Parsing TEX into Mathematics

Richard J. Fateman
Eylon Caspi
Computer Science Division, Electrical Engineering and Computer Sciences
University of California, Berkeley

August 4, 1999, revised July 2, 2002, May 6, 2004

Abstract

Communication, storage, transmission, and searching of complex material has become increasingly important. Mathematical computing in a distributed environment is also becoming more plausible as libraries and computing facilities are connected with each other and with user facilities. TEX is a well-known mathematical typesetting language, and from the display perspective it might seem that it could be used for communication between computer systems as well as an intermediate form for the results of OCR (optical character recognition) of mathematical expressions. There are flaws in this reasoning, since exchanging mathematical information requires a system to parse and semantically "understand" the TEX, even if it is "ambiguous" notationally. A program we developed can handle 43% of 10,740 TEX formulas in a well-known table of integrals. We expect that a higher success rate can be achieved easily.

1 Introduction

One goal of our work in digital library data acquisition has been to read and "understand" printed pages of mathematics that have been scanned into a computer. In particular, we wish to process formulas into a computationally tractable form so that computer algebra systems can use them. In this paper we address a practical issue that repeatedly arises, which is to process legacy documents that need not be scanned because they are "born digital." Yet the formulas are not ready for further processing (except for re-typesetting!) because they are in TEX [14], not in a semantically unambiguous form that could be used by (say) a computer algebra system (CAS). For this we need a semantically unambiguous notation specific to a CAS or perhaps some interchange format such as the recently proposed MathML[1] `http://www.w3.org/Math` or OpenMath [13] notations. An associated question arises if we start with scanned pages of material: can we consider transforming the collection of glyphs into some form of TEX as a useful intermediate form, and then subsequently try to figure out the semantics? [21].

1.1 2-D notation

We begin by distinguishing two kinds of 2-dimensional (2-D) input mathematical notation, and imagine that we wish to "understand" this notation somehow. Our intent is to relate this to using TEX.

The first and most intuitive version of 2-D mathematics *input* is the notation of simple numbers and variables, divide-bars, and the operators and symbols of "elementary" algebra. This notation is rich enough that it can be used for impressive brief demonstrations of hand-written input to computer programs, and it can be modeled neatly as a formal language. Programs for interpreting such material are commonplace today[2]. Unfortunately, most human observers read too much into the demonstrations of apparent success in

[1] (

[2] Our current favorite is JMathNotes, a free download from http://page.mi.fu-berlin.de/ tapia/JMathNotes

this domain. Partly the demonstration does not illustrate the restrictions to the language or the user-training involved, and partly the success conceals the problems associated with *extensions* to this first-level version. In fact an approach to more advanced math presents significant barriers to a complete solution.

In this second advanced level we place a fuller set of notation appropriate for more elaborate mathematics. This includes conventional communications as might appear in mathematical or physical sciences texts or journal articles. This level must necessarily include integrals, derivative notations, summations, multiple-line equations, equation numbers, side-conditions, subscripts, and a full character set including additional alphabets, fonts, and various operators and brackets. It also include notations for various algebras, diagrams, tabular representations, and should allow notations that are constructed for special purposes "on the fly". Among the most difficult tasks here is to decode the meaning of the spaces between glyphs. For example, some spaces denote "functional application" or multiplication, and other, perhaps larger, spaces might denote "subject to the condition" or "indexed by the equation number".

In this fuller set, can we simplify our task of understanding the arrangement of symbols as handwritten or as scanned from a static printed page by first deducing the TEX input that would have produced the 2-D display?

Perhaps: but even this is not a full solution: Portraying 2-D display as TEX, but without some context, fails to disambiguate the expressions. Indeed, some of the hardest parts of recognizing 2-D typeset mathematics are present even in a linear form!

How could we describe this context? One *ad hoc* method is to write a special-purpose program to decode a body of TEX. The context is implicit in the program interpreting the image. A more systematic approach is to find a description for the context separate from the program, to make the program more easily adapted. This context could take the form of a dictionary and set of rules or grammar for the particular work or body of work or a "running commentary" on the semantics of the mathematics. It is easily demonstrated that a single context for "all" mathematics cannot be constructed, since advanced notations often are re-used or "abused." Yet if the notation is to be useful as a communication mechanism among some group of humans, it must be understandable in *their* common context[3].

Of course it is clearly possible to map the TEX recognition problem to a 2-D math recognition since the readily-available TEX program converts TEX input from linear text to 2-D positions on a page (in dvi format)!. For our purpose here, we take the view that TEX is (or can be) a convenient entirely-linear notation for describing graphical mathematical formulas, and in that sense is superior to the 2-D dvi for further work. If we can solve recognition problems on this common notation incorporating positional information, namely parse it into a semantic version, we have made headway in understanding mathematical documents by computer.

Why do we find ourself concerned about using a TEX-centric approach, rather than (say) some other notation, such as that proposed for computer algebra systems (CAS) or for Openmath[13]? The strength of TEX, and in fact the near inevitability of having to deal with it at some level, is based on its centrality in some technical publishing environments. TEX allows us to (a) represent virtually any article published by the AMS (American Mathematics Society) in recent years, and (b) take advantage of the programs for typesetting, auxiliary display, printing, etc. that have grown up around TEX. (One of our favorites is AsTeR [1], a program to read mathematics out loud).

Before going further, perhaps we should explain for the benefit of readers who are familiar with TEX but who have not thought about this problem, why it is that TEX *per se* does not "understand" the mathematics in any direct sense. Whether we have TEX or just ordinary "natural" mathematics notation, resolving ambiguity by the use of local and global context seems necessary to finally map the notation to semantically useful mathematics. A simple example we will revisit can be helpful: one cannot tell from TEX if $u(v+w)$ is a multiplication or an application of the function u. TEX merely encodes the appearance, ambiguity included.

[3]It is possible that "punning" notations, intended to be read in several different ways, are used by authors.

1.2 Some Perspective

The typesetting of mathematics by computer is a well-studied area with the emphasis on the generation of quality results [14]. In the past 15 years there has been growing interest in the optical recognition of non-text written material, including typeset mathematics [3]. Judging from the absolute volume of printed scientific material compared to all printed work, input or output of mathematics is low on the scale of importance for commercial word processing or OCR scanning programs. Yet an economical and effective solution is important for scientific documents: In this era of digital libraries, effective storage and efficient retrieval of text which includes mathematics, seems to be of interest; in fact, specific retrieval of formulas from within documents has applications as well. A specific example of a highly-useful on-line scientific retrieval which could benefit from better processing of formulas is the NEC ResearchIndex program at `citeseer.nj.nec.com/cs`.

Furthermore, and from our perspective more interestingly, it is possible to turn this kind of recognition activity to the future, and hope to improve scientific productivity. It is now more plausible than ever to engage in a two-way dialog with computer algebra systems (CAS): these are now routinely producing unwieldy expressions that can appear in documents. Sometimes the documents are contained and composed in a proprietary "notebook" form with a limited mathematical notation, as in Mathematica, Maple, Macsyma, Axiom, Theorist. These are CAS-centric, and therefore as a practical matter are limited in scope (mathematically) to the commands and data representation requirements of their host CAS. Yet they typically allow typesetting of other notations and can be extended programmatically to other "constructive" mathematics, so viewed more abstractly, the ultimate limitations of the CAS-centric view are effectively the limits of computation (arguably, these same limits apply to humans or computers.)

We have now seen several prototype attempts to free notebooks from their ties to specific systems (e.g. CAS/PI, Mathscribe , SUI, GI/S, CaminoReal, JMathNotes, TeXmacs [4]), and emerge with a software component approach [13] It is not our intent to provide a survey (for which you may see the bibliographies of [22],[18], [16]), but just a few comments. In effect these interpose a WYSIWYG layer between the CAS and the human user. The front-end program translates from a human-oriented input notation to the "necessities of computing" notation.

A different approach is to start afresh and develop new "standards" or base solutions to mathematical representation. A success here would *a forteriori* provide a solution for mathematical representation for CAS. On the down-side, an attempt to encode all mathematics: past, present (and future?) is, to say the least, quite ambitious. Nevertheless, we are aware of several activities that seem to have this goal, either implicitly: OpenMath or similar component technologies [13], Mathematical SGML/ MathML [12]; or explicitly: the QED project [19] or Zhao's system [22]. Zhao et al. states explicitly, "Our research aims to... study the structure and meaning of all mathematical expressions so as to recognize their linguistic and logical properties." The Polymath group at Simon Fraser University has also tried to address this problem[5]. While some of the advocates of these some of these forms claim they are complementary, it seems that they are generally in competition with one another.

We agree that there are good reasons for trying to come up with a single universal grammar and semantics for mathematics notation, but we are concerned that any efforts to find a fixed and complete notation must founder on the shoals of ambiguity. Even granted some oracle of disambiguation, it appears that total generality must require fairly substantial extensibility and the ability to incorporate context into interpretation. One simply cannot expect to represent all past and especially all future mathematics with a fixed set of notations. Therefore one must provide tools for extension that are sufficiently "universal" for all further work.

Any such coverage of mathematics apparently must be extensible, and it appears that each group relies

[4] www.texmacs.org

[5] There is a demo of a "latex2openmath" tool at `http://pdg.cecm.sfu.ca/semantics/LaTeX2OM.html` which,if it worked, would solve this problem. It doesn't seem to work, and its strategy is not described except to say it uses an AI language called Wildlife.

to some extent on type-systems (in the sense of programming languages), to help in disambiguation. We are not convinced that types hold the key to being effective, although that may help.

The success we demonstrate in this paper shows an alternative, that *if one sets a finite context and looks primarily at its notations and semantics*, the prospects grow brighter.

There is yet another approach, which is to place at the center of the activity some WYSIWYG word-processing program incorporating mathematical notation, add a user-interface (menus, commands) allowing the selection of expressions to be shipped to CAS, and the values returned. Current programs that have an independent life as word processors but provide links to such computer algebra systems include Techexplorer [11] with an optional link to Axiom as well as Scientific Workplace [20] with an optional link to Maple or Mathematica or (as of 2004) MuPAD. However, this can only work for material that is "born digital," or being entered into a system *de novo*. This is a major limitation for the large number of "legacy" mathematical documents already on paper or in TEX.

1.3 Where does TEX come in?

"TEX is designed to handle complex mathematical expressions in such a way that most of them are easy to input." says the TEXbook [14]. Judging from the many pages that have been typeset using TEX it appears that Donald Knuth has been successful with the design. What does Knuth mean by "handle" here? He certainly does not mean that TEX as used normally, is an unambiguous *encoding of all of mathematics.* Without sufficient context beyond that needed to typeset, even simple expressions are ambiguous. That is, in spite of being specifically, precisely, and neatly typeset by TEX an expression out of context may not provide an unambiguous statement to a mathematically literate observer. As a simple example, consider the expression mentioned earlier: $u(v + w)$ which has at least two distinct conventional interpretations: the function u applied to the sum $v+w$, or the product of u and $v+w$. Without further context one cannot tell if u is a function or an operator (e.g. derivative with respect to x), or an operand to the implicit multiplication. There are other common interpretations for this same expression including the boolean expression more expressively denoted $u \wedge (u \vee w)$, as well as possible expressions in algebras of strings or other domains. And by the magic of mathematics, one can overload this expression with freightloads of meaning. This is not a detriment to TEX. Indeed, if Knuth had approached the task of encoding all of "mathematics," the simpler objective of the encoding of "conventional mathematical typesetting," might have been lost.

1.4 Why bother?

Let us review the practical matters at hand: why TEX , and why now?

1.4.1 TEX is already in use

TEX has merit as an intermediate language associated with a mathematical graphical user interface. Commercial tools such as IBM *TechExplorer* [11], TCI *Scientific Workplace* [20] and *MathType*[15] exist today which allow the user to build mathematical formulas on a computer screen using positioning templates (superscript box over base-text box, integral sign with boxes for integration-limits and integrand, etc.). The tools then emit a TEX or other visual encoding of the formula (e.g. JPEG, PNG, GIF) to be pasted into a word processor. This method of user input, because it is graphical, is arguably the next most natural method apart from handwriting formulas. Because it is based on rectilinear templates, this method is in fact one-to-one transformable to TEX and is burdened with all the semantic ambiguities present in TEX code. Hence a mathematics recognition engine must still be run in order to disambiguate the mathematics notation. Some of the aforementioned tools include recognition engines with interfaces to computer algebra systems (*TechExplorer* to *Axiom*, *Scientific Workplace* to *MuPAD*). The same job could be done by a modular TEX-based recognition engine.

1.4.2 We would like to read Gradshteyn and Rhyzik

Often the general case can be illuminated by a well-chosen special case, and the special case we have examined is the domain of "advanced calculus" as exhibited by common texts, reference works (tables of integrals), and the relevant areas of physics and engineering that rely on this level of mathematics. Suppose that we wish to parse mathematics as normally typeset, into "semantically reasonable" representations suitable for use by a computer algebra system. Let us be quite specific: how hard is it to parse the recently re-typeset reference *Table of Integrals, Series, and Products* by Gradshteyn and Rhyzik, Academic Press (5th edition) from TEX to the moral equivalent of a CAS? This well-known reference, henceforth G+R, was re-published on CDROM in 1996. The internal version combining SGML and TEX can be viewed and downloaded from CDROM to conventional ascii files. Academic Press's initial goal in the CDROM was to provide a "Dynatext" version[7] which from our perspective was inadequate for CAS use.

Is it possible to take a domain broadly classified as "advanced calculus" and parse most of G+R within this domain? Our thesis is: yes. And in fact we believe we can translate wholesale much of the work done by Academic Press and its partners in converting this reference into TEX.

We realize that it is somewhat pointless to write a program to translate automatically the occasional one-of-a-kind details that appear in this book, especially if there is no obvious corresponding semantic structure in computer algebra systems. But if the bulk of the text is parsed automatically the remaining details can be converted by hand.

Alternatively, if labor is sufficiently inexpensive, the whole book could be encoded in a CAS language (or a macro-language devised especially for the book by Daniel Zwillinger) from scratch, as was in fact done subsequent to the writing of our program and this paper. (G+R, 6th edition)

1.4.3 The challenge

Such work (typesetting of math) has been done by humans, who subsequently can, at least sometimes, make sense of it, so we have an existence proof. From the computer perspective, this is a challenge in representing, communicating, and computing symbolically using mathematics that must be overcome for us to succeed completely with OCR, also.

Is it plausible to use TEX in some substantial way for this task? Our initial impulse is to say it is just too ambiguous, and provide, as we do in later sections of this paper, examples of difficulties. Yet, while TEX unassisted, cannot do it all, we believe we are able to combine TEX with other notations effectively. We can use contexts and disambiguation explicitly in expressions, e.g. `{\the-operand u}` vs. `{\the-operator u}`. Alternatively, and the method used here, we can use implicit contexts which amount to computing with declarations of the form "until further notice, all symbols are variables except for sin, cos, tan, and f."

2 How to Proceed

There are several steps needed, including

- Describing formally some domain of interest $\mathcal{D}$ in some computer-oriented natural (presumably textual but non-TEX) form. This is the target into which the TEX will be translated. Tradition in Lisp provides for $ab\sin c$ to be represented as `(* a b (sin c))` but we cannot cite historical precedent for Lisp forms for domain restrictions, provisos, citations, geometric descriptions of branch-cuts, etc. Although once one unambiguous structure is provided for the mathematics, converting it to another is generally straightforward.
- Describing all the TEX usages that would commonly be found in describing expressions in that domain. This would ordinarily be described by formal tools. In practice, formal grammars are the tools for automatically deriving parsing/translating software. A more practical approach is to divide the task into lexical analysis and context-free parsing (if such is possible).

- Devising a translator from the TEX into $\mathcal{D}$. Ordinarily one should describe the language of interest, usually at the same time that one devises a formal grammar for that language. One also defines appropriate augments to the grammar so as to produce a representation of sentences in that language. The conventional computer science approach is to present this grammar in a formal way to a parser generator such as `yacc` or `bison` for consideration. After some careful grammar modifications required by such tools, one then has a custom-built parser.

For this to work we must ask: does mathematics (as expressed in TEX) have a formal grammar? Ideally it would be context free and no worse than LALR(1), for the common tools to work [2]. Initial work by Fateman was based on the premise that context-free simply would not work, and after some prototype programming, the task was abandoned. He then suggested it as a project in a graduate class (CS282, Fall, 1997, UC Berkeley) and Eylon Caspi took up the challenge. This paper recounts some of Fateman's efforts and then Caspi's further results.

2.1 Calculus, and a view from TEX

TEX already knows about certain functions like sin, cos and log. G+R uses others such as the logarithmic integral li or Re (for Real Part). Given such a list of functions it appears we know that $\log x$ is not multiplication but function application. Oddly enough, even though functions such as erf (the Error Function) and J_ν (Bessel Function of the first kind) also appear in such a list, erf$x$ and $J_\nu x$ are never used. We see only $\mathrm{erf}(x)$ or $J_\nu(x)$. Go figure.

In any case, once we have decided that for some f, $f(x)$ is a function application, we know that $\int_a^b f(x)\,dx$ probably makes sense. We suspect that $\int_a^b d(x)\,fx$ does not make sense, although TEX really doesn't object to the latter.

2.1.1 TEX Oddities

TEX enjoys, or suffers from, a number of oddities compared to common mathematical notation.

2.1.2 Numbers

Numbers are not treated as compound tokens composed of digits. They are separate tokens that just happen to appear to fit together. This is counter-intuitive and leads to a common error we (and apparently many others) commit, namely to write `$x^23$` to be typeset as x^{23}. Instead one gets $x^2 3$. One must utter `$x^{23}$` to get the expected result. And there is no offense taken by TEX with odd number-like objects like $123,456.78.0.0$ Furthermore, the notations `$1234$` and `${1}2{{3}}4$` typeset to the same appearance, namely 1234.

2.1.3 Precedence of operations

Although Knuth was obviously well aware of the nature of parsing and the role taken in interpretation of mathematics by operator precedence, such interpretation is outside the scope of TEX for the most part. Precedence is revealed only by certain subtle TEX rules such as providing decreased spacing around the / compared to the space around + in $a + b/c$.

Indeed, if one uses the `\over` construction, one sees that "usual precedence" doesn't work: note this: `$$a + b \over c$$` typesets as

$$\frac{a+b}{c}$$

That is, the whole formula to the left of `\over` is the numerator. In fact, `$a \over b \over c$` provokes a warning message, although it typesets as equivalent to $a/(bc)$. Knuth also warns against `$x^y^z$` which it will typeset as x^{yz} if you insist.

2.1.4 Faithfulness of bracketing

Here is another decision point facing us. In our analysis of TEX input, must we interpret two utterances as identical in meaning if they produce the same output? Using the TEX input language as a conveyance for mathematics, one might wish to clarify the order of operations by the braces `{}` notations, even though they are invisible in the typeset output. One choice would be to allow braces to affect the "meaning", as for example
`${a+b}c$` does the addition *then* the multiplication versus
`$a+{bc}$` the usual precedence, or
`$a{+b}c$` which seems bizarre at first glance. (This has tighter spacing than the identical previous two lines. Conceptually it consists of the "product" of a, $+b$ and c: $a{+}bc$ and reflects the fact that TEX identifies the $+$ as a unary prefix operation, not an infix binary operator. If we write `$a{{}+b}c$`, the results are again the usual $a + bc$.).

But grouping the input by braces does not particularly change the output of TEX, and so it may be more sensible to ignore the braces, and this is the rule we finally used: remove almost all unnecessary braces.

Braces have another unfortunately arbitrary characteristic, grammatically speaking. `{\int}`, or for that matter, `{{{\int}}}{}` can be used anywhere that `\int` can be used. This presents complications to certain simple parsing techniques, in particular forcing one to have only one syntactic non-terminal symbol.

2.1.5 Ambiguity of high-level spaces

Normally spaces as multiplication imply very tight binding: the only tighter-binding operation is exponentiation. That is, ab^c means $a \cdot (b^c)$ not $(ab)^c$. And in particular, $a = bc$ conventionally is grouped as $(a) = (bc)$. But some spaces are very loosely binding. Consider the space to the left of $n \neq -1$ in:

$$\int x^n dx = \frac{x^{n+1}}{n+1} \qquad n \neq -1.$$

To any even vaguely attentive mathematics student, in this expression "=" binds more tightly than the space, and it would be quite erroneous to interpret this as

$$\left(\int x^n dx\right) = \left(\frac{x^{n+1}}{n+1} \qquad n \neq -1\right).$$

While some disambiguation can be based on the *amount of space*, this is not foolproof.

2.1.6 Variable vs. Function

Several ambiguities arise when it is not known whether a symbol represents a variable or function. For instance, one cannot tell in such a case whether the form $a(b)$ is a multiplication or a function invocation. For a variable a, a' typically refers to a related second variable, whereas for a function f, f' typically refers to some derivative of f. Similarly, for a variable a, a^{-1} is a reciprocal, whereas for a function f, f^{-1} is typically the inverse of f.

Ambiguities of this form are easily overcome *if one can determine the unique type for the symbol in question.* If a symbol dictionary is not available, a recognition engine may be able to determine the type from context. For instance, the form $a(b, c)$ is easily seen to be a function of two variables. If a symbol is always followed by parentheses, even when parentheses are unnecessary for a product, then it is likely to be a function. Humans have many such cues to guess at a variable type, and clever AI techniques can seek to make use of such cues. When all else fails, a recognition engine might ask the user interactively for information.

2.1.7 Operator Notation

Operator notation allows one to specify a function without parenthesizing its expression, as in $\sin x$. The primary difficulty with such notation is in deciding how much of the right-side product expression is actually affected by the operator. In practice, the extent is different for different operators. It is conventional, for instance, to split trigonometric operations at spaces, as in:

$$\sin x\ yz = (\sin x) \cdot (yz),$$

as well as at the next operator, as in:

$$\sin x \sin y = (\sin x)(\sin y).$$

This is not the case with the sigma summation operator, which nests, rather than breaks, at the next summation:

$$\sum_n n^2 \sum_m m^2 = \sum_n \left(\sum_m (n^2 m^2) \right).$$

In this case there is no real ambiguity, since different operator classes are known to have certain spatial binding.

Context-sensitive ambiguities arise when the extent of an operator depends on the nature of its arguments. For instance, the size and content of a fraction may determine whether or not it belongs to an operator, as in:

$$\sin \theta \frac{\pi}{2} \stackrel{?}{=} \sin \left(\theta \frac{\pi}{2} \right),$$

versus:

$$\sin \theta \frac{x+y+z}{n!} \stackrel{?}{=} \sin(\theta) \cdot \frac{x+y+z}{n!}$$

and similarly, whether a term to the right of the fraction belongs to the operator:

$$\sin \frac{\pi}{2} \theta \stackrel{?}{=} \sin \left(\frac{\pi}{2} \theta \right),$$

versus:

$$\sin \frac{3\pi\theta}{2} x \stackrel{?}{=} \sin \left(\frac{3\pi\theta}{2} \right) x$$

It is not always clear whether a function belongs inside or outside an operator argument. For instance, with f being a function, we cannot conclude whether:

$$\sin n\pi f(b) = \sin(n\pi)\, f(b),$$

or:

$$\sin n\pi f(b) = \sin(n\pi\, f(b)).$$

For a capitalized function F one may be more inclined to choose the former.

Contextual dependencies on variable names also exist for the division operator. For instance, it is not clear whether:

$$1/2\pi(a+b) = \frac{1}{2\pi(a+b)},$$

or:

$$1/2\pi(a+b) = \frac{1}{2\pi}(a+b),$$

or even:

$$1/2\pi(a+b) = \frac{1}{2}\pi(a+b).$$

The way in which variable names and fraction sizes indicate binding of an operator is subjective to each user and sensitive to the domain of computation. In such cases, a recognition engine cannot make a decision on operator precedence without consulting some user-preferences guide or even making an interactive inquiry.

2.1.8 Derivatives and Integrals

Calculus notation has some interesting ambiguities when dealing with differentials. Syntactically, a differential dx has the same form as a product. One may wish to think of the d as an operator, but its binding may be ambiguous, for instance with juxtaposed differentials. Matters are ever more complicated if one introduces a variable d which is not meant to form differentials.

The derivative form $\frac{dy}{dx}$ and derivative operator $\frac{d}{dx}$ are syntactically indistinguishable from fractions, and their semantic meaning can be quite subtle. For instance, $\frac{dy}{dx}$ is a stand-alone fraction, whereas $\frac{d}{dx}$ is an operator affecting some expression to its right. Also, $\frac{d}{dx^2}$ is a first derivative (with respect to x^2), whereas $\frac{d^2}{dx^2}$ is a second derivative. Analyzing so many parts of an expression to determine its collective meaning as a derivative is cumbersome at best.

The integrating variable of an integral resides in a differential which may appear in several positions, depending on the form of the integrand. For instance, the following forms are equivalent:

$$\int \frac{1}{\sin x}dx = \int \frac{dx}{\sin x}.$$

With multiple integrals, differentials may appear most anywhere in the integrand, as in this unconventional but mathematically sound form for a spherical-coordinate volume integral:

$$\int_a^b dr \int_0^{2\pi} \int_0^{\pi} f(r,\theta,\phi) r^2 \sin\theta d\theta d\phi.$$

Placing derivatives in the integrand further complicates the job of finding the correct differential.

Dealing with syntactic ambiguities requires special care in the parser. Because differentials are syntactically equivalent to products, it is possible for the two visually-adjacent characters of a dx differential to become separated in the syntax tree. A parser may prevent this by including a differential form beginning with the letter "d" in the grammar, and by applying a semantic pass later to split products that merely look like differentials. A similar technique can be employed to keep a suspected function and its parenthesized argument together in the syntax tree. Such design decisions bring mixed blessings, as they complicate a grammar and introduce new difficulties in the syntax tree, such as having to split up an incorrectly grouped argument for an operator. For instance, if parenthesized forms such as $b(c)$ are always parsed as function invocations, then a subsequent correction when b is a variable requires transforming a possible initial parse of $\sin ab(c)de$ as $\sin(a)b(c)de$ into: "sin(ab) (cde)".

2.2 Criteria for Parsers

One might argue that we can use any parsing technique that passes a "sanity check" on our interpretation. The check is produced by taking the algebraic result of the parse and converting back to TEX. A pattern

matching program can then make some effort to reconcile the re-conversion to TEXwith the original input, or one can produce a printed page with the original and the "round-trip" result. This does not provide a full check, and is not as foolproof as might seem at first: Our current program cannot always tell if two horizontally adjacent expressions are actually multiplied together, or might represent a operator/argument relationship. The parser then just produces a neutral structure saying only that these two expressions are concatenated. They are re-typeset in the same ambiguous fashion, and the human reader, naturally motivated to believe that the two expressions are the same, accepts them as such, not noticing the ambiguity. Some expressions are really done poorly. Following the rule that parentheses, when present, delimit arguments of functions, $\cos(p)r$ means the argument of the cos is p. Thus $\int \ln(\sin \pi x) \cos(n+1)\pi x dx$, appearing in GR as formula 4.384.3 would seem to involve $\cos(n+1)$. In the context of an integral with respect to x this is apparent nonsense, and so one must interpret it as $\cos((n+1)x)$. (We provide more examples in which humans or computers may have trouble.)

Even if the sanity check detects differences, these may be harmless; the result of removing some explicit spacing, noting the size specifications of brackets, etc. Another difference may involve, the translation (as we have done), of non-standard notation and abbreviations to standard ones, e.g. the cotangent function is written as "ctg" rather than "cot", or the regularization of notations involving summation limits, etc.

More seriously garbled results either suggest that the input language is "more advanced" than we anticipated in our parser or perhaps that our parser is incorrect for some inputs.

We are also a bit uncomfortable about the prospect that humans may easily modify precedences in relation to context in a non-algorithmic way: Two humans may agree that $1/2\pi(\cos nx + \cos(n+1)x)$ means

$$\frac{\cos(nx) + \cos((n+1)x)}{2\pi}.$$

(Take a minute to check this out yourself!) But some computer programs would interpret this as

$$\frac{1}{2} \cdot \pi \cdot ((\cos(n)) \cdot x + (\cos(n+1) \cdot x)).$$

We can argue that $1/2\pi$ means $1/(2\pi)$ because if we had meant $\pi/2$ we would have written it that way.

Indeed, such hackishness has been encoded deliberately in Scientific Workplace, which interprets $\sin \pi/2$ as `sin(pi/2)` but the more general $\sin a/(b+c)$ as $\frac{\sin a}{b+c}$. There is an opportunity to ask the system to explain unambiguously what its interpretation is in such cases, but in our experience it is in human nature to fail to check such matters when they really count. See the appendix on "Spaces the final frontier".

Other declarations of values or functions must also be included in some contexts. An explanation in a table of integrals may assert that $u = \sqrt{a^2 - x^2}$ and thus a CAS must realize this substitution is relevant when u appears in a result. Another example is when one is told H stands for "any Bessel function". (The three Bessel functions are normally denoted by I, J, or K, and each appears as a single-subscripted function of one argument.)

3 Fateman's First TEX Parser

It is, in some sense, trivial to define a formal grammar that models TEX's math mode. Unfortunately such a grammar models only the notion of a math-expression, and a few positional operators. It does not address the disambiguation of operations and their precedences. Undergraduate compiler courses would make one believe in the power of formal grammars and automatic parser generators: the generative prospects of TEX as used in mathematics is so far from a context free language.

In an effort to specifically solve the problems from the Academic Press CDROM [10], we started with a recursive-descent (partly automatically generated, but mostly hand-written) parser written by R. Fateman (in Lisp) previously for a computer algebra system. We extended it to absorb TEX as used by AP, and convert

it to data in Lisp, in a form that is easily digested by a computer algebra system. Not every utterance in the reference has an immediate correspondent in a computer algebra system, and so we have had to make up some notation "not yet available" in the algebra system. It is clear that completely encoding the knowledge of this volume requires thoughtful extension of computer algebra systems.

4 Caspi's Better Parser

Based on similar requirements, Caspi wrote a multi-pass mathematics recognition engine designed with the specific intent of transcribing formulas from the electronic form of G+R into LISP statements suitable for a computer algebra system. The engine was aimed at transcribing a sample of 210 integration and summation formulas in the domain of real, scalar calculus[6]. It succeeded in understanding 154. A more challenging test on 10,740 formulas is described in a later section of this paper.

4.1 New TEX Parser Overview

Ideally, a computer program would convert the bulk of formulas into a disambiguated output language, leaving only a residue of difficult, syntactically dubious, or clearly ambiguous cases for human inspection and transcription. We would like the output language to be general and easily parsed by a computer so that it may be mapped into the languages of other computer algebra systems. We chose LISP symbolic expressions as the target data language for its simplicity in reconversion, and the fact that it is in wider use than other possible choices. Several computer algebra systems in use today, for instance Axiom, Reduce and Macsyma, are written in LISP and/or allow the user to interact with the system directly in LISP. Conversion from LISP to Mathematica or Maple syntax is rather simple as well.

Fateman's efforts described above suffered by the need to repeatedly modify the hand-coded parser, based on the assumption that the mathematics notation would not yield to ordinary context-free parsing techniques. By contrast Caspi (not so keen on modifying Fateman's LISP programming) tried to push the "conventional wisdom" of using compiler-development tools including *Perl*, *lex*, and *yacc*. An initial recursive-descent pass written in *Perl* expands TEX macros and removes unnecessary (invisible) curly braces. A second pass written in *flex* and *bison* with C++ converts the adjusted TEX code into an abstract syntax tree based on a context-free attribute grammar for TEX expressions. Several passes over the syntax tree then deal with context-sensitive and semantic aspects of mathematical expressions, including disambiguating the use of primes and parentheses on function symbols versus variables, and identifying the integrating variables of integrals. A final pass prints a parenthesized version of the tree suitable for read-in by LISP. The final pass over the expression could then adapt the notation to a specific structural requirement.

The initial performance of the recognition engine was encouraging. It was able to successfully convert 154 of 210 formulas from those ten of G+R's eighteen chapters which deal with scalar integrals, summations, and products. These formulas comprise the stand-alone formula listing of the reference — the narrative text and its shorter embedded expressions were filtered out of the parser's input.

4.2 Parser details

4.2.1 Input Domain

The table of integrals contains eighteen chapters, covering such topics as basic series, definite and indefinite integrals of elementary as well as special functions, vector field calculus, matrix calculus, and differential equations. To limit the complexity of the recognition engine, we chose to limit the input to the first ten chapters, namely those dealing with integrals, summations, and products of real, scalar quantities. The

[6]The encoding of G+R in Dynatext made it difficult to dump more than this number of expressions. We subsequently hacked a way around this limit.

latter chapters involving vector, matrix, and complicated derivative notation remain beyond the scope of this project.

Within the ten chapters considered, limitations of the CD-ROM "export" software provided easy access only to the first 210 formulas those chapters of the reference which we handled for our experiment.

The `$$` formula blocks comprise the main listing of formulas in the reference. Most of them appear in a common format consisting of an optional formula number, a primary equation, and an optional list of relations denoting conditions required in order for the primary equation to hold. The spacing and punctuation varies among formula blocks and provokes some otherwise needless special-case syntax rules.

The TEX code appearing in the formula blocks consists of plain-TEX math code and several $\mathcal{AMS}$-TEX macros such as `\dbinom`. Trigonometric and other familiar functions appear in non-italic Roman font, using backslash control sequences, `\hbox` constructions, or `\operatorname` constructions. The source uses many trigonometric functions with alternate spellings, for instance `tg` for tangent and `ctg` for cotangent. A variety of spacing constructions are employed, including the $\mathcal{AMS}$-TEX `\align` macro.

4.2.2 The Passes

The recognition engine uses multiple passes written using a number of parsing tools and computer languages to go from TEX to LISP. The modularization of passes allows us to employ multiple parsing techniques in succession, breaking the recognition problem into more easily managed phases.

4.2.3 Expanding TEX Macros

A first pass is done to expand user macros defined using `\def` and to apply `\input` file inclusions. Note that we do not include the $\mathcal{AMS}$-TEX style files in this pass, as they would only complicate parsing by expanding $\mathcal{AMS}$-TEX macros into low-level TEX formatting commands. We do use macros to define away some unwanted constructs from the input, including, for instance, the `\align` macro. This pass is written as a simple recursive-descent parser in *Perl*, as it need only consider the `\` backslash and `{}` curly brace characters to recognize TEX macro syntax.

4.2.4 Adjusting {} Curly Braces

A second pass is done to strip away unnecessary curly braces from the TEX code, and to insert them in critical places that assist the next pass. Braces are used in TEX as syntactic separators and, in addition to being non-printing characters, do not affect the displayed results except in specific constructions as mentioned earlier. Brace pairs are removed in this pass unless they follow a backslash control sequence (or known multiple-argument control sequence such as `\dbinom`), a `^` exponentiation caret, or a `_` subscripting underscore, or unless they contain the `\over` or `\choose` sequences. Additional braces are explicitly added around these latter two sequences when they appear in the argument block of a backslash construction, primarily to simplify the formal grammar of the next pass. This pass, like the first, is written as a recursive-descent parser in *Perl*.

4.2.5 Parsing a Formal Grammar

The third pass consists of parsing an attribute grammar for TEX mathematical expressions. The parser is a shift-reduce implementation using *flex* and *bison* with C++ code, so that the grammar is context-free and LALR(1). The language of mathematics is, unfortunately, neither context-free nor LALR(1), so that the abstract syntax tree produced by the parser contains syntactic as well as semantic ambiguities, to be addressed in subsequent passes. The lexical analyzer recognizes over 300 backslash sequences which are collected into token classes (Greek letters, relation operators, etc.) by the lowest-level rules of the grammar.

4.2.6 Semantic Passes

Several passes are made to modify the abstract syntax tree into a valid, disambiguated expression tree. Some passes are syntactic in nature to address the 1-token-lookahead limitation of the grammar, for instance handling prime and conjugate markings embedded in exponents. Other passes are semantic in nature and address the context-free limitation of the grammar, such as disassembling $a(x)$ function-like constructions whose left symbol is not really a function. One pass identifies the integrating variable of an integral and removes its differential from the integrand (it presently does not handle nested integrals, as none appear in the source text). These passes are written in C++ and are linked with the *bison*-based parser, so that a sequence of them may be invoked from the parser for each `$$` formula block.

4.2.7 Emitting LISP

A penultimate pass is done to print a LISP representation of the resulting expression tree. In addition to arithmetic and relational operations native to LISP, the expression tree employs such constructions as `(integrate` *`integrand (variable lower-limit upper-limit)`*`)` and an outermost construction `(stmt` *`formula-num relation relations...`*`)` to represent complete formula rules from the *Table of Integrals...*. In the `stmt` construction, the first relation represents the primary formula, and optional subsequent relations represent conditions required in order for the first relation to hold. Syntax errors encountered in the *bison* parser as well as semantic errors discovered in the semantic passes are flagged by the construct `(parse_error` *`line-num`*`)`. This pass, like the semantic passes, is written in C++ and is invoked from the *bison* parser for each `$$` formula block.

4.2.8 LISP to CAS

We read the LISP data into a LISP-based computer algebra system. This allows us to check the formulas for various syntactic consistencies. In our particular case, we algebraically simplified the formulas, and converted them to TEX for printing and comparison.

5 Results

The performance of the recognition system on its first input set was fairly good. It was able to parse 154 of 210 `$$` blocks without error (i.e. 73% of all such blocks). Of the 56 erroneous blocks, 29 are series and product formulas which use ellipsis notation with `\ldots` or `\cdots`. We expect that ellipsis patterns, while possible to process, can be more easily handled on a case-by-case basis by explicit human intervention. The remaining half of all errors are more conventional and remediable syntax errors, including unexpected punctuation or text comments embedded around formulas, as well as unexpected bracing not handled by the brace-stripping pass.

The error rate quoted may be somewhat low because we may have allowed suspicious or ambiguous expressions to pass through, and be re-typeset. It is plausible to strengthen the semantic pass dedicated to identifying constructions allowed by the parser whose meaning is not clear. We have written a version of a general context-free parser that will return multiple parses for ambiguous inputs, but have not used it for this task yet. For the integral table it is sometimes plausible to test the formulas (e.g. differentiating the anti-derivatives, or doing numerical spot checks on the definite integrals). While such checks might help disambiguate some formulas, sometimes it requires substantial subtlety to confirm a formula. A spot check might reveal which of several interpretations is sensible; e.g. it is not obvious whether the $\mathbf{K}(k)$ belongs to the ln operator on the right side of this elliptic integral formula:

$$\int_0^{\frac{\pi}{2}} F(x,\, k)\mathrm{ctg}x\, dx = \frac{\pi}{4}\mathbf{K}(k') + \frac{1}{2}\ln k\mathbf{K}(k).$$

Despite its shortcomings, the recognition engine is able to convert such compound, multi-line formulas as:

$$\int_u^n E(x,\,k)\frac{dx}{\sqrt{(\sin^2 x-\sin^2 u)(\sin^2 v-\sin^2 x)}}=$$
$$\frac{1}{2\cos u\sin v}\mathbf{E}(k)\mathbf{K}\left(\sqrt{1-\frac{tg^2\,u}{tg^2\,v}}\right)+$$
$$+\frac{k^2\sin v}{2\cos u}\mathbf{K}\left(\sqrt{1-\frac{\sin^2 2u}{\sin^2 2v}}\right)$$
$$[k^2=1-\mathrm{ctg}^2\,u\mathrm{ctg}^2\,v].$$

from the original TEX form:

```
$$
\def\UUU{\quad {\int_{u}^{n}}E(x,\tsp k)
{dx\over\sqrt{(\sin^{2} x-\sin^{2} u)
(\sin^{2} v-\sin^{2} x)}}}
\def\UU{\hphantom{\UUU}}\displaylines{\UUU
={}{1\over2\cos u\sin v}\mbi{E}(k)\mbi{K}
\left(\sqrt{1-{tg^{2}\tsp u\over
tg^{2}\tsp v}}\right)+\hfill\cr
\UU{}\hphantom{{}={}}{}+{k^{2}\sin v\over2\cos u}
\mbi{K}\left(\sqrt{1-{\sin^{2}
2u\over\sin^{2} 2v}}\right)\hfill\cr
\hfill[k^{2}=1-\ctg ^{2}\tsp u\ctg^{2}\tsp v].
\qquad\cr}
$$
```

using the expanded intermediate form:

```
$$
  \quad  \int_{u}^{n} E(x,\thinspace k)
 {dx\over\sqrt{(\sin^{2} x-\sin^{2} u)(\sin^{2}
  v-\sin^{2} x)}}
=    {1\over2\cos u\sin v} \bold{E}(k)
\bold{K}\left(\sqrt{1- {tg^{2}\thinspace u\over
tg^{2}\thinspace v} }\right)+\hfill\quad
        + {k^{2}\sin v\over2\cos u}
\bold{K}\left(\sqrt{1- {\sin^{2}
2u\over\sin^{2} 2v} }\right)\hfill\quad
\hfill[k^{2}=1-\hbox{ctg} ^{2}\thinspace u
\hbox{ctg}^{2}\thinspace v].\qquad\quad
$$
```

into the LISP form:

```
(stmt () (== (integrate (* (userfunc E x k) (/ 1 (power
(* (- (power (sin x) 2) (power (sin u) 2)) (- (power
(sin v) 2) (power (sin x) 2))) (/ 1 2)))) (x u n )) (+
(* (* (/ 1 (* (* 2 (cos u)) (sin v))) (userfunc bold_E
k)) (userfunc bold_K (power (- 1 (/ (* (* t (power g
```

```
2)) u) (* (* t (power g 2)) v))) (/ 1 2)))) (* (/ (*
(power k 2) (sin v)) (* 2 (cos u))) (userfunc bold_K
(power (- 1 (/ (power (sin (* 2 u)) 2) (power (sin (* 2
v)) 2))) (/ 1 2)))))) (== (power k 2) (- 1 (* (power
(ctg u) 2) (power (ctg v) 2)))))
```

A closer examination shows that while this was correctly converted as written, there is a bug in the input: tg^2u was written as a product, and not as the intended tg^2u, the tangent squared. Furthermore the upper limit of the integral should have been v not n. The printed copy of the table is somewhat vague in this area.

The entire recognition process is reasonably fast. Processing our initial 33 kilobyte TEX source extracted from G+R containing 210 `$$` formula blocks, requires 13.6 seonds for the *Perl*-based passes and 0.3 seconds for the C++-based passes, on a contemporary personal computer[7]. The *Perl*-based passes, whose task is conceptually simpler than the C++-based passes, could probably gain an order-of-magnitude speedup from reimplementation in a compiled language.

6 More exhaustive testing

We worked around the built-in limits on quantities allowed in the G+R formula export, we extracted some 10,740 formulas. These were contained in the file `GRAD.DAT` on the CDROM for G+R). A slightly modified version of the recognizer announced 5906 errors and presumed success for the remaining 4,834. Some of the alleged successes were in fact really successful since they included some 170 formulas with unhandled derivative forms, as well as some semantically questionable forms.

Of the 5,906 reported errors[8], some 1878 are caused by unrecognized TEX control sequences or macro definitions that we have not yet considered, including matrix constructions, equation alignment sequences, and macros for many special function names. These were not encountered in our original test set "training", and therefore were not included in our grammar. An additional 804 errors were caused by `\hbox` constructions with unrecognized contents, including more special function names and embedded narrative comments. We believe that simply adding more special function names (and their more complicated super/sub-scripting) one at a time would allow the engine to recognize several thousand additional formulas out of this 5906. Other errors are due to formula forms not handled by the grammar, including some 300 ellipsis constructions, and forms with unexpected punctuation or bracing[9].

7 Conclusion

7.1 Concerning our Implementation

An automatic recognition engine has been presented with the goal of converting natural mathematics notation typeset in TEX into a disambiguated, linearized LISP form. The engine has proven to be effective in recognizing a large subset of the electronic reference *A Table of Integrals, Series, and Products* [10] consisting of integral and series formulas in the domain of real, scalar calculus. The engine demonstrates that recognition of TEX code is feasible and fast using multiple-pass parsing and semantic analysis techniques. We believe that a complete "semantic" re-typesetting of this work could proceed more rapidly based on the output from our program, although sensible checking would still be needed[10]

[7] Macintosh PowerBook 3400c, 240 MHz PowerPC 603e processor

[8] the stated error counts are in fact estimates which come from tallying parser error diagnostics, and may therefore be inaccurate due to cascading of errors

[9] We do not count the 300 ellipses as "unrecognized control sequences" since we recognize them – we just do not know what to do with them!

[10] In fact, subsequent to our project, Daniel Zwillinger and Academic Press have totally retypeset G+R and produced a new CDROM. Zwillinger recoded the mathematics to provide a semantically appropriate underpinning for the work, from which

Future work for the recognition engine would include going beyond G+R to other work, perhaps based on particular journal conventions, expanding the grammar and adding explicit semantic processing as needed. We would like to expand support for various additional notations even within G+R. For example, absolute-value brackets, which are currently handled only around the simplest expressions, cause difficulty because of the ambiguity inherent in using the same vertical bar symbol for both sides of the character grouping. We would also like to add support for complicated function forms involving subscripts and superscripts, for instance $P_\nu^\mu(z)$ for associated Legendre functions. Such special functions arguably form the most useful heart of a table of integrals. Other desirable expansions to the system include support for derivatives, better handling of exponent semantics (f^{-1} for inverse function, $f^{(n)}$ for n^{th} derivative), support for symbol accents (bars, tildes, etc.), and a full complement of vector operations.

One lesson learned in this project is that the high-level structure of a document, even for pieces as small as an integral formula statement with conditional relations, may be buried under irregular punctuation and annotations which are not well-modeled by a single grammar. A future change to the engine might use an initial pass to separate the input into text and equation subcomponents based on spaces and punctuation, then send each component to a formula parser that need not worry about contextual punctuation. Trying to parse an equation using several domain-specific parsers may prove easier than constructing a single universal parser, and this may be the key to wider use of TEX as a viable communication form when others are not available, or even when others are available but not fully supported.

7.2 Concerning TEX for CAS Communication

Our view is that TEX in context and with suitable "understanding" programs, is a historically important encoding for considerable bodies of mathematics, and continues to be used for new publications. It can also be used to encode in a "naive" positional form some material initially parsed from 2-D into TEX.)

If we are to make semantic sense of this body of material, we must deal with the consequences: parsing TEX. In fact, we may thereby be forced to face the reality of dealing with a fuller notation than is routinely available in any computer algebra system. Certainly, faced with communication problems between humans and computers, we cannot just require all pure and applied mathematicians to use the subset of mathematics present in CAS of today. Methods for encoding extensions, then transmitting a unique name for the notational and semantic extension as part of the communication, seem plausible. OpenMath with its content dictionaries has such a design, and might provide a more neutral encoding target for G+R or other documents.

References

[1] T. V. Raman, "Audio System for Technical Readings," `http://simon.cs.cornell.edu/Info/People/raman`

[2] A. Aho, R. Sethi, J. Ullman. *Compilers: Principles, Techniques, Tools,* Addison Wesley.

[3] Dorothea Blostein and Ann Grbavec. Recognition of Mathematical Notation. Chapter 22 in P.S.P. Wang and H. Bunke (ed) *Handbook on Optical Character Recognition and Document Image Analysis,* World Scientific Publ. Co, 1996.

[4] G.F. Carrier, M. Krook, and C.E. Pearson, *Functions of a Complex Variable*, McGraw Hill, 1966 (first printing).

[5] Courant and Hilbert, *Methods of Mathematical Physics*, volume 1. Wiley, (1953).

[6] Digital Library discussion (email) (Univ. Illinois). 1996.

either TEX or a computer algebra system style output could then be generated. This direction of conversion is, in our opinion, the correct way to tackle this problem.

[7] EBT Dynatext `http://www.inso.com/frames/consumer/dynatext/index.htm`

[8] Richard J. Fateman and Taku Tokuyasu. "Progress in recognizing typeset mathematics," *Proceedings SPIE* Document Recognition III Vol. 2660, Jan. 1996. 37—50.

[9] R. Fateman and T. Einwohner.
`http://http.cs.berkeley.edu/~fateman/htest.html`

[10] I. S. Gradshteyn and M. Ryzhik, edited by Alan Jeffrey and Daniel Zwillinger, *Table of Integrals, Series, and Products,* Academic Press (5th edition), (also CDROM) 1996. Academic Press (6th edition), 2000. xlv+1163 pages.

[11] IBM. Techexplorer hypermedia browser.
`http://www.ics.raleigh.ibm.com/ics/techug.htm`

[12] W. F. Hammond. Setting Mathematics with SGML `http://math.albany.edu:8800/hm/sgml/about.html`

[13] P. Iglio and G. Attardi, "Software Components for Computer Algebra," in *Proc. ISSAC 1998.* ACM, Rostock, Germany, 62–69. `http://www.openmath.org/`

[14] D. E. Knuth, *The TEXbook*, Addison-Wesley, 1984.

[15] Mathtype, Design Science Inc,
`http://www.mathtype.com`

[16] Norbert Kajler and Neil Soffer. "A survey of user interfaces for Computer algebra systems," J. Symbolic Computation.

[17] A. Prudnikov, P. Brichkov, O. Marichev. Integrals and Series (Moscow) 1983.

[18] Neil M. Soiffer, *The Design of a User Interface for Computer Algebra Systems.* Ph.D thesis, EECS Dept., Univ. of Calif, Berkeley, April, 1991. See Also "Mathematical Typesetting in Mathematica," ISSAC-95 140–149.

[19] The QED project
`ftp://ftp.mcs.anl.gov/pub/qed/manifesto`

[20] TCI: Scientific Workplace
`http://www.thomson.com/brookscole/SWCAT_96/swp_2.0_swcat.html`

[21] M. Yang and R. Fateman, "Extracting Mathematical Expressions From Postscript Documents," Proc. ISSAC 2004 (to appear).

[22] Y. Zhao, H. Sugiura, T. Torii and T. Sakurai, "Knowledge-based method for mathematical notations understanding," *Trans of Inf. Proc. Soc. of Japan,* vol 35 no 11 (Nov. 1994) 2366–2381.

[23] Daniel Zwillinger, personal communication, August, 1998.

Spaces, the final frontier

Often a space is absent between two symbols multiplied together. Sometimes a space means nothing, as with spaces around "+". Oddly enough, sometimes a space means "DIVIDE" as a result of an an interesting ambiguity in the use of the "/". a/bc written as `$a / b c$` which might be transcribed in TEX as `${a \over b c}$` is displayed as $\frac{a}{bc}$. This is different from the usual programming language interpretation

in which we would first substitute a multiplication operator for the space between b and c making it `a / b * c` and then providing the computational semantics of $\frac{a}{b} \cdot c$. The argument for the TEX version is that no human mathematicican would use $1/2\pi$ to mean $\pi/2$. Therefore the former expression must mean $1/(2\pi)$. Does this happen? Sure. Here's one from Courant and Hilbert vol 1 [5] where p. 349 has $u_n = \sqrt{2/\pi\rho_0}\sin nx$ by which is meant

$$u_n = \sqrt{\frac{2}{\pi \cdot \rho_0}}\sin(n \cdot x).$$

Perhaps because of the opportunity for misunderstandings, the occurrences of "/" in typeset mathematics are not nearly as common as they are in computer programs, where the linear format imposed on most input (files of characters) makes it popular. It seems that for older printed work or reference texts, the horizontal divide bar is preferred. Slashes are used when space is tight (common in journals today), or a division must be denoted in a cramped space: an exponent, an in-line expression, a formula's side-conditions. A typical example of a use in display form:

$$\ldots \left(\frac{x-y}{z}\right)^{v/2} \ldots$$

in [17] p. 619 3.2.4.2. In fact in this large table, the denominators following the "/" are almost always single digit numbers, with rare occurrences of single symbols e.g. $1/r$. When the denominator is larger there might be parentheses: e.g. on p 337, $b^{(1-2\alpha)/(2r)}$. An alternative notation that is also used: $a(b+c)^{-1}$ makes the slash entirely dispensable when the denominator is more than a few characters. Though TEX doesn't seem to have shallow slashes, these are sometimes used for small rational fractions I will call semi-slash expressions. Thus instead of $1/2$ we have something like $\frac{1}{2}$ where the numerator 1 is shrunk and slid up a bit and the 2 is similarly moved down a bit. While $1/2\pi$ may be ambiguous, $\frac{1}{2}\pi$ is definitely $\pi/2$. (We could facetiously suggest it is another way of writing $2^{-\pi}$).

A carefully written complex variables text [4] uses the semi-slash as indicated, yet (p. 318) uses $\cos(\pi y/2b)$ to mean $\cos(\pi y/(2b))$. The subtlety catches these authors on p. 285 where $w = yexp[-\int \frac{1}{2}a_1(z)dz]$ appears when it should be $w = yexp[-\int 1/(2a_1(z))dz]$.

Of course slashes are used in dy/dx or even d/dx as operators, additionally complicating the lives of simple-minded parsers.

Spaces can also be used as separators between formulas, or between equation numbers, formulas, side-conditions, notes, etc.