions.

Examples from Mathematica

Mathematica, starting in version 3.0, included a Notation extension package intended to provide a user-friendly interface to augmenting both input and display. The user-programmable display was an feature in the earliest Mathematica versions, but hardly user-friendly. Jason Harris's work [8] pushes at the border of what can be displayed. His work allows him to declare an equivalence between two expressions, the first being a form typed in a template-directed display with interspersed pattern-variables, and the second being a conventional Mathematica expression for internal consumption. A simpler version allows simple infix expressions to be defined as functions, such as
`a_ newsymbol b_ :=newfun[a,b]`
where the newsymbol is typically an otherwise illegal symbol.

The designers of Mathematica have clearly made a substantial effort to apply newer computer technology to this notation problem. This includes taking advantage of huge and varied character sets and the graphical display technology to produce them at various sizes on interactive displays and printers.

The impressive demonstrations included with the current release (I'm looking at version 4.1) include notations for (square) commutative diagrams, and physics notations (brakets). This goes far beyond the essentially linear Macsyma notation.

The evidence to date is that it is not used. A question sent in to the Mathematica users newsgroup in May, 2002 elicited only two responses, one saying this package was not worth the effort, and the other from a Mathematica expert who nevertheless admitted that [Harris] "can get his package to do things that I cannot work out how to implement.".

Is this then the death knell of a solution?

Is there another solution that is more appealing? Is there an alternative?

Developing new Mathematica notations (or entering them once they are defined) requires that the user develop skills either with palettes/menus or special escape notations. It also requires that anyone using the notation follow substantially the same route or sequence of operations to set up the notation. That is, one cannot take a Mathematica notebook and cut/paste sections into a fresh notebook that is ignorant of the introduced notations. (or far worse, has conflicting rules). It may seem unreasonable to ask Mathematica to solve this kind of problem which is pervasive in other contexts. Even without notational changes, you would not ordinarily clip a piece of text from the source code of one program and expect it to work in the body of another program without regard to use of names, declarations, etc. Pasting text into a WYSIWYG document seems to assume a mixture of attributes of the origin and destination site contexts. We address this issue in our recommendations.

Notations generally

Gottfried Wilhem von Leibniz was not alone in thinking that having the right notation was somehow critical to thought[9]. Computer science readers may be more familiar with Kenneth Iverson's thoughts on notation and programming [4].

[8]http://library.wolfram.com/conferences/devconf2001/harris/harris.html

[9]See the text and illustrations from an interesting illustrated talk on notation by Stephen Wolfram [9]. The on-line text provides illustrations of pages from Leibniz and Newton, a page of notation in *Principia Mathematica* and other historical documents.

Certainly researchers in many fields rely on the work of Donald Knuth and his collaborators on TEX [5] in applying the computer to problems of typesetting and printing.

Current and Future Practice

Has syntax extension caught on? Not very much: we speculate that the reason for this is a combination of ignorance (who knew it could be done!) and caution: it takes a steady hand to define collections of notations with more than a few definitions. The scope of the newly defined syntax is indefinite (that is, it lasts until it is explicitly removed) and so there is the hazard of two modules redefining the same syntax. This is not a new problem in principle: There is already a problem with redefining functions with the same name, or re-using a global variable. It is just not as familiar because so few languages allow for syntax extension in this manner.

How is such a parser constructed?

Where did this come from, anyway? The parsers for Macsyma, Reduce, and Scratchpad were based on the same technology, a top-down precedence parser described in detail by Vaughn Pratt [8].

The implementation of this parser is available in various Lisp source repositories.

Since it is actually independent of computer algebra systems, why has it not been used more widely? The abstract power of the parser is substantial; it may be that popularity of tools like Lex and Yacc have dominated, or that CGOL carries the stigma of being written in Lisp. We have no benchmarks to suggest this parsing technique is slower; the size of the parser, assuming one has a Lisp system present, is probably not much of an issue. The error detection and correction possibilities are probably more difficult to quantify than static parsers; at least we know of no serious attempts to study this aspect of Pratt parsers.

The Mathematica parser is essentially undescribed in the literature. The notation extension consists of an *ad hoc* pattern match finding replacements in a left-to-right scan of the input, where the patterns are ordered last-in first-used. This does not correspond to any of the well-studied models of programming syntax.

What about handwritten input?

Each year there are announcements of newly developed demonstration programs that are recognizers for handwritten mathematics. These programs are indeed getting more capable generally, along with new input devices, glyph-recognition programs, and (in a few cases) a more principled approach to the problem of understanding and parsing 2-D expressions. (e.g. D. Blostein [2]).

We suspect that most current application users of a computer system would rebel at the inadequate accuracy of handwritten input, and the limited grasp of notation, as currently programmed. (Basically super/sub-scripts, divide bars, integral signs and sometimes square-roots.) The rebellion would be based on having to repeatedly re-write or re-enter symbols and expressions to get a satisfactory recognition, or alternatively to learn a new language for editing 2-D mathematics to correct the mistakes. Note that in 2-D, ambiguities of interpreting blank space are amplified. Even linear notation $s(v + w)$ does not tell us if this is a multiplication or an application of a function named s.

A future audience more familiar with menus and stylus input[10] may change our views, but given the prevalence of keyboards today and the relative efficiency and accuracy of keyboarding (versus recognizing handwriting) we expect there will still be a substantial role to be played by linear character input forms and typewritten formulas, both for programming and user input to a computer algebra system.

What about TEX as input?

We do not discard entirely the possibility of computers "recognizing" typeset or handwritten formulas. In fact, we have participated in projects providing substantially correct conversion of some PostScript into TEX and from TEX into expressions suitable for use by computer algebra systems. One transformation tool, written with student Elon Caspi, designed to convert from TEX to "Macsyma" was run over 10,740 formulas from the first digital edition of a major table of integrals (Gradshteyn and Rhyzik), and "understood" about 4834 of them. This required a special context dictionary specific to the reference. Adding more context would translate more formulas, but in reality, some of the remainder essentially require one-off translation. They include various *ad hoc* constructions using natural language, or require some novel inference to treat ellipsis ("...") in a sequence.

Our experience suggests that the recognition of advanced mathematics, once typeset in TEX requires a deft touch to translate into unambiguous mathematics, and in particular may requires a fair amount of contextual knowledge.

Stephen Wolfram claims [9] that Mathematica 3.0 can solve the problem of converting Mathematica's `TraditionalForm` display —tantamount to a WYSIWYG math typesetter or TEX —to Mathematica's unambiguous `StandardForm` by "... a few hundred rules that are heuristics for understanding traditional form expressions. And they work fairly well. Sufficiently well, in fact, that one can really go through large volumes of legacy math notation–say specified in TEX –and expect to convert it automatically to unambiguously meaningful Mathematica input."

A minute of experimentation (with Mathematica 4.0) suggests that Wolfram's claim either is overly optimistic, or vastly underestimating the range of possible "TraditionalForm" mathematics. Large quantities of very routine formulas, perhaps those converted by Mathematica from "StandardForm" does not constitute a proof of versatility. In particular the kinds of problems that required special solutions in our integral table are unsolved by Mathematica. Consider $1 + x + \cdots + x^n$, an expression which is transformed into a rather different "StandardForm" of $1 + x + x^n + \cdots$ To my mind, a solution *recognizable as such only to Mathematica fans*, might be `Plus@@Table[x^i,{i,0,n}]`. This is, however, an error as long as `n` is not an explicit number; better is `Hold[ Plus@@Table[ x^i, {i,0,n}]]`, yet this is

[10]The Tablet PC is currently (2002) being touted as the next big thing.

an expression which is almost useless. Another example for Mathematica fans is to try `Dot[1,2]-1.2` which typesets as $1.2 - 1.2$ in version 4.0. Perhaps one needs a few (hundred?) more heuristics to improve Mathematica's operating rules. There is certainly evidence that no single organizing principle is going to make traditional mathematics consistent, and plenty of evidence that human users of mathematics, expert and non-expert, are not consistent[11]. Steve Swanson[12] who has substantial experience in Scientific WorkPlace, suggests that a steady accumulation of the important cases, combined with a way to organize the relevant data/rules will lead to progress.

Another direction through the notation issue is taken by programs that are primarily document-oriented. These can include web-browsers, document search engines using full-text search or modified (marked up) document indexing, and the programs that produce such material. A prime example of a scientific word-processing programs is Scientific WorkPlace (`http://www.mackichan.com`). In some sense this is the flip side of the computer algebra system "notebook" interface in Mathematica, Maple, MuPad, Macsyma, where text/commentary sections are secondary to the computation/display sections. With Scientific WorkPlace, the presumed major interest of the user is the text: composition and communication of explanatory material with mathematical content. The target could be a laboratory report, journal article, examination, or a textbook. The important difference with traditional material (printed, TEX or other word-processor based) is that there is strong support for embedded formulas, plots, diagrams and associated computation. It is also inherently interactive, whereas some of the other efforts to join such pieces (say, using WWW links) have had tenuous links.

In Scientific WorkPlace the mapping from the user's typeset notions to the underlying computation engine has the appearance of a WYSIWYG editor. This difficult task is managed by *ad hoc rules*; in the words of developer[13] "We've tried very hard to find the expected meaning of common expressions. It's also true that asking the user is hopeless. Most users don't understand what the problem is."

It is difficult to broach the subject of techiques for syntax extension when the user may not have any experience in programming, and more than likely has no perception of the ambiguity of mathematical utterances.

In summary: the Scientific WorkPlace perspective changes the focus from programming languages to document object models. It must then address a complex situation where there are competing commercial technologies with orthogonal concerns such as distribution, data bases, persistence, security. We step away from this for now[14], but suggest changes to the notebook model in the section on Recommendations.

[11]We collect about 150-200 free-form input formulas daily from an on-line symbolic integration server, Tilu. A few years ago, with student Rick Warfield, we [10] wrote and installed a parser for an ambiguous grammar with post-parsing heuristics for disambiguating notation, to help deal with the many (40%) technically ill-formed but still humanly recognizable inputs.

[12]email 8/22/02

[13]Steve Swanson of Mackichan, August, 2002

What about a universal grammar?

Even a primitive notation is going to be "Turing equivalent" to any other notation so formally all we really need is lambda calculus, a Turing machine, a string-rewriting system, or a binary digital computer. This is a pointless perspective since humans are uncomfortable with the many layers of representation needed for such an encoding to handle non-trivial mathematics. What can we say about higher-level functionality of notation?

In the previously cited talk [9], Wolfram suggests that he has hit upon the universal grammar for all of mathematics; it is his own invention, Mathematica. It is easy to set this aside as sheer marketing fluff; but what principled arguments are there against Wolfram's claim? Since there is a mapping from Mathematica to (and from) Lisp, I will at the same time argue that Lisp doesn't hack it either!

Even very careful authors abuse notation. One of the features that distinguish most of us from careful authors is that they *announce their abuses*, sometimes in a table of notations, but sometimes in plain text just before the abuse. For a computer system to automatically and correctly encode such notations it must have sufficient power to use the intermediate text to add context to the notation which follows. Arguably then the computer system must be "intelligent" enough to read the journal article and understand it before it can fully decode the notation of the article. Most readers of this paper probably recall experiencing distress from having to read an article several times before understanding it. We doubt that notation reformists (even those who are as expert at sales as Wolfram) will be able to make such articles illegal.

Actually our hope lies not in law, but in less formal constraints placed on authors hoping to be published.

Computers programs (CAS initially) can be successful in decoding notation human has encoded the context already, most obviously if the human already presents a Mathematica, Maple, (etc) program, or as some seem to believe, into MathML[15]. A natural perspective is that the *author* or *editorial staff* should provide this translation rather than (as is the current practice) *each reader*.

[14]However, during the summer of 2002 we experimented with OLE, using it to link an Excel spreadsheet to Common Lisp, with the intention of making a computer-algebra facility, including typesetting, available

[15]We have reservations about MathML: if the translation is only into *presentation MathML*, the most likely scenario, then nothing is added beyond typesetting commands. A translation into the somewhat nebulous *content* MathML could be semantically useful; the only usage in this area seem to be from computer algebra systems which produce MathML as a presentation and then a brief attachment that indicates the equivalent CAS string in their own language. This presents no more or less information than the CAS string, since CAS can routinely produce presentation MathML from their content. So the only way MathML makes sense is if it includes one of the other choices as well.

The advantage should seem clear to anyone with a modern constructive outlook on communicating mathematics: the inevitability of an extensible open-ended mathematics language (with MathML we would have style-sheets, with OpenMath we would have content dictionaries, and with a computer algebra systems we would have a subroutine library and possible data-type extensions). The alternative is to condemn the future of mathematical discourse to a continuation of the current mode of communication, which in some cases degrades to essentially waving our hands at each other and hoping our audience either understands, or in the typical teacher-student relationship, the student is cowed into not raising objections.

Taste

As in many programming language issues, taste is important. It is possible to paste together an unworkable mess; however with the recently increased popularity of unicode (now supported in some Common Lisps, as well as the computer algebra system Mathematica), there is a chance to overcome the particular fear of rapidly running out of the few mnemonic ASCII non-alphabetic symbols favored as operators (favorites are `@` and `%` which are not used by Macsyma and ordinarily left unbound). The parser can be extended with symbols of any length, and one could even make binary operators of symbols like "x" and "o" or "//" if one is willing to forego their use as variables.

Other notations

There are obvious additional examples within computing, logic, and design including graphical models (circuit diagram, timing diagrams, Venn diagrams, directed graphs, pictures of trees, matrices, difference tables). Going further afield we would certainly have to consider the rendering of music, mechanical drawings, street maps and travel timetables. We are ourselves experimenting with a graph editor to be attached to a computer algebra system[16].

A web search can find many notations: Here's a quantum physics summary of notations: `http:// www.adi.uam.es/ Docs/Knowledge/ Fundamental_Theory/ quantrev/node19.html`

XML, MathML, OpenMath

We are faced with the prospect that systems with fixed syntax will be imposed upon us by recent work in making mathematics web-aware. The likelihood of your work being representable then becomes, in part, a political matter: can you convince some committee that your notation is the proper one, and that therefore it should be supported on the browser display systems used for web math. An extensible syntax might go a long way toward making it possible to utter arbitrary higher mathematics. The simultaneous adoption of the typeset output can be accomodated as well. The MINSE/GlyphD system presents a kind of work-around: as a model it might be useful to revisit it, or for those who have never seen it, to visit it for the first time: MINSE is an established typesetting agent on the internet. If you can convince it to allow a notation and corresponding typeset appearance, then anyone else on the internet can get the same picture. (To be quite universal, MINSE uses GIF now, but PNG, a non-patented graphics format is possible).

[16]Finding a congenial symbolic notation or file interchange format eventually comes up. It appears that GML (graph modeling language) has gained some popularity as a file format.

Conclusions and Recommendations

The CAS community is in constant danger of damaging itself by syntax. The dominant *student view* of "symbolic computer calculus labs" is that the CAS is another burdensome topic to learn. This is in stark contrast to the *teachers' view* that the computer lab is a wonderful "enriching experience." If the lab didn't require learning a new and mysterious syntax, perhaps the student experience would be more aligned to the intent. Frankly, how enriching is learning about semicolons, quotes, percent marks, distinguishing between functions and expressions, saving files, etc?

We believe that syntax extension at run-time is a viable, although currently unpopular technique for easing the transition from conventional mathematical notation to a programming notation. It may transfer the burden of learning notation to the system builder who will model existing notation. It is also possible to use this technology to develop novel notations not based on prior conventions.

Few systems present the user of a system with syntax extension even though technology to provide it has been implemented and available for years in the form of CGOL.

A less flexible version of syntax extension which requires a revision of a grammar and recompilation of a parser at each change is perhaps more comfortable for many computer scientists. Those who have taken (or taught!) a compiler course in which tools like YACC or BISON or CUP are used, are thus familiar with a model in which a programming language syntax is carefully devised as part of the language design and then debugged by the compiler-writer before being presented to the user. The syntax is usually not presented as a user-mutable part of a system, although for the sophisticated user, the grammar and source-code and tools may be revisited.

The application programmers or more particularly users of mathematics' notation, have historically not used this technology directly. They are faced with the rather bleak prospect that, notationally speaking, all computer science has to offer is a sequence of essentially non-responsive answers to the question of application-oriented computer languages. Occasionally an application package emerges from a combination of complexity and the personal ambition of a clever programmer. A system-oriented application programmer may hide the complexity behind a kind of macro-expansion layer. Packages for interval arithmetic, polynomial arithmetic, high-precision software floats, etc. emerge. In newer languages these efforts are subsumed in some object-oriented framework, which may or may not provide adequate support.

Without some good luck, the physicist is most likely advised to recode thoughts into Fortran, C++ or Java. And that instead of writing in vector or tensor notation, the scientist should learn about classes and templates.

If notation is a key to thought, what does it mean if computer scientists require abandoning conventional notation and adopting perhaps poorly-fitting notation? Is there a solution to making use of a more adaptable syntax?

It appears to us that syntax extension can be successful in two distinct approaches.

- The simple syntax "hack" which can add a modest operator or two to the standard computer algebra system for avoiding what would be considered "something too tricky/messy to explain" to an otherwise naive user.
- The new language design. Part of this is close to the approach given by Jason Harris with Mathematica: He showed that a skilled worker can come up with a CAS display that matches the standard typeset appearance in technical articles in a field. The other half of the problem is not solved if the input to produce this requires considerable skill.

  A more serious systematic approach would look like this: identify a "killer application" for which many users are eager to use a computer system, and which one (or a few) developers are willing to devote considerable resources to the careful design of a specialized language: the new syntax and semantics for that application. In general the developers will either have to carefully model an existing text, journal article, or other popular reference. Alternatively one could produce what amounts to a new reference text. This may also require introductory user material development ("Killer App XXX of CAS for Dummies").

This line of reasoning assumes that the main program governing interaction is the CAS, and so of course all the facilities will necessarily be integrated with the CAS. It is not necessary that an application involve the development of new algorithms: it could be that the CAS is essentially hosting an expanded notion of the notebook formats currently available. The expansion is to incorporate a new prologue consisting of the new syntax and perhaps a link to a tutorial on its use.

An alternative approach is to produce an entirely new front end in which the user interaction is not moderated by the CAS; the syntax of the front end, perhaps web-based, may be some kind multiple-choice forced-input menu, a template selection system, or even a traditional re-invented parser (a recent example that crossed my desk is an implementation of an Einstein-summation "array language" [1]). In the case of a separate front end, the conversation with the CAS back end could be done without syntax extension of the CAS. The external syntax has, after all, been taken over by the user interface program, and the rest of the program could converse in CAS-speak or even something like MathML. The independent development of a front-end parser "from scratch" strikes us as more labor-intensive, especially when the CAS already has essentially similar facilities in its front end. Of course using unfamiliar (and perhaps proprietary) tools is always a barrier..

Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. David Park and Paul Abbott have provided some additional insight into the usability of the extensive Mathematica notation extension package. Steve Swanson of MacKichan has offered helpful insights from the perspective of Scientific WorkPlace.

1. REFERENCES

[1] K. Ahlander. "Einstein Summation for Multidimensional Arrays," *Computers and Mathematics with Applications 44* no. 8-9, 1007–1018.

[2] D. Blostein and A. Grbavec, "Recognition of Mathematical Notation," in *Handbook of Character Recognition and Document Image Analysis,* Eds. H. Bunke and P. Wang, World Scientific, 1997, 557–582.

[3] Digital Library of Mathematical Functions (NIST project) `http://dlmf.nist.gov/`

[4] Kenneth E. Iverson. *Notation as a Tool of Thought*[17].

[5] Donald E. Knuth, *The TEXbook*, Addison-Wesley, 1984.

[6] MINSE `http://web.lfw.org/math/`

[7] MathML `http://www.w3.org/Math/`

[8] Vaughn Pratt, "Top down operator precedence",*Proceedings of the ACM Symposium on Principles of Programming Languages*, Boston, October 1973. Association for Computing Machinery. 41–51. One stand-alone application of this was an Algol-style top-level for MacLisp called CGOL written by Pratt in 1977. A version of CGOL translated to Common Lisp by Thomas Phelps and Richard Fateman, along with some associated documents can be found in the web directory `http://www.cs.berkeley.edu/~fateman/cgol`.

[9] Stephen Wolfram, "Mathematical Notation: Past and Future" in MathML conference, 2000. `http://www.stephenwolfram.com/ publications/ talks/mathml` or `http://www.mathmlconference.org/2000/ Talks/Keynotes/sw/`

[10] Rick Warfield, Ambiguous Mathematics Parser (part of Tilu, `http://www.torte.cs:8010/tilu`).

[17]Comm. ACM 23(8) 1980, 444–465