2-D Display of Incomplete Mathematical Expressions

Richard Fateman
Computer Science Division, EECS Department
University of California at Berkeley

March 10, 2006

Abstract

Entering mathematical expressions into a computer can be done in a variety of ways. One favorite is a constrained input in which the human is (in more or less subtle ways) forced to enter syntactically correct formulas through the use of prompts, templates, or selections from menus. Done well, the user does not feel so much constrained as assisted. To be comfortable, the system must anticipate the nature of expressions being entered and provide sufficient flexibility to allow what appears to be natural input.

An alternative is free-form input, where the user can type or (handwrite) or speak essentially anything, and the computer system must try to make sense of it or signal errors.

In supporting free-form input, especially by speech or handwriting, it is logical for us to assume that some of the symbols will be misrecognized or missed. Even with keyboard input, users will often enter a syntactically erroneous or incomplete expression. We would like to preserve partial expressions or some good parts, and we should supply assistance in repairing the errors.

For this purpose we would like to display the expression "as best we can" in the nature of a constrained input form or template. With this displayed for the user, we can welcome corrections to this best guess. To do this nicely we show how to provide a framework for display of *broken expressions*. Tools akin to error-correction in parsers seem most relevant. However, in our experience only a small fragment of "real" mathematics notation is unambiguous and context free, typically that used for programming languages such as Fortran. For real math, we don't even know the full grammar, and probably do not wish to know it. For these reasons the usual error correction methods may not be quite on target.

In summary, our goal is to say, "the expression you entered can be typeset as shown. But before going much further, please fix it up so there are no loose ends".

1 Introduction

Given an "incorrect" expression, say something like `a+b/+c`, we want to display it and show what is broken. One of several possibilities is

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

Presented with such a display, a user of this interface could

- Click on the box and type, write, or speak the missing part.
- Reject the display and return to the string representation to modify it, perhaps resulting in a structurally different form. (For example here, delete the `/` to produce `a+b+c`.
- Reject the display and ask for an alternative correction as a 2-d display.
- Insert the missing part(s), but also edit other components in the display.

Of course an expression this small could easily be re-entered from scratch, so any tools we suggest must work well for much larger expressions. We show how this might be done in the following sections.

2 Examples

Assume that the expression is supposed to follow a commonplace grammar for mathematics. This typically differs from the ordinary programming language grammar in allowing implicit multiplication, far more numerous operators, an ambiguous precedence hierarchy, and a larger character set. For the computer science professional, what we are doing is making corrections on the expressions that can be typeset in TEX rather than the formulas that can be understood by Fortran. TEX while it incorporates a great many subtleties in typography, has only modest restrictions on the syntax of what you can attempt to typeset. Most of these have to do with balancing brackets {}.

A reasonable person might think that our example from the introduction `a+b/+c`, is an erroneous utterance that is, however, close to one of the following correct strings (where `E` is some unknown expression): `a+b/E+c` [add the token E] or `a+b/c` [remove a token +] or `a+b+c` [remove a token /]. There are more corrections possible, but they are longer.

There are other utterances which are *not* erroneous, even though they are open to different interpretations. Consider `a+cos x`. Perhaps this means $a + (\cos) \times x$ or $a + \cos(x)$. We might think that the latter is more plausible, but what about `a+f x`? An invisible times ($\times$) can be supposed between `f` and `x`, or it can be a function application $a + f(x)$. In fact there are no “missing operator” errors possible in our loose grammar because we can always hypothesize that two adjacent operands are multiplied.

Consider unbalanced parentheses such as `)a (b(`. In a pre-processing step we add enough parentheses front and back to make this into `()a(b())` before proceeding, so we never encounter this issue on the fly.

We also must deal with incorrect suffix operations: `a ! ^ b` is probably acceptable as a form $a!^b$ but `a ^ ! b` is hard to figure out. It could be changed to $a^{▮}!b$ or $a^{▮!b}$.

We suspect that extensions to our program can be made easily to accomodate other variations on the typical grammar of mathematics; most hazards are exposed by { prefix, infix binary, suffix} operations, and “invisible” multiplication. Note that we do not pretend to do a full parse and in particular do not necesarily disambiguate expressions.

2.1 Multiple errors

Our objective is to keep stumbling forward, using as much material as has been supplied. The key to termination of the parsing is the end of the input material, not the construction of a complete expression. In our grammar, where adjacent expressions always possible (as if multiplied together), any sequence of legal expressions is also a legal expression.

2.2 Deletion versus insertion

Our current system always inserts material, but only enough to complete a pending syntactically incomplete component. This does not always lead to the “right” correction which requires mind-reading. It does not always lead to a minimal-size correction either, but it always advances the parse. Since it absorbs tokens at each correction it cannot lead to infinite streams of corrections. The basis for construction is a recursive descent parser with a few subtleties involved in choosing the location of insertions. A Lisp program of some 120 lines (63 non-comment lines) can make corrections of this form: $(a'b+'\sin)(b+/)$ to $(a'b+▮'\sin▮)(b+▮/▮)$.

The program can be found attached to the TEX source of this paper, or can be obtained from the author. I expect that the size of the program can be reduced by 1/3 by removing some error checks. The program to insert balanced parentheses is 14 of the 63 lines.

3 Incorrect forms may be correct in some context

Ordinarily we might view $(a+\ |b+)*$ as nonsense because it appears to have binary operators with missing operands. But it is a perfectly valid form in the notation common for describing sets related to finite-state grammars where it could be a "regular expression" over an alphabet including sets of strings a and b. A more careful typesetter would use superscripts operators like this $(a^{+}|b^{+})^{*}$, but we have seen it typeset with less care; and with a handwriting program the distinctions might be difficult to discern. Our point is that our program might be following context related rules that would wrongly offer the correction template $(a+\blacksquare|b+\blacksquare)*\blacksquare$.

Many similar examples can be understood by defining separate contexts, and relying on some external activity to make sure that they are not unsuitably mixed. A suitable grammar for regular expressions would have unary suffix superscript operators $\{*, +\}$ for Kleene closure and transitive closure respectively.

The grammar we have incorporated in our sample implementation has suffix operators, but only `prime, hat, bar !`. We could easily add to the list of operators, especially considering that we have no investment in their semantics.

If we were committed to require correct semantics, we would have to follow a path comparable to that of the OpenMath project [8]. This group has initiated the development of "content dictionaries" or CDs, attempting to relate a textual and syntactic representation of mathematics with a simultaneous semantic interpretation. The OpenMath approach for semantic representation is demonstrated by an example Java applet JOME [5], which asks you to type an expression (or select from templates) in order to see their equivalent displayed or encoded in OpenMath with respect to a common initial set of OpenMath CDs. JOME treates syntactically incomplete or erroneous expressions in somewhat the same way we have illustrated here, namely correcting by inserting missing operands when useful. Because JOME is tied to a rigid grammar and requires unambiguous content, it must make some choices where we are not forced to do so. Thus typing `sin x` produces $\sin *x$. Typing `sin(` produces a display sin([?]) where [?] stands for our box. Thus it is impossible to insert unmatched left parentheses in JOME. Our algorithms for inserting boxes differ though. In JOME `a+)b` becomes $a+b$; our program produces $(a+\blacksquare)b$.

A more interesting approach is that given by Tapia's JMathNotes[10], which is a handwriting-based math editor. In this system, after each stroke, the whole canvas is re-examined to see what it might say. There is no compulsion to have a complete expression, and when converted to (say) TEX it may be quite ill-formed. In our view the JMathNotes model is excessively free-form and hard to use.

4 A note on ambiguity

Note again that, especially in contrast to JOME, we don't care (at least not yet) if `sin x y` means $\sin(x)y$ or $\sin(yx)$ or for that matter, $\sin *x*y$. It is just the list $\sin xy$ to TEX. While to the computer scientist it may seem unnecessary and counterproductive to allow such lax notation, working mathematicians may be (perhaps with justification) attached to exactly the notations that make them comfortable. It is, after all, a mathematician who wrote:

> "When I use a word," Humpty Dumpty said in rather a scornful tone, "it means just what I choose it to mean—neither more nor less."
>
> "The question is," said Alice, "whether you CAN make words mean so many different things."
>
> "The question is," said Humpty Dumpty, "who is to be master—that's all."
>
> —Lewis Carroll, *Through the Looking Glass*, Chapter 6.

5 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley.

References

[1] A. Aho et al. Compiler design and construction.

[2] A. Bonadio. Theorist, MathEQ
http://www.livemath.com/matheq/

[3] S. Dooley, http://www.mathmlconference.org
/2002/presentations/dooleyxml/

[4] R. Fateman. Speaking math, Colorboxes, voice+hand. web page

[5] JOME. http://mainline.essi.fr/jome/jome-editor-en.html

[6] S. Lavirotte, Emath editor.
http://www-sop.inria.fr/cafe/Olivier.Arsac/emath/emath.html

[7] MacKichan. http://www.mackichan.com/

[8] Open Math Society.
http://monet.nag.co.uk/cocoon/openmath//index.html

[9] N. Kajler, N. Soiffer. "A Survey of User Interfaces for Computer Algebra Systems." J. Symb. Comp. vol 25 no 2 Feb 1998. 127-159

[10] E. Tapia, JMathNotes (free software)