Parsing Mathematics Typeset in TEX

CS-282 Course Project
Prof. Richard Fateman
University of California, Berkeley

Eylon Caspi

December 18, 1997

Abstract

The recognition of mathematics notation by a computer is made difficult by the two-dimensional nature of the parsing problem as well as by the richness and ambiguity of the notation. Parsing mathematics typeset in TEX constitutes a simplified, idealized 2D recognition problem, allowing the recognition engine to concentrate more on semantic understanding. Choosing TEX as an input form for mathematics is immediately desirable for document recognition because of the availability of many published works in TEX form. It is also desirable as a linearized, intermediate form emitted by a mathematically-oriented graphical user interface, as in Tech-Explorer [6]. A multi-pass mathematics recognition engine is described, designed with the intent of transcribing formulas from the electronic reference *A Table of Integrals, Series, and Products* [5] into LISP statements suitable for a computer algebra system. The engine is currently capable of transcribing 154 of 210 integral and summation formulas in the domain of real, scalar calculus.

1 Introduction

1.1 Parsing Mathematics Notation

While optical character recognition (OCR) of text documents is commonplace today, automatic recognition of mathematics notation enjoys limited success. The recognition of mathematics is difficult for a number of reasons. For one, it is a two-dimensional problem involving a large set of symbols. Mathematics notation is very rich, encompassing many domains with specialized notation, for instance, differential and integral calculus, matrix mathematics, set theory, bra-ket notation for quantum mechanics, etc. Within any one domain of notation there exist subtler ambiguities. Consider that with elementary function notation, commonly learned in high school, it is not even obvious whether $a(b)$ is a multiplication or a function. While ambiguities such as this one are easily disambiguated by knowing whether the symbol a represents a variable or a function, a symbol dictionary may not always be available. Many ambiguities in mathematics notation are more sinister and require an explicit declaration of notation convention. In fact, one of the most useful aspects of mathematics notation, and one which makes it impossible to capture formally at any given time, is that it is extensible.

Mathematics notation is (arguably) the most natural and desirable notation with which a human and a computer can communicate. Computer algebra systems (CAS) such as *Mathematica* and *Maple* currently require the user to use a linearized syntax based on ASCII characters. These languages, while easily digested by a computer, tend to generate large expressions that are difficult for a human to learn, type, and visually inspect. Compare, for instance, the notations for an expectation-value computation in natural form:

$$\frac{\int_{-\infty}^{\infty} x f(x) dx}{\int_{-\infty}^{\infty} f(x) dx}$$

and in *Mathematica* form:

```
Integrate[ x*f[x], {x,-Infinity,Infinity} ] / Integrate[ f[x],
{x,-Infinity,Infinity} ]
```

The latter form has the additional visual complication of line wrapping. Most humans would have a far easier time writing the expression on an electronic tablet or entering it in a visual mode on the screen.

The concern over user interfaces might be aleviated if a user were only ever required to learn one linearized syntax. Unfortunately, most computer algebra systems have their own unique syntaxes and cannot directly communicate with each other. Standardization efforts for a universal mathematics specification language, such as *OpenMath* [11] and mathematical SGML [10] are slow to mature, due in part to the richness and extensibility of the language of mathematics.

Natural mathematics notation is also the only form to consider in automated document recognition. There is a wealth of publications — journals, textbooks, reference tables — containing mathematics that one may wish to enter into a computer algebra system. The problem of optical character recognition for mathematics is a formidable one, requiring 2-D parsing techniques and the recognition of a very large character set. While the automated recognition of a basic algebra text, including fractions, powers, and basic set notation, seems relatively simple, the recognition of a typical AMS (American Mathematics Society) journal paper is a far harder task.

1.2 TEX as a 2D OCR Problem

Consider TEX as a language for specifying natural mathematics notation. With the addition of such popular modules as AMS-TEX, TEX becomes a most extensive engine for specifying the look of mathematical text using standardized syntax. In fact, AMS-TEX enjoys a special type of "completeness" in that it is the prescribed system with which to typeset mathematical journal papers today. As such, there is a wealth of publications available in TEX form which we may consider for automatic recognition.

Recognizing mathematics from TEX code amounts to a simplified, idealized 2D OCR problem. TEX represents a noise-free character recognition operation. The linear structure of TEX code also represents a nearly-complete character placement operation which makes character relationships apparent. This is especially true for vertical relationships, for instance, the unambiguous association of summation limits with a sigma character:

$$\sum_{n=0}^{N} x^n,$$

produced by the code: `\sum_{n=0}^N x^n`. With these simplifications to the OCR problem, a TEX-based mathematics recognition engine can concentrate its effort on the semantic understanding of mathematics notation, relying on readily-available text-processing tools such as *lex* and *yacc* to parse its input.

TEX also has merit as an intermediate language associated with a mathematical graphical user interface. Commercial tools such as IBM *TechExplorer* [6], TCI *Scientific Workplace* [13] and MathSoft *MathType* [?] exist today which allow the user to build mathematical formulae 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 (ex. Macintosh PICT) 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 formulae. 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 *Maple* and *Mathematica*). The same job could be done by a modular TEX-based recognition engine.

1.3 Formulation of a Class Project

In the scope of a graduate course on symbolic computer algebra, we tackled the problem of recognizing mathematics from TEX. Specifically, we set out to transcribe formulae from Gradshtein and Rhyzik's *Table of Integrals, Series, and Products* [5], an electronic mathematical reference on CD-ROM capable of displaying, navigating, and exporting its content of SGML, TEX, and bitmapped illustrations. While this export format is suitable for copying and pasting into publications, it not readily usable in a computer algebra system. Such an extensive collection of formulae would certainly be useful for table-based integration and summation algorithms. An immediate, industrial solution to converting the formulae into an appropriate computer language today would necessarily involve manual effort to retranscribe the formulae. We would like to automate the conversion process as much as possible.

Ideally, a computer program would convert the bulk of formulae into a disambiguated output language, leaving only the most difficult 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 algera systems. We chose LISP as the target language for its simplicity and wide use. Several computer algebra systems in use today, for instance Macsyma, are written in LISP and/or allow the user to interact with the system directly in LISP.

Previous efforts at U.C.Berkeley to convert this electronic *Table of Integrals* ... were based on a recursive-descent parser in LISP and enjoyed limited success [8]. The parser was able to convert selections of the source into an intermediate form based on rectilinear grouping (h-boxes, v-boxes). Work remained to be done, however, in the semantic understanding of the grouped expressions. A major hindrance suffered by this project was in the slow development and modification of the mostly hand-coded parser.

The mathematics recognition engine developped for this project is a multipass system based on rapid development tools such as *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 *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 on 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 emits a LISP representation of the syntax tree.

The initial performance of the recognition engine is encouraging. It is able to successfully convert 154 of 210 formulae from from those ten of the *Table of Integrals*'s eighteen chapters which deal with scalar integrals, summations, and products. These formulae 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.

2 Ambiguities in Advanced Calculus Notation

The input domain for this project is the mathematical language for high-school level advanced calculus — integration and series of real, scalar-valued variables and functions. This domain has a variety of ambiguities arising from operator notation, confusion between function and variable symbols, and other issues.

2.1 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. Also, For a variable a, a' typically refers to a related second variable, whereas for a function f, f' typically refers to the derivative of f. Similarly, for a variable a, a^{-1} is a reciprocal, whereas for a function f, f^{-1} is typically the inverse function.

Ambiguities of this form are easily overcome if one can determine the type of 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.

2.2 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:

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

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:

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

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 an operator is subjective to each user and sensitive to the domain of computation. In such cases, a recognition engine cannot, in good faith, make up its mind about operator precedence without consulting some user preferences or making an interactive inquiry.

2.3 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 standalone 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 somewhat unconventional form for a spherical-coordinate volume integral:

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

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 begining 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 splitting a differential or function into a product with the leftmost character belonging to an operator. For instance, b being a variable necessitates the transformation:

$$\sin ab(c)de \rightarrow \sin(ab)(cde)$$

3 The Recognition Engine

A prototype mathematics notation recognition system has been constructed with the intent of extracting formulas from the electronic reference, *A Table of Integrals, Series, and Products* [5], a commercial publication of Academic Press. The reference is essentially an electronically-typeset, computer-navigable book on CD-ROM, with no inherent interface to any computer algebra system. The reference can, however, export its content in a combination of SGML code for narrative text and TEX plus $\mathcal{AMS}$-TEX code for mathematics. Our recognition engine attempts to convert the TEX portion of the reference into a disambiguated LISP form.

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

Within the ten chapters considered, we also had to curtail what content the recognition engine be made would handle. Ideally, we would like to transcribe the full text of the *Table of Integrals ...* , retaining for context even the non-mathematical narrative. The full text, however, can only be exported in SGML form or a text-only form devoid of any formatting, limiting its appropriateness and usefulness as context. In addition, the export feature suffers from a number of problems such as omitting ends of sections and merging lines so that TEX comments symbols swallow more than their original comment line. We decided, therefore, to extract and recognize only the stand-alone formulas embedded as `$$` TEX blocks in the SGML code. Two hundred ten (210) such formulas appear in those chapters of the reference which we handle.

The `$$` formula blocks comprise the main listing of formulae 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 causes some otherwise needless syntax errors.

The TEX code appearing in the formula blocks consists of plain-TEX math code and several 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 AMS-TEX `\align` macro.

3.2 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.

3.2.1 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 AMS-TEX style files in this pass, as they would only complicate parsing by expanding 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.

3.2.2 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. 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 explicitely 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*.

3.2.3 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.

3.2.4 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 the 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* parser, so that a sequence of them may be invoked from the parser for each `$$` formula block.

3.2.5 Emitting LISP

A final pass is done to emit 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.

3.3 Results

The performance of the recognition system on its present input set is fairly good. It is 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 formulae which use ellipsis notation with `\ldots` or `\cdots`. Ellipsis patterns are quite difficult for any automated recognition system to handle and typically require explicit human intervention. The remaining half of all errors are more conventional and remediable syntax errors, including unexpected punctuation or text comments embedded around formulae, as well as unexpected bracing not handled by the brace-stripping pass.

The error rate quoted herein may be artificially low because there is presently no flagging of suspicious or ambiguous expressions. In the future, we would like to have a semantic pass dedicated to identifying constructions allowed by the parser whose meaning is not clear. For instance, 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 formulae as:

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

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)))))
```

The entire recognition process is reasonably fast. Processing the 33 kilobyte TEX source extracted from the *Table of Integrals ...* , 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 [1]. 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.

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

4 Conclusion

An automatic recognition engine has been presented which converts natural mathematics notation typeset in TeX into a disambiguated, linearized LISP form. The engine has proven to be effective in recognizing a subset of the electronic reference *A Table of Integrals, Series, and Products* [5] 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.

Future work for the recognition engine would include expanding the grammar and adding more semantic passes. We would like to expand support for absolute-value symbols, which are presently handled only around the simplest expressions, primarily 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, squiggles, 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 conditioning 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. Such techniques should make the recognition system more robust and capable of recognizing more of the *Table of Integrals ...* as well as recognizing mathematical documents in general.

References

[1] Aster thesis

[2] A, Aho, Sethi, 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] Digital Library discussion (email) (Univ. Illinois). 1996.

[5] Gradshteyn and Rhyzik. *Table of Integrals, Series, and Products* Academic Press (5th edition), (also CDROM) 1996.

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

[7] Donald E. Knuth: The TeX book. Addison Wesley, 1970.

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

[11] openmath http://www.can.nl/

[12] N. Soiffer, Mathscribe

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

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