Toward math-literate computers

Scott MacLean* George Labahn† Edward Lank‡ Mirette Marzouk§ David Tausky¶

University of Waterloo

ABSTRACT

Sketch recognizers are an important part of the design of natural interfaces for many domains. However, current recognition technology is generally quite crude and difficult to work with. This paper explores Blostein's concept of "literate" systems [1] in the context of recognizing handwritten mathematics. We describe some of the difficulties arising in complex sketch recognition domains, and offer some ideas for handling these difficulties that we have found useful in practice. Our suggestions focus on problems related to managing ambiguity, evaluating system performance, and obtaining data for training and testing recognizers.

Index Terms: I.5.5 [Pattern Recognition]: Implementation—Interactive systems

1 INTRODUCTION

New media encompasses a wide range of technologies and systems facilitating interactive, computer-mediated communication. Traditionally, such communication has been solely between users, with computer systems acting as messengers (e.g., e-mail), or as common storage and access points (e.g., collaborative hypertext, wikis). But, over time, computers have assumed a more and more prominent role in online communication, to the extent that users now entrust them with such tasks as selecting which messages to display (e.g., Google News, Facebook feeds).

In these systems, computers are passive observers of user interaction. There are many domains, though, which could benefit from the active participation of computer programs in human discourse. Given appropriate instructions, the brute efficiency of computer hardware can often provide insights that would otherwise be tedious or even impossible to discover. For example, consider the following scenarios:

1. While working out a chord progression, A musician queries the computer. Software analyzes the melody, reports what other composers have done in similar situations, and offers to play musical samples. The composer adapts a harmony that fits well with the next section of the piece.
2. A mathematician derives an equation, and instructs the computer to find parametrized solutions. Upon solving the equation, the software displays its results, but continues processing. It finds that the solution set satisfies the definition of a group, and reports this fact to the mathematician, who now has a large amount of new theory to apply to the problem.

For a computer to be capable of performing such tasks requires not only a significant amount of domain knowledge on the part of the software, but also a sophisticated and flexible means for communicating with the user. Blostein has characterized such software in the context of mathematics as "math-literate" [1].

These scenarios are both amenable to sketch-based interfaces. Over time, specialized domains, like music, mathematics, chemistry, etc. have naturally developed their own notations. While the written word, in most Western languages, is a simple linear representations of the sounds we make over time when speaking, these specialized notations denote more complex relationships (e.g., pitch, mathematical structure, chemical bonds), and are often depicted in two dimensions. Interpreting such notations is natural for people, but is difficult for computers, which organize information in only one dimension. To avoid discontinuity between users' thought processes and their interactions with a software system, it is important that computers understand the language in which people naturally express their ideas. In many cases, that language is a sketch.

Blostein pointed out that working with current math recognition systems is akin to working with someone quibbling over syntax, unable to decipher the meaning of an input. For example, a user may wish to use an integral transform to view a function from a different perspective, and sketch the appropriate formula. The software might report back, in effect, "did you draw 'x' here, or 'X'? I'm confused. Please try again." Such a primitive level of understanding is distracting, annoying, and disappointing from the perspective of a user who believes that a software program should help them to solve real-world problems more efficiently.

To be literate implies not only that one understands a domain, but also that one is able to meaningfully communicate and reason about it. The challenge of building literate software thus includes deep connections with problems in artificial intelligence and human-computer interaction. The present paper is an exploration of some of these problems. Based on our own experiences in developing MathBrush [6], a pen-based system for interactive mathematics, we offer some insight into specific difficulties that arise on the path to literate software for sketch-based notations, and point out some approaches that we have found useful for handling the complexity of these problems.

2 MODELING AMBIGUOUS DOMAINS

It is possible that the most meaningful communication many of us have had with our computers is through a compiler. By programming, we are able to communicate elaborate and complicated ideas to computers. This process is far from painless — programming is essentially a complex linearization process by which one translates abstract ideas about program behaviour into a long sequence of instructions meticulously indicating to

*e-mail: smaclean@uwaterloo.ca
†e-mail: glabahn@uwaterloo.ca
‡e-mail: lank@uwaterloo.ca
§e-mail: mmarzouk@uwaterloo.ca
¶e-mail: datausky@uwaterloo.ca

the computer exactly how to produce such behaviour. Still, despite its complexity, programming does facilitate a meaningful transfer of ideas from a person to a computer.

A compiler cannot be characterized as "literate". All of the interesting ideas originate from the programmer, while the compiler acts as the syntax-oriented quibbler described above. But, because of its familiarity, the idea of a programming language provides a convenient point from which to illustrate the complexity of sketch recognition.

There are three issues which contribute to this complexity:

1. *Two-dimensional notation.* When parsing the text of a program, the order in which the compiler should read the symbols is obvious: it should start with the first, then proceed to the second, and so on until it reaches the end. The text is linear and so has an implicit ordering. Two-dimensional sketches have no such ordering, so it is unclear which parts of the input form meaningful symbols or combine in meaningful ways.
2. *Syntactic ambiguity.* Unlike the known text of a program, handwritten input is inherently ambiguous. People often have trouble reading others' handwriting (or even their own). This ambiguity affects not only how individual symbols should be interpreted, but also how those symbols may combine to form larger expressions. For example, depending on whether the p symbol in Figure 1 is interpreted as lower- or upper-case, it may be most reasonable to interpret the whole expression as either A^p or AP.

Figure 1: A simple, ambiguous math expression.

3. *Semantic ambiguity.* The text of a computer program is either well-formed or not, depending upon whether its structure matches the grammar of the programming language. Conversely, natural languages are informally defined and often contain ambiguities. Even if the structure of a sketch is known with certainty, its meaning may be ambiguous. For example, without contextual information, it is impossible to say whether the expression $u(x+y)$ denotes a function application or a multiplication operation.
   When people communicate with each other, the intended interpretation of a sketch is made clear either by its context or by an explicit indication (often in words) of how it should be interpreted. It is common to introduce completely new notations and to augment existing notations with new meanings. (For example, textual instructions in musical scores, or the introduction of new notations in math papers.) It is very difficult to program a computer to handle such open-ended languages.

As with programming languages, formal grammars provide a natural framework in which to model the structure of a sketch domain. But traditional context-free grammars lack the expressiveness required to represent two-dimensional notations. As such, researchers have developed several extended parsing techniques. For math recognition, for example, researchers have proposed parsing techniques using tree transformations and operator precedence [11], LR parsing techniques [4], constraints on bounding box position [10], and many others.

These grammars model mathematical notation with varying degrees of generality and efficiency. What they lack, though, is a model of the ambiguity that is inherent in any process of recognizing complex two-dimensional sketches. Parse confidences are typically derived using ad-hoc scoring schemes, making it difficult for parsers to meaningfully compare alternative interpretations of the input. As a result, most math recognizers simply report the "best" interpretation of the user's input based on their particular scoring scheme. If that interpretation is not the one the user intended, they must erase their input and try again. An important exception to this rule is the Math Input Panel provided by Microsoft with Windows 7 [2] (shown in Figure 2), which provides users with a wide range of recognition alternatives. This approach is promising, though Microsoft's implementation offers perhaps too many unreasonable alternatives.

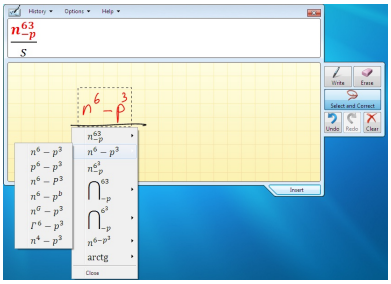

Figure 2: Microsoft's Math Input Panel

One way to deal with the pervasive ambiguity of sketch recognition is to incorporate a formal model of recognition ambiguity directly in the grammar formalism. For MathBrush, we developed a grammar formalism called *fuzzy relational context-free grammars* (fuzzy r-CFGs) that models the two-dimensional structure of handwritten mathematics as well as recognition ambiguities. The fuzzy r-CFG model provides a structured framework in which parsers can compute and reason about the validity of particular interpretations of the user's writing.

In the fuzzy r-CFG setting, the structure of mathematics is described using productions similar to those in traditional CFGs. In addition, each production is associated with a *relation* that describes the spatial arrangement of subexpressions. For example, the production `[ADDITION]` $\Rightarrow$ `[ADD_TERM]` $\rightarrow$ $+$ $\rightarrow$ `[ADD_TERM]` models a typical addition expression. The nonterminal `[ADD_TERM]` stands for a subexpression, and the $\rightarrow$ symbol indicates that the subexpressions are written from left to right. Similarly, `[FRACTION]` $\Rightarrow$ `[NUMER]` $\downarrow$ — $\downarrow$ `[DENOM]` models fraction notation.

The recognition process is viewed as the construction of a fuzzy set of parses of the user's writing. This view is applied recursively, so that each subset of the input is associated with a fuzzy set of interpretations. For example, consider parsing the addition production above. There are three tokens on the RHS, so the input must be partitioned into three subsets such that the $\rightarrow$ relation is satisfied. The grammar relations are also fuzzy, so different partitions have different grades of membership in $\rightarrow$.

There are typically many partitions with non-zero membership grade. For each one, the parser recursively constructs fuzzy sets of interpretations for the nonterminal `[ADD_TERM]` on the left- and right-most subsets. Note that the central sub-

set must be interpreted as the terminal symbol +, so symbol recognition results are sufficient to determine whether a parse exists there. Then the Cartesian product of these fuzzy sets is taken to find interpretations of `[ADDITION]`.

By this process, the recognizer obtains a large number of potential interpretations of the user's writing. Using the fuzzy ambiguity model, it systematically evaluates the goodness of each interpretation relative to other interpretations. In the end, the recognizer presents to the user a variety of recognition alternatives in order of recognition confidence. Examples of the parser's output are shown in Figure 3.

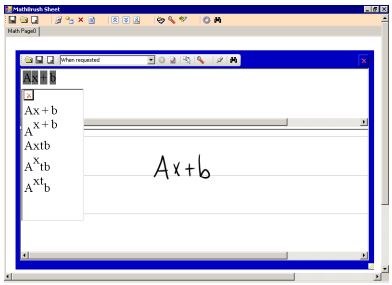

Figure 3: Output of the fuzzy r-CFG parser in MathBrush.

We intend MathBrush to be a platform for mathematical experimentation. As such, once an input expression has been recognized, users can manipulate it using context-sensitive computer algebra system (CAS) commands. The output of these operations is returned onto a math worksheet. Furthermore, the output can be edited with the pen and re-recognized, at which point new CAS commands may be applied, etc. We try to emulate the process of doing math with pen and paper, except that CAS software performs calculations that are too mundane or too large to do by hand.

Working with our correction interface, like Microsoft's, often amounts to syntactic quibbling. Still, it is quicker for users to select the intended interpretation of their writing than to rewrite an entire expression. In cases of semantic ambiguity or sloppy handwriting, working with the interface is no worse than having to explicitly disambiguate one's writing for a colleague.

An important caveat is that the colleague will remember, for example, that $u(x+y)$ represents a function application, so they will not need to ask again when looking at subsequent expressions. Our recognizer currently lacks such contextual knowledge and considers each input expression independently. We plan to add contextual features so that users may declare, for instance, $u : \mathbb{R} \to \mathbb{R}$.

3 Evaluating System Literacy

There are nearly as many schemes for evaluating math recognizer accuracy as there are math recognition systems. For example, Chan and Yeung measured the proportion of correctly-recognized symbols and operators [3], LaViola measured the proportion of correct subexpression groupings [7], Garain and Chaudhuri weighted recognition errors based on their depth in the parse tree [5], and so on. Each proposed measurement scheme depends significantly on the implementation details of the system it was used to evaluate. As such, it is extremely difficult to meaningfully compare the relative performance of math recognition systems.

Furthermore, these types of metrics essentially label each test expression as either "correct" or "incorrect", possibly including some indication of how severe the incorrectness is. Such measurements seem appropriate to recognizers intended for batch processing operations, in which there is little or no user interaction, and the recognizer simply looks at one expression after the other.

For applications designed for people to use in real-time, though, these measurements are unable to characterize what makes a recognition system more or less useful. Certainly, recognition accuracy is important, but so are other considerations. If an expression is not recognized correctly, how easy is it to tell the recognizer what the correct expression is? How easy is it to teach the recognizer new notations or new domain concepts? (E.g., teaching the MathBrush recognizer how to communicate with a CAS about Bessel functions.) Does the system adapt to one's writing style and habitual notations as a student or colleague would? In short, how literate is the system?

The answers to such questions are difficult to quantify. User studies are a good starting point. They provide some answers, and can suggest new questions that should be asked when evaluating system literacy. Some user studies on math recognizers have been published [6, 9], essentially concluding that correction and misrecognition are annoying, and errors that cannot be understood are worse. As recognition technology improves, the findings of similar studies will become more and more valuable.

As a preliminary metric of system literacy, we aim to measure user effort. One way to quantify this is as the amount of information the user must process in order to get the software to do what they want. For example, suppose a user draws Figure 4, meaning the expression $a^X + b$, but the first result returned by the recognizer is $a^x tb$. In our system, the user can correct the result to an addition, $a^x + b$, and correct the x to X. We can therefore count two corrections, or two "units of effort". An alternative measurement is to count the total number of results that the user sees while making those corrections. (So if the second candidate after x was y, and the third was X, for example, that single correction would count as two units.)

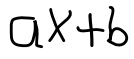

Figure 4: A hypothetical input expression.

This scheme has the side-effect of measuring recognition accuracy. If an input is recognized perfectly, then it counts as zero units of effort. Similarly, if it is recognized "almost correctly", it counts as fewer units than if the recognition is quite poor. But the primary goal of the metric is to quantify how good or bad the user experience is, not to count recognition errors.

We cannot claim that this approach in fact measures system literacy. But, by focusing on user-level concerns instead of software-level concerns, we believe that it provides a better indication of recognizer usability than the traditional percentage of correctly-recognized expressions, and that it offers a useful first step toward designing more sophisticated literacy metrics.

Furthermore, this effort-based metric is generally applicable to any recognition system, though it clearly is intended to be used with systems providing some correction or feedback mechanism. One could similarly navigate the recognition alternatives provided by Microsoft's recognizer, for instance, count the relevant operations, and obtain comparable measurements. Our evaluation scheme thus provides an abstract way to

compare the performance of varied recognition systems without direct reference to their implementation details.

4 Coping with limited resources

User studies with real users can be quite expensive, in terms of time (recruiting participants, managing sessions, organizing any qualititative results) and money (remunerating participants). It is infeasible to interview a wide range of users each time an incremental change to a recognizer is made. An investigation into more mechanical methods of evaluating system literacy is therefore worthwhile.

Our present approach is to replace the user by a software program. Given a handwritten math expression annotated with ground-truth, the program invokes the recognition system, examines the results, and makes corrections as a knowledgeable user would, counting how many units of effort were required to obtain the correct answer. This operation is easily batched into large scale testing, is fast, and does not require supervision.

Of course, there is no substitute for real users, but this automated "simulated user" method is valuable for testing incremental changes to our recognizer, and for maintaining a large suite of regression tests. Also, given an appropriate interface to other recognizers, it provides a way to mechanically compare the performance of different sytems.

The simulated user method requires a corpus of handwritten, ground-truthed expressions, which can also be time-consuming and expensive to obtain. In previous work, we developed some automatic tools for the creation of such corpora [8]. The product of this work, a ground-truthed corpus of over 5,000 hand-drawn mathematical expressions, is available online from http://www.cs.uwaterloo.ca/scg/mathbrush/corpus. Since it is relevant for training and evaluating recognition systems, we summarize the high-level process below.

1. *Generation.* Using a fuzzy r-CFG describing the syntax of mathematical expressions, generate random mathematical expressions by producing random grammar derivations. The expressions, represented as parse trees, are easily converted to LaTeX strings and rendered as images.
2. *Transcription.* We recruited students to transcribe math expressions. Using Tablet PCs running a custom collection program, participants were shown images of randomly-generated math expressions, which they transcribed as digital ink files.
3. *Annotation.* The structure of each transcribed expression was known a priori from the generation process. The problem of labelling expressions with ground-truth is therefore easier in this case than performing full recognition. We developed a program to generate ground truth automatically. The program matches the symbols and subexpression relationships appearing in the expected expression structure with digital ink strokes the observed transcription. To validate this automated approach, the entire corpus was also annotated manually.

Aside from the time required to develop the (reusable) programs, and to manually annotate the transcriptions, our corpus was obtained for only $200. (Each of the twenty participants were thanked with a $10 gift certificate for Tim Horton's.) Furthermore, our automatic annotation tool worked well enough that, in the future, manual annotation will be mostly unnecessary.

5 Conclusions

In order for computer-based tools to fit naturally into users' existing habits and workflows, it is important that they are, in Blostein's terminology, "literate". They should possess a flexible understanding of their domain, provide intelligent feedback, and generally enhance, rather than modify, users' existing practices. For many domains, including mathematics, this often means interpreting users' sketches and diagrams.

Sketch recognition is a difficult problem. Its two-dimensional nature and the ambiguity inherent in any recognition process pose significant complexity challenges. We have proposed a fuzzy r-CFG formalism which deals with these challenges by modeling the structures as well as the ambiguities apparent in hand-drawn sketches.

It is also challenging to evaluate sketch recognizers. Without standardized benchmarks and metrics, researchers generally use ad-hoc measurements particular to their own recognizer implementation. We have tried to step toward literacy-based metrics by proposing a measurement of recognizer effectiveness based on user effort.

User effort can be measured automatically by simulating user behaviour. Similarly, we have found automatic techniques useful for constructing large corpora of ground-truthed data. Such data is necessary for systematic training and evaluation of recognition systems.

MathBrush is not a math-literate system. But it is a system intended to match users' existing customs and expectations for "doing math". Math literacy provides a worthwhile goal, and serves as a useful guiding principle when making design decisions.

References

[1] D. Blostein. Math-literate computers. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Calculemus/MKM*, volume 5625 of *LNCS*, pages 2–13. Springer, 2009.

[2] R. Bogdan, G. Predovic, and B. Dresevic. Geometric parsing of mathematical expressions. U.S. Patent Application #20080253657, Filed 2007. Microsoft Corporation.

[3] K.-F. Chan and D.-Y. Yeung. Error detection, error correction and performance evaluation in on-line mathematical expression recognition. *Pattern Recognition*, 34(8):1671–1684, 2001.

[4] J. Fitzgerald, F. Geiselbrechtinger, and T. Kechadi. Mathpad: A fuzzy logic-based recognition system for handwritten mathematics. In *Proc. ICDAR*, pages 694–698, 2007.

[5] U. Garain and B. Chaudhuri. A corpus for ocr research on mathematical expressions. *IJDAR*, 7(4):241–259, 2005.

[6] G. Labahn, E. Lank, M. Marzouk, A. Bunt, S. MacLean, and D. Tausky. Mathbrush: A case study for interactive pen-based mathematics. In *Proc. Sketch-Based Interfaces and Modeling (SBIM)*, pages 142–150, 2008.

[7] J. J. Laviola, Jr. *Mathematical sketching: a new approach to creating and exploring dynamic illustrations*. PhD thesis, Providence, RI, USA, 2005. Adviser-Dam, Andries Van.

[8] S. MacLean, G. Labahn, E. Lank, M. Marzouk, and D. Tausky. Grammar-based techniques for creating ground-truthed sketch corpora. *IJDAR*, (to appear).

[9] T. O'Connell, C. Li, T. S. Miller, R. C. Zeleznik, and J. J. LaViola, Jr. A usability evaluation of algosketch: a pen-based application for mathematics. In *Proc. SBIM*, pages 149–157, 2009.

[10] D. Prusa and V. Hlavac. Mathematical formulae recognition using 2d grammars. In *Proc. ICDAR*, pages 849–853, 2007.

[11] R. Zanibbi, D. Blostein, and J. Cordy. Recognizing mathematical expressions using tree transformation. *PAMI*, 24(11):1455–1467, 2002.