

Automated recognition of handwritten mathematics

by

Scott MacLean

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

© Scott MacLean 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Most software programs that deal with mathematical objects require input expressions to be linearized using somewhat awkward and unfamiliar string-based syntax. It is natural to desire a method for inputting mathematics using the same two-dimensional syntax employed with pen and paper, and the increasing prevalence of pen- and touch-based interfaces causes this topic to be of practical as well as theoretical interest. Accurately recognizing two-dimensional mathematical notation is a difficult problem that requires not only theoretical advancement over the traditional theories of string-based languages, but also careful consideration of runtime efficiency, data organization, and other practical concerns that arise during system construction.

This thesis describes the math recognizer used in the MathBrush pen-math system. At a high level, the two-dimensional syntax of mathematical writing is formalized using a relational grammar. Rather than reporting a single recognition result, all recognizable interpretations of the input are simultaneously represented in a data structure called a parse forest. Individual interpretations may be extracted from the forest and reported one by one as the user requests them. These parsing techniques necessitate robust tree scoring functions, which themselves rely on several lower-level recognition processes for stroke grouping, symbol recognition, and spatial relation classification.

The thesis covers the recognition, parsing, and scoring aspects of the MathBrush recognizer, as well as the algorithms and assumptions necessary to combine those systems and formalisms together into a useful and efficient software system. The effectiveness of the resulting system is measured through two accuracy evaluations. One evaluation uses a novel metric based on user effort, while the other replicates the evaluation process of an international accuracy competition. The evaluations show that not only is the performance of the MathBrush recognizer improving over time, but it is also significantly more accurate than other academic recognition systems.

Acknowledgements

Thanks first of all to my supervisor George for the years of useful advice and guidance. It has been a privilege to work with you over the course of three degrees, and to learn from and take advantage of your professional experience has been a great help to me at many points along the way. Your ability to crash and confuse the recognizer never fails to impress (and, sometimes, depress) me.

I have enjoyed my years in the SCG lab, and I thank all of the students and professors for making my time as a student pleasant and productive. Special thanks to the folks involved with MathBrush: David, Mark, Ed, and especially Mirette for her collaboration and extensive help with debugging and testing the recognizer.

Thanks also to my parents for their constant encouragement. You taught me early on about the importance of persistence and hard work, without which I could not have finished this thesis. Knowing that you would fully support any decision I make has been a source of comfort and inspiration for many years.

Nobody can work all of the time, and I'm grateful to have such great friends, especially Nathan, Kim, Colin, Amy, and Karl. We are lucky to have all stayed in the same place for so long, though we become more geezerly each year we remain.

Finally, thanks to Jennifer for keeping me happy, reminding me that there is always much more to a person than their work, and making sure I still make dinner.

Dedication

For Sarah, who still laughs at my jokes.

Table of Contents

List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 MathBrush recognizer overview	2
1.2 Two viewpoints on math recognition	3
1.2.1 The abstract view	3
1.2.2 The system-oriented view	4
1.3 Ambiguity and interpretation	4
1.4 Contributions and thesis organization	6
2 Related work	8
2.1 Anderson	8
2.2 Zanibbi	9
2.3 Winkler, Farhner, and Lang	10
2.4 Miller and Viola	10
2.5 Garain and Chaudhuri	11
2.6 Laviola, Zeleznik et al.	12
2.7 Alvaro et al.	13
2.8 Awal et al.	13
2.9 Shi, Li, and Soong	14
3 Stroke grouping	15
3.1 Relevant input characteristics	16
3.2 Grouping algorithm	18
3.3 Evaluation	20

4	Symbol recognition	22
4.1	Recognition algorithms	22
4.1.1	Feature vector norm	23
4.1.2	Greedy elastic matching	23
4.1.3	Functional curve approximation	27
4.1.4	Hausdorff measure	28
4.2	Combining recognizers	29
4.3	Evaluation	31
5	Relation classification	33
5.1	Rule-based approach	34
5.1.1	Fuzzy sets review	35
5.1.2	Relation membership functions	35
5.2	Naive Bayesian model	36
5.3	Discriminative model	37
5.3.1	Discretization grid	38
5.3.2	Margin trees	38
5.4	Evaluation	43
6	Relational grammars	46
6.1	Preliminary grammar definition	46
6.1.1	Observables and partitions	47
6.1.2	Relations	47
6.1.3	Set of interpretations	48
6.1.4	Productions	48
6.2	Examples	49
6.3	Linkage parameters	49
6.3.1	Examples	51
6.4	Set of interpretations as a directed set	52
6.5	Semantic expression trees and textual output	53

7	Two-dimensional parsing	54
7.1	Shared parse forests	54
7.2	Shared parse forest construction	57
7.2.1	The ordering assumption and rectangular sets	57
7.2.2	Parsing algorithms	59
7.3	Extracting interpretations	64
7.3.1	The monotone assumption	65
7.3.2	Extracting elements of I_o^S	66
7.3.3	Handling user corrections	71
8	Scoring interpretations	73
8.1	Fuzzy logic-based scoring	73
8.2	Probabilistic scoring	74
8.2.1	Model structure	74
8.2.2	Expression variables	76
8.2.3	Symbol variables	76
8.2.4	Relation variables	77
8.2.5	Symbol-bag variables	77
8.2.6	Calculating the joint probability	78
9	System evaluation	80
9.1	Evaluation on Waterloo corpus	80
9.1.1	Correction count metric	80
9.1.2	Methodology and results	82
9.2	Evaluation on the CROHME corpora	85
10	Conclusion	88
10.1	MathBrush requirements	88
10.2	Low-level classifiers	89
10.3	Grammars and parsing	90
10.4	Scoring methods	91
	References	92

List of Figures

1.1	Syntactic ambiguity in simple expressions.	5
3.1	A simple input for the stroke grouper.	15
3.2	A more difficult input for the stroke grouper.	15
3.3	Distance between strokes is not an infallible grouping criterion.	16
3.4	Bounding box overlap is another good indicator of stroke groups.	17
3.5	Bounding box overlap is not always a reason to group strokes into symbols.	17
4.1	Many unmatched model points remain after matching every input point.	25
4.2	No more model points are available, but several input points remain to be matched.	25
4.3	These symbols may be more easily recognized by offline algorithms than by online algorithms.	28
4.4	Voting cannot choose between alternative groupings.	29
4.5	Distributions of the four distance measures.	30
4.6	Symbol recognition accuracy.	32
5.1	The correct choice of relation depends on symbol identity.	33
5.2	Hierarchical organization of semantic labels.	34
5.3	θ function component of relation membership grade.	36
5.4	Data clusters and splitting possibilities.	40
5.5	Relation classifier accuracy on 2009 data set.	44
5.6	Relation classifier accuracy on 2011 data set.	44
6.1	An expression in which the optimal relation depends on symbol identity.	48
6.2	An expression demonstrating a variety of subexpression linkages.	52
7.1	An ambiguous mathematical expression.	55

7.2	Shared parse forest for Figure 7.1.	56
7.3	Expressions with overlapping symbol bounding boxes.	59
7.4	Recursive rectangular partitions in an expression.	60
7.5	Illustration of expression extraction for a two-token production.	68
7.6	Interface for displaying and selecting alternative interpretations.	72
9.1	Default scenario results.	84
9.2	Perfect scenario results.	84

List of Tables

3.1	Variables names for grouping concepts.	17
3.2	Grouping evaluation results.	21
5.1	Relational class labels.	34
5.2	Angle thresholds for geometric relation membership functions.	36
9.1	Evaluation on CROHME 2011 corpus.	86
9.2	Evaluation on CROHME 2012 corpus.	87

Chapter 1

Introduction

Many software packages exist which operate on mathematical expressions. Such software is generally produced for one of two purposes: either to create two-dimensional renderings of mathematical expressions for printing or on-screen display (e.g., \LaTeX , MathML), or to perform mathematical operations on the expressions (e.g., Maple, Mathematica, Sage, and various numeric and symbolic calculators). In both cases, the mathematical expressions themselves must be entered by the user in a linearized, textual format specific to each software package.

This method of inputting math expressions is unsatisfactory for two main reasons. First, it requires users to learn a different syntax for each software package they use. Second, the linearized text formats obscure the two-dimensional structure that is present in the typical way users draw math expressions on paper. This is demonstrated with some absurdity by software designed to render math expressions, for which one must linearize a two-dimensional expression and input it as a text string, only to have the software re-create and display the expression's original two-dimensional structure.

Mathematical software is typically used as a means to accomplish a particular goal that the user has in mind, whether it is creating a web site with mathematical content, calculating a sum, or integrating a complex expression. The requirement to input math expressions in a unique, unnatural format is therefore an obstacle that must be overcome, not something learned for its own sake. As such, it is desirable for users to input mathematics by drawing expressions in two dimensions as they do with pen and paper.

Academic interest in the problem of recognizing hand-drawn math expressions originated with Anderson's doctoral research in the late 1960's [2]. Interest has waxed and waned in the intervening decades, and recent years have witnessed renewed attention to the topic, potentially spurred on by the nascent Competition on Recognition of Handwritten Mathematical Expressions (CROHME) [25]. We will explore several recent contributions to the field in the next chapter.

Math expressions have proved difficult to recognize effectively. Even the best state of the art systems (as measured by CROHME) are not sufficiently accurate for everyday use by non-specialists. The recognition problem is complex as it requires not only symbols

to be recognized, but also arrangements of symbols, and the semantic content that those arrangements represent. Matters are further complicated by the large symbol set and the ambiguous nature of both handwritten input and mathematical syntax.

My doctoral research has focused on this problem of recognizing handwritten mathematics in the context of the MathBrush project [13]. This thesis summarizes the theory and implementation techniques developed to produce the recognition components of MathBrush.

1.1 MathBrush recognizer overview

Before diving into technical details, it is worthwhile to explain the context of our math recognition system, as it grounds this research in a concrete, real-world application and places many useful and important constraints on the recognizer.

The math recognizer was developed for use in the MathBrush pen-math system, which allows users to input mathematical expressions by drawing them on a Tablet PC or iPad tablet screen. Following the input step, the expression is embedded in a worksheet interface in which the user may interact with it further by invoking computer algebra system (CAS) commands through context-sensitive menus, examining the results, and so on. As such, it is necessary for the recognizer to construct a semantic interpretation of the input so that it may communicate effectively with the CAS; a syntactic representation of the input is insufficient for this purpose.

The design goals of MathBrush emerged from earlier experiments [14] which discovered, unsurprisingly, that users became frustrated when incorrect recognition forced them to erase and re-write their input. This problem was exacerbated by a lack of feedback from the recognizer to indicate what had gone wrong, and by a disconnected recognition process in which symbols were recognized and then passed to a separate expression recognition system, which could fail even if all the symbols were recognized correctly. As a result of these early experiments, the two primary goals of the current MathBrush system became:

1. To present the user with constant feedback about the state of the recognizer during the input process. In particular, the recognizer’s current “best guess” is updated after each new stroke is drawn.
2. To allow the user to easily correct any recognition errors without erasing and re-writing their input. This is accomplished by selecting alternative recognition results from drop-down menus.

The goals of MathBrush as a whole place three requirements on the recognizer. The first is that, since recognition results are reported in real-time as the user writes, the recognizer should be fast enough to avoid an unreasonable delay between writing and viewing recognition results. This is the primary motivation for many of the simplifying assumptions that will appear later in the thesis (see Chapter 7).

The second requirement is that users are able to train the recognizer to match their handwriting style. This is most important for symbol shapes (e.g., American vs. European handwriting styles). It must therefore be straightforward to adapt the recognizer to a particular writing style. We comment on how such adaptation is possible in the MathBrush recognizer in Section 4.1.

The final requirement is that users must be able to correct erroneous recognition results. This implies that, rather than recognizing ink input as a single, definite math expression, the recognizer should have the following capabilities:

1. to obtain and present multiple interpretations of a given input,
2. to correct a particular subexpression of an interpretation, and
3. to maintain that correction as writing continues.

These requirements significantly complicate the recognition problem, and motivate our development of algorithms and data representations throughout the thesis, but particularly in Chapters 6 and 7.

1.2 Two viewpoints on math recognition

Given a sequence of hand-drawn input strokes, each of which is itself an ordered sequence of points in the plane, the math recognition problem is to represent the mathematical expression depicted by the strokes as a mathematical expression tree. From such a tree, the linearized text formats described above may easily be obtained. Of course, one may allow for more extensive input data – timings, pen pressure, tilt, and so on – but the structure of the problem remains the same, and most authors do not consider these additional sources of information because they are not reliably available on current hardware (touchscreen devices in particular).

Because of its complexity, it is worthwhile to break down the math recognition problem into smaller subproblems which may be more easily analyzed. We adopt two distinct viewpoints for this purpose: an abstract, theoretical view, and a practical, system-builder’s view.

1.2.1 The abstract view

On a large scale, the math recognition problem can be divided in three: 1) to determine what mathematical content a group of the input strokes represents; 2) to decide which of several interpretations of a group of strokes is the most plausible; and 3) to identify which groups of input strokes possess interpretations important to the structure of the expression as a whole, and how those groups combine together. These three problems may be conveniently called *recognition*, *scoring*, and *searching*. Together, they form a rubric

which serves to organize and focus discussion of the various inter-related parts of the math recognition problem.

These three subproblems are mutually dependent. To recognize a math expression requires an understanding of its constituent parts. To find those parts necessitates a search through alternatives. To compare alternatives requires some means of scoring them. To combine parts into larger expressions requires an understanding of whether and how the parts fit together. Despite their dependence, though, each subproblem has a quite distinct structure, and may be solved by distinct, independent strategies. Chapter 2 surveys several existing approaches to math recognition in terms of how researchers have addressed these three aspects of the problem.

1.2.2 The system-oriented view

Practical math recognition systems typically include a number of subsystems which are combined through some unifying formalism. Symbol recognizers, relation classifiers, grammar-based parsers, and other components may each be a part of a larger math recognition package. Each of these components may be implemented in many ways, and may have a different theoretical basis. For example, symbol recognition may be done via neural networks, Markov models, similarity metrics, etc., whereas a stroke grouping system may use timing information, convex hulls, and other geometric computations.

From the systems view, there are three high-level steps to solving the math recognition problem: 1) identifying how to organize the math recognition system into smaller subsystems; 2) building and optimizing the accuracy and performance of each individual subsystem; and 3) deciding how to combine the results of the subsystems into meaningful mathematical output.

The first two of these points are fairly straightforward. The next section outlines the high-level system organization of the MathBrush recognizer, and the bulk of this thesis (Chapters 3 to 7) covers the implementation of recognition subsystems in detail. The third point on combining subsystems is more subtle. It is a significant practical challenge to combine various subsystems in a way that both produces consistently meaningful results, and is theoretically sound. There is no a priori guarantee that the results produced by one subsystem are compatible with those produced by another, so effort is required when integrating recognition subsystems with one another. Chapter 8 examines our solution to this issue of subsystem output compatibility through the MathBrush recognizer's scoring framework.

1.3 Ambiguity and interpretation

Mathematical writing is inherently ambiguous at both the syntactic and semantic levels. Semantically, math is a relatively formal natural language [4] in which one statement often affords multiple interpretations, depending on context. For example, the notation $u(x+y)$ could be either a function application or a multiplication (at least). Similarly, f' might

imply differentiation of a function, or it could simply be a variable related in some way to another variable f .

These semantic ambiguities are impossible to resolve without contextual knowledge, and they are not addressed in this thesis. Rather, we assume a fixed set of supported mathematical semantics, and report all recognized interpretations of the user’s writing so that they may select the semantic interpretation they meant.

Syntactic ambiguity, though, is an unavoidable and interesting problem addressed throughout this work. For example, the expressions shown in Figure 1.1 might be reasonably interpreted as $Ax + b$, $A^x tb$, $Ax + 6$, and P^x , pX .



Figure 1.1: Syntactic ambiguity in simple expressions.

In such cases, contextual knowledge is extremely helpful and is used by people when reading mathematics. The answers to questions such as “which of P, p and t are known to be variables?” and “is A a matrix and x a vector?” help to disambiguate the writing. Unfortunately, this contextual knowledge is not currently captured by MathBrush (or any other math recognition software that we are aware of).

Informed by the interactive nature of MathBrush, we adopt a user-centric viewpoint that cuts across both the abstract and practical perspectives described above. When the user writes an expression, they have a particular interpretation of their writing in mind. Rather than discarding the ambiguity present in the user’s writing and selecting a single interpretation of the input, we opt to capture that ambiguity as completely as is practical, and organize a variety of interpretations in such a way that it is easy for the user to select their intended interpretation.

This decision influences the abstract subdivision of the math recognition problem into recognition, searching, and scoring. Recognition maps syntax onto semantics, searching organizes semantic interpretations into hierarchically-meaningful interpretation, and scoring orders those interpretations according to plausibility. Ideally, the highest-scoring interpretation of the input matches the user’s interpretation. But, because of imperfect recognition algorithms and the ambiguities just described, it is not possible for this to be the case all of the time.

In such cases, our goal is to make it as easy as possible for the user to obtain their interpretation from the recognizer. This is important in an interactive math system like MathBrush because recognition itself is not the goal, but computation. The system must understand the user’s interpretation for the subsequent mathematical operations to have any meaning. The recognition process thus resembles a preliminary conversation ensuring that the user and the system are on the same page, much as a student might ask a professor for clarification on what an equation or variable represents before the professor goes through a proof during a lecture.

The requirement to capture and organize ambiguity in the input also poses significant technical challenges in terms of software construction. Much more data must be organized than if only a single interpretation was to be reported, which in turn makes it more difficult to attain real-time execution. The user-oriented viewpoint informs the design of recognition algorithms throughout our software, and much of the novel work in this thesis originated from efforts to satisfy user-oriented requirements.

1.4 Contributions and thesis organization

In this thesis, we attempt to balance the abstract and system-oriented views described earlier. My focus has been mainly on the system-building viewpoint, and the theory we have developed plays largely a supporting role, explaining how to interpret the results of concrete systems, and how to combine those systems together. As such, this thesis documents the particular recognition system we have constructed, developing and explaining what theory is necessary as it progresses.

The recognizer consists of several primary subsystems:

- The *stroke grouping* system examines the input and identifies groups of strokes which may correspond to distinct symbols.
- The central *parsing* system searches for meaningful subdivisions and re-combinations of the input according to a grammar which specifies valid mathematical syntax.
- The parsing system interacts with *symbol* and *relation* classification systems:
 - The symbol classifier prepares a scored list of symbol identities that a particular group of input strokes is likely to represent.
 - The relation classifier prepares a scored list of spatial relationships which a pair of input subsets are likely to satisfy (superscript, horizontally adjacent, etc.)
- The symbol classifier makes use of a *symbol database* which organizes and controls access to a library of symbol examples written in various styles.

The rest of this thesis is organized as follows:

- The next chapter surveys related work in math recognition through the three-part rubric of recognition, searching, and scoring.
- Chapter 3 begins a sequence of three chapters on lower-level classification systems with a look at the stroke grouping algorithms used in the math recognition system. This topic is rarely discussed in academic literature, but as the first and most low-level point of contact between MathBrush and the recognizer, its design has important ramifications throughout the recognition process.

- Chapter 4 covers symbol recognition, considering several individual classification techniques including a novel variant of the traditional elastic matching algorithm. This chapter also discusses the hybrid classification system developed for the Math-Brush recognizer.
- Chapter 5 discusses geometric relationship classification and completes our look at isolated classification problems. Several methods are explored and compared including rule-based and probabilistic approaches. A new data structure called the margin tree is described and evaluated for the purpose of relation classification.
- Chapters 6 and 7 explore the theory and algorithms behind relational grammars and two-dimensional parsing. We first develop a sophisticated extension of existing relational grammar formalisms in Chapter 6, then describe in Chapter 7 some practical constraints affording efficient parsing algorithms. We give both bottom-up and top-down algorithms, describe how to simultaneously capture and represent multiple parse trees, and how to efficiently report those trees in decreasing score order.
- Chapter 8 develops fuzzy-set and probabilistic models of the recognition problem and applies them to the software systems described in earlier chapters. Each can be seen as a particular strategy for solving the abstract scoring problem, overlaid on the lower-level software systems. In the probabilistic case, an algebraic technique permits efficient calculation of probabilities despite the large number of variables present in the formal model.
- Chapter 9 wraps up the technical content of the thesis with two accuracy evaluations of the recognition system. A novel user-based accuracy metric is employed, as well as the detailed subsystem-oriented metrics from the Competition on Recognition Of Handwritten Mathematical Expressions (CROHME).
- Finally, Chapter 10 concludes the thesis with a summary of the research presented and a look at promising areas for future research.

Chapter 2

Related work

Over the last four and a half decades, many researchers have examined the problem of recognizing hand-drawn math input. In this chapter, I will use the three-pronged rubric of recognition, searching, and scoring developed in Chapter 1 to review the math recognition techniques developed by several of these researchers, focusing on more recent work.

2.1 Anderson

Anderson developed a grammar model in which each rule was associated with one or more predicates [2]. The predicates constrain the positions and sizes of the bounding boxes of the grammar rule's right-hand side elements. For example, the grammar rule for a fraction might include constraints specifying that the bottom of the bounding box of the numerator must lie above the middle of the bounding box of the fraction bar, which must in turn lie above the top of the bounding box of the denominator.

Anderson assumes that the symbols comprising the math expression have already been recognized with no errors or ambiguity. His input consists of bounding boxes labeled with symbol names. To obtain the math expression represented by a particular symbol arrangement, a depth-first search (DFS) is performed. Starting with the first grammar rule, each symbol is assigned to a RHS element such that the rule's constraints are satisfied, and then the algorithm recurses on each RHS element. If no assignment exists which satisfies any relevant rule's constraints, the algorithm backtracks. When a complete expression tree is detected, it is reported and the algorithm terminates. The grammar rules must therefore be carefully ordered so that the "correct" expression is selected before an incorrect expression is detected. (E.g., the rule for *sin* must appear before the multiplication rule to prevent *sin* being recognized as a multiplication.)

Anderson thus solves the recognition problem by assigning meaning to symbols through the semantic interpretation of his grammar rules, and he solves the searching problem quite literally through a depth-first search of the possible subdivisions of the input into smaller semantic units. His system does not compare alternative interpretations of the input, but merely reports the first valid interpretation it discovers, so he does not address the

scoring problem. While the system is impressive for its generality (Anderson also gives grammars for recognizing matrices and labeled graphs), it is ultimately inefficient (because of the DFS) and brittle (because of hard-coded grammar predicates and the dependence on grammar order).

2.2 Zanibbi

Some decades after Anderson’s seminal work, Zanibbi proposed a three-pass math recognizer based on the notion of *baseline structure trees* (BST) [38]. The first pass – the *layout* pass – is the most important. In it, baselines in the input are identified and structured in a tree that represents their hierarchical relationship with one another. For example, the fraction expression $\frac{a}{b}$ would be represented by a tree with the fraction bar as the root node, and two children, labeled “above” and “below” representing the numerator and denominator.

These baseline trees are obtained by determining which symbols represent mathematical operators which “dominate” other symbols. (E.g., the fraction bar dominates the a and b symbols, as they are arguments to its operation.) Dominated symbols are assigned to spatial regions afforded by the dominating operators by means of thresholds on bounding box position. (E.g., the fraction bar affords “above” and “below” regions, and symbols dominated by the fraction bar are assigned to one of those regions by comparing the symbols’ vertical centroids with the fraction bar’s bounding box.) This process of assigning symbols to baseline tree nodes is somewhat less flexible than Anderson’s predicate-based approach, but it is also much faster. BST construction is Zanibbi’s solution to the searching problem.

The *lexical* pass transforms the BST created by the layout pass by grouping together over-segmented symbols (e.g., two horizontal lines might be combined into =) and group-related symbols (e.g., the individual numbers 2 and 2 might be combined to form 22). Finally, the *expression* pass applies tree transformations to convert the BST into a math expression tree. One could say that Zanibbi solves the recognition problem by associating particular BST tree patterns with math semantics. However, in this case, the recognition process appears to an extent in all three passes. The first pass is largely devoid of recognition, being concerned solely with syntactic features, yet the rules used to generate baseline trees assume that mathematical semantics will be applied. So recognition is taking place to some extent when, for example, the algorithm searches for symbols above and below a fraction bar. The second pass also performs recognition, as it groups related syntactic units together into larger semantic groups such as function names and floating-point numbers.

Like Anderson, Zanibbi assumes that the input strokes have already been grouped into distinct symbols and recognized with no errors. Ambiguity in his system is only possible when the spatial regions of two or more operators overlap. In such cases, hard-coded rules are used to assign symbols in ambiguous regions to one region or another. Scoring is thus obviated in Zanibbi’s work as any ambiguity is eliminated a priori.

2.3 Winkler, Farhner, and Lang

Winkler, Farhner, and Lang also assume that symbols have already been recognized perfectly [35]. They solve the searching problem similarly to Zanibbi, by identifying symbols that represent math operators and assigning dominated symbols to spatial regions around the math operator. (In fact, the work of Winkler et al. predates that of Zanibbi.) Unlike both works discussed above, though, Winkler et al. address the scoring problem and hence permit their system to choose between multiple interpretations of the same input.

They do this by taking the reciprocals of the distances that a dominated symbol would need to be shifted in order for its bounding box to lie completely within each of the dominating symbol’s pre-defined spatial regions. These reciprocals are then normalized and treated as probabilities. Using our example $\frac{a}{b}$, the fraction bar possesses three regions: “above” and “below” as in Zanibbi’s work, and also “out”, which links to the next symbol in the same baseline.

In the case of fractions, only the most probable result is used. This is, in effect, identical to not using scoring at all. Although multiple candidates are considered, only one is used, and it is determined a priori by the distance calculations used to generate the probabilities. More interestingly, Winkler et al. permit symbols to participate ambiguously in superscripts, subscripts, and in-line relations. Each possibility is assigned a probability, as with the above and below relations just discussed. But rather than discarding everything but the most probable option, each possible relation leads to a different parse, and all parses are reported to the user. Winkler et al. apply thresholds to the probabilities in order to limit the number of parses reported, which is naively exponential in the number of input symbols.

The searching procedure of Winkler et al. thus has the same basis as that of Zanibbi, but is made more sophisticated through its allowance for ambiguous relations. It can be viewed as searching through the space of directed acyclic graphs (DAGs) with the symbols as nodes, and edges labeled with the relations linking the symbols.

Like Zanibbi, Winkler et al. only assign semantic meaning to symbols and relations as a final step, when the DAG generated by their algorithm may be converted into an expression tree. But, as we saw with Zanibbi’s work, recognition is implied in the earlier stages of the algorithm, which organize the symbols and relations based on the assumption that they possess mathematical significance.

2.4 Miller and Viola

Miller and Viola expanded upon the ambiguity permitted by Winkler et al. [24]. In their system, symbol recognition was *not* assumed to be complete and perfect. In fact, as well as permitting ambiguous relations between symbols, Miller and Viola permit the symbols themselves to be ambiguous. They introduced the notion of a *syntactic class* which captures the expected position of a symbol with respect to its baseline. For example, the number 9 extends from the baseline to the “top line”, the letter c from the baseline to the “mid line”,

and the letter q from below the baseline to the mid line. Miller and Viola use syntactic classes to manage ambiguity in symbol identity by seeding their parsing algorithm with the top-ranked alternative from each class for each symbol. They do not describe how the symbol recognition itself is performed, nor how the input strokes are grouped into distinct symbols.

The parsing algorithm itself proceeds in a bottom-up fashion. Beginning with the seeded symbols, it combines recognized subsets of the input to form larger recognized sets by applying grammar rules, similarly to the CYK algorithm for parsing CFGs. Rather than exhausting all possible subsets combinations, Miller and Viola introduce two key ideas to limit the search space. The first is a hard constraint that requires any set of strokes considered by the algorithm to have no other stroke intersecting its convex hull. This constraint was later expanded upon by Liang et al. who used rectangular hulls and partial-ordered sets for a similar purpose [18].

The second idea of Miller and Viola is to use the A-star search algorithm to guide the parser’s choice of subset combinations. They use the negative log likelihood as the underestimate of the “distance” to the search goal. Assuming independence of all symbols and relations, they treat the symbol recognition results probabilistically, and model relations between symbols as two-dimensional Gaussians. This approach builds on earlier work of Hull [12].

The parsing algorithm therefore performs a best-first search as it combines subsets which satisfy the convex hull criterion. In this way, scoring is integrated into the search process as a guiding factor, and recognition is a by-product of applying grammar rules during subset combination. Compared with previous work, the search process has again become more sophisticated, and the scoring more comprehensive and systematic.

2.5 Garain and Chaudhuri

In 2004, Garain and Chaudhuri proposed another system incorporating both symbol and expression recognition [7]. In it, symbols are recognized by measuring the distance and angle between consecutive points in each input stroke. These features are compared against template strokes by two classifiers (a feature vector classifier, and a hidden Markov model (HMM) classifier), the results of which are combined to obtain final recognition results. Symbols are recognized immediately as they are drawn, but Garain and Chaudhuri do not describe how the system determines that a symbol is complete. For example, after three strokes, an E may look exactly like an F, and it is not clear whether the system is able to revise its previous recognition results to account for this.

In any case, after a symbol is drawn and recognized, it is assigned to a particular “level” of the expression. The expression’s main baseline, to which the first symbol is assumed to belong, is level 0, and higher or lower levels represent writing lines above and below the main baseline. Levels are not directly analogous to baselines. For example, in the expression $2^n e^x$, the symbols n and x are distinct baselines, but are both on level 1 using Garain and Chaudhuri’s terminology.

By ordering symbols horizontally, spatial relations such as superscripts, subscripts, and horizontal alignment are determined by the level assignments. Mathematical semantics are directly associated with these relationships. More complex relationships like limits of integration are found by segmenting the input strokes into vertical and horizontal “stripes”, or contiguous regions, and then using a grammar to guide stripe recombination. Some specialized rules are invoked for vertical structures such as summations with limits. Thus Garain and Chaudhuri use a combination of baseline analysis and rectangular regions to solve the searching problem.

The system uses scoring to an extent when ranking symbol recognition results. But it does not discriminate between alternatives so much as reject invalid choices. After recognizing the input using the top-ranked symbol recognition results, the \LaTeX representation of the recognized expression is compared against a validation grammar. If validation fails, alternative character recognition results are explored, though it is not entirely clear how this search through alternatives proceeds.

2.6 Laviola, Zeleznik et al.

The MathPad project at Brown University ([17, 40]) was a long-running research programme for creating math sketching systems. Its high-level goals differ from those of MathBrush in that the Brown project aims to facilitate learning and problem-solving through interactive sketches controlled by mathematical formulae, while MathBrush intends to be a research tool for mathematicians and students. However, many software features required to attain each of these goals are similar, and so MathPad includes a math recognition system.

MathPad’s recognition system may be viewed as an informal application of grammar-based parsing. Rules governing how symbols are arranged into math expressions are explicitly coded into the program. Input is processed in a way analagous to how a particular organization of grammar productions could be naively parsed. The recognition system includes sophisticated heuristics specialized to particular mathematical structures (fractions, integrals, etc.). While this approach is highly tunable, it is less flexible than other, less explicitly-specified approaches.

A study of the recognition accuracy of MathPad was presented by LaViola [16]. In the study, 11 subjects individually provided the system’s symbol recognizer with 20 samples of their handwriting for each supported symbol. Each subject then provided 12 additional samples of each symbol as well as drawings of 36 particular math expressions as test data. The expressions were partly taken from Chan and Yeung’s collection [6] and partly designed by the author. This data was used to test MathPad’s symbol recognizer and math expression parser, respectively. Laviola measured the proportion of symbols recognized correctly and the proportion of parsing decisions which were correct. A parsing decision arises in LaViola’s system whenever symbols must be grouped together (or not) into a subexpression (e.g., \tan as multiplication versus the function name \tan) or a choice must be made about the type of a subexpression (e.g., superscript vs. inline)

2.7 Alvaro et al.

The system developed by Alvaro et al. [1] placed first in the CROHME 2011 recognition contest [25]. It is based on earlier work of Yamamoto et al. [36]. A grammar models the formal structure of math expressions. Symbols and the relations between them are modeled stochastically using manually-defined probability functions. The symbol recognition probabilities are used to seed a parsing table on which a CYK-style parsing algorithm proceeds to obtain an expression tree representing the entire input.

In this scheme, writing is considered to be a generative stochastic process governed by the grammar rules and probability distributions. That is, one stochastically generates a bounding box for the entire expression, chooses a grammar rule to apply, and stochastically generates bounding boxes for each of the rule's RHS elements according to the relation distribution. This process continues recursively until a grammar rule producing a terminal symbol is selected, at which point a stroke (or, more properly, a collection of stroke features) is stochastically generated.

Given a particular set of input strokes, Alvaro et al. find the sequence of stochastic choices most likely to have generated the input. However, stochastic grammars are known to be biased toward short parse trees (those containing few derivation steps) [23]. In our own experiments with such approaches, we encountered difficulties in particular with recognizing multi-stroke symbols in the context of full expressions. The model has no intrinsic notion of symbol segmentation, and the bias toward short parse trees caused the recognizer to consistently report symbols with many strokes even when they had poor recognition scores. Yet to introduce symbol segmentation scores in a straightforward way causes probability distributions to no longer sum to one. Alvaro et al. allude to similar difficulties when they mention that their symbol recognition probabilities had to be rescaled to account for multi-stroke symbols.

2.8 Awal et al.

While the system described by Awal et al. [3] was included in the CROHME 2011 contest, its developers were directly associated with the contest and were thus not official participants. However, their system scored higher than the winning system of Alvaro et al., so it is worthwhile to examine its construction.

A dynamic programming algorithm first proposes likely groupings of strokes into symbols, although it is not clear what cost function the dynamic program is minimizing. Each of the symbol groups is recognized using neural networks whose outputs are converted into a probability distribution over symbol classes.

Math expression structure is modeled by a context-free grammar in which each rule is linear in either the horizontal or vertical direction, simplifying the search problem. Spatial relationships between symbols and subexpressions are modeled as two independent Gaussians on position and size difference between subexpression bounding boxes. These

probabilities along with those from symbol recognition are treated as independent variables, and the parse tree with maximal likelihood is reported as the final parse.

This method as a whole is not probabilistic as the variables are not combined in a coherent model. Instead, probability distributions are used as components of a scoring function. This pattern is common in the math recognition literature: distribution functions are used when they are useful, but the overall strategy remains ad hoc.

2.9 Shi, Li, and Soong

Working with Microsoft Research Asia, Shi, Li, and Soong proposed a unified HMM-based method for recognizing math expressions [29]. Treating the input strokes as a temporally-ordered sequence, they use dynamic programming to determine the most likely points at which to split the sequence into distinct symbol groups, the most likely symbols each of those groups represent, and the most likely spatial relation between temporally-adjacent symbols. Some local context is taken into account by treating symbol and relation sequences as Markov chains. This process results in a DAG similar to one obtained by Winkler et al., which may be easily converted to an expression tree.

To compute symbol likelihoods, a grid-based method measuring point density and stroke direction is used to obtain a feature vector. These vectors are assumed to be generated by a mixture of Gaussians with one component for each known symbol type. Relation likelihoods are also treated as Gaussian mixtures of extracted bounding-box features. Group likelihoods are computed by manually-defined probability functions.

This approach is elegant in its unity of symbol, relation, and expression recognition. The reduction of the input to a linear sequence of strokes drastically simplifies the search problem, while probabilistic rules and the probabilistic models selected by the authors constitute a scoring mechanism. The interpretations associated with particular assignments of the model's variables provide recognition, in combination with the features selected for extraction.

But this assumption of linearity comes at the expense of generality. The HMM structure strictly requires strokes to be drawn in a pre-determined linear sequence. That is, the model accounts for ambiguity in symbol and relation identities, but not for the two-dimensional structure of mathematics.

Chapter 3

Stroke grouping

The role of the stroke grouping system is to determine, given a set of ink strokes, which subsets of those strokes correspond to distinct symbols. For example, numbering the strokes in Figure 3.1 1,2,3,4,5 from left to right, the stroke grouper should report $\{1\}$, $\{2, 3\}$, $\{4, 5\}$ as likely symbols. This task, though necessary for any handwriting recognition system, is generally neglected in most publications. Note that the identities of those symbols (i.e., a, +, and b) is irrelevant here. We are concerned only with proposing reasonable subsets of the ink as candidates for symbol recognition.

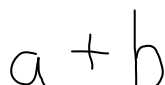
The image shows the handwritten text 'a + b'. The 'a' is a simple lowercase letter. The '+' is a plus sign. The 'b' is a lowercase letter with a vertical stem and a rounded bottom.

Figure 3.1: A simple input for the stroke grouper.

Figure 3.1 might suggest that this task is straightforward, so consider the input shown in Figure 3.2. It is difficult to determine whether this input is supposed to represent aoH or $ad - 1$, even for a mathematically-literate human. But the user who drew this input presumably knew what they intended it to represent. We will rely in such cases on our strategy of presenting the user with several reasonable interpretations of their input so that they can select the one they intended.

The image shows the handwritten text 'aoH'. The 'a' is a lowercase letter. The 'o' is a lowercase letter. The 'H' is an uppercase letter.

Figure 3.2: A more difficult input for the stroke grouper.

It is vitally important for the grouper to avoid false negatives. If an ink subset that corresponds to a symbol in the user's interpretation of the ink is not reported by the stroke grouping system, then that subset will not be subject to symbol recognition, and will not be included in any recognition results. Therefore, false negatives preclude correct recognition of the input and should be avoided at all costs.

The example in Figure 3.2 indicates that the grouper need not limit itself to a single partitioning of the ink into symbol groups. Rather, any subset of the input that could reasonably be considered a distinct symbol should be reported, so as to maximize the likelihood of identifying the user’s preferred interpretation of their writing.

These two observations imply that false positives are acceptable. That is, it is acceptable for the grouper to identify as a potential symbol a subset of the input which does not actually correspond to a symbol in the user’s interpretation. We may reasonably assume that many false positives will be filtered out by later stages of recognition. As an extreme example, suppose the stroke grouping system identifies the $a+$ portion of Figure 3.1 as a potential symbol. These strokes are unlikely to be recognized with high confidence as a single symbol. So long as the grouper also proposes the a and $+$ subsets individually as potential symbols, overall recognition should still give the result the user expects.

To summarize, the role of the stroke grouper is to select subsets of the input for symbol recognition such that all subsets corresponding to symbols in the user’s interpretation of the input are included. The rest of this chapter describes and evaluates our techniques for accomplishing this goal.

3.1 Relevant input characteristics

When is a group of ink strokes a good candidate for symbol recognition? One obvious answer is “when they are close together.” Strokes that are far apart ought not to belong to the same symbol. However, we should not be too hasty to group nearby strokes. Consider Figure 3.3 which contains many touching strokes (i.e., separated by zero distance), yet each stroke corresponds to a distinct symbol. Nonetheless, distance is a useful metric, as many symbols like B , $+$, π , λ , k may all be written with multiple strokes close to one another.



Figure 3.3: Distance between strokes is not an infallible grouping criterion.

Another seemingly obvious answer is “when they are drawn one after another.” It is certainly true that consecutive strokes are more likely to be part of the same symbol than non-consecutive strokes, but again, care is needed. A user may write $\sin(2\theta)$ without the dot and go back later to dot the i . Or they may write the numerator of a fraction and the fraction bar, but while writing the denominator realize that they did not draw the bar long enough and extend it. In the context of MathBrush, they might write a polynomial, do some computation with it, and then realize based on unexpected results that a minus sign in the input should have been a plus, and correct it. These are all plausible examples of symbols built from non-consecutive strokes. As such, we do not use stroke ordering when determining potential symbol groups.

Note that all of the cases just described follow a pattern in which some strokes for a symbol are drawn, then some unrelated strokes are drawn, then some strokes for that symbol are added *and are nearby the original strokes*. So stroke proximity alone is able to capture these patterns.

In some cases, such as in the π and F symbols shown in Figure 3.4, not all of the strokes making up a symbol will be near to one another. Or at least, the distance between the strokes will be large enough that it is quite ambiguous whether the strokes belong to the same symbol, or whether the writing is just closely spaced, as in Figure 3.3. In these cases, though, it is often the case that the bounding box of each stroke has a large intersection with the bounding box of the other strokes.



Figure 3.4: Bounding box overlap is another good indicator of stroke groups.

Bounding box overlap is thus another useful feature to consider when grouping strokes. But here too care is needed. Some symbol arrangements, in particular the syntax for square roots, cause large bounding box intersections but ought not to be grouped together, as shown in Figure 3.5. So large bounding box intersections should suggest symbol grouping unless a square root or other containment notation is involved. Conversely, a group should not have a large overlap with strokes outside of the group, unless a containment notation is involved.

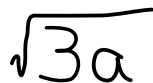


Figure 3.5: Bounding box overlap is not always a reason to group strokes into symbols.

Labeling the concepts in this discussion as shown in Table 3.1, the following logical equation summarizes the characteristics of a “good” group identified above:

$$G = (D_{in} \vee (L_{in} \wedge \neg C_{in})) \wedge (\neg L_{out} \vee C_{out}) \tag{3.1}$$

Name	Meaning
G	A group exists
D_{in}	Small distance between strokes within group
L_{in}	Significant bounding box overlap of strokes within group
C_{in}	Containment notation used within group
L_{out}	Significant bounding box overlap of group and non-group strokes
C_{out}	Containment notation used in group or overlapping non-group strokes

Table 3.1: Variables names for grouping concepts.

Because of the ambiguity of handwritten input and the requirement of avoiding false negatives, it is unreasonable to assign binary values to the variables in 3.1. Instead, we

assign quantitative scores to the variables in a probabilistic evaluation scheme. This process is described below.

3.2 Grouping algorithm

Because the MathBrush recognizer receives input strokes one-by-one from the interface, we formulate the grouping problem as follows. Given a set of existing stroke grouping candidates (not necessarily a partition of the input; i.e., the groups may overlap) and a new input stroke, find a new set of grouping candidates incorporating the new stroke. This amounts to identifying the existing groups to which the new stroke may be reasonably added, and potentially augmenting those new groups with existing strokes. This second augmentation step is necessary because of cases like the π shown in Figure 3.4. If the two vertical strokes of π are first drawn fairly far apart, it is unlikely that they will be identified as being part of the same symbol. Only after the third stroke is drawn can the entire symbol be reliably identified.

At a high level, group identification proceeds according to Algorithm 1.

Algorithm 1 High-level stroke grouping algorithm.

Require: A set $S = \{s_1, \dots, s_n\}$ of input strokes and a set $G = \{g_1, \dots, g_N\}$ of subsets of $S \setminus \{s_n\}$
Initialize $G' \leftarrow \{\}$
(add s_n to existing groups)
for each group $g \in G$ **do**
 if $\text{score}(g, s_n) > 0$ **then**
 add $g \cup \{s_n\}$ to G'
(augment new groups with existing strokes)
while $G' \neq \{\}$ **do**
 $G \leftarrow G \cup G'$; $G'' \leftarrow \{\}$
 for each group $g \in G'$ and each stroke $s \in S$ such that $s \notin g$ **do**
 if $\text{score}(g, s) > 0$ **then**
 add $g \cup \{s\}$ to G''
 $G' \leftarrow G''$

The interesting portion is, of course, the score function referenced by the algorithm. The value of $\text{score}(g, s)$ corresponds to augmenting a group g with a stroke s . To compute it, we take several measurements of the inputs:

- $d = \min \{\text{dist}(s', s) : s' \in g\}$, where $\text{dist}(s_1, s_2)$ is the minimal distance between the curves traced by s_1 and s_2 ;
- $\ell_{in} = \text{overlap}(g, s)$, where $\text{overlap}(g, s)$ is the area of the intersection of the bounding boxes of g and s divided by the area of the smaller of the two boxes;

- $c_{in} = \max(C(s), \max\{C(s') : s' \in g\})$, where $C(s)$ returns the extent to which the stroke s resembles a containment notation, as described below;
- $\ell_{out} = \max\{\text{overlap}(g \cup \{s\}, s') : s' \in S \setminus (g \cup \{s\})\}$;
- $c_{out} = \max(c_{in}, C(s'))$, where s' is the maximizer for ℓ_{out} .

To account for containment notations in the calculation of c_{in} and c_{out} , we annotate each symbol supported by the symbol recognizer with a flag indicating whether it is used to contain other symbols. (Currently only the $\sqrt{\quad}$ symbol has this flag set.) Then the symbol recognizer is invoked to measure how closely a stroke or group of strokes resembles a container symbol. The details of how this is done will be discussed in the next chapter which covers symbol recognition. For now, we simply model the symbol recognizer as a function $C(T)$ of a set T of strokes that returns a value in $[0, 1]$ indicating the degree to which T resembles a container, with 0 being not at all and 1 being a very close resemblance.

Using DeMorgan’s law, the logical predicate in Equation 3.1 can be rewritten as

$$\begin{aligned} G &= \neg(\neg D_{in} \wedge \neg(L_{in} \wedge \neg C_{in})) \wedge \neg(L_{out} \wedge \neg C_{out}) \\ &= \neg(\neg D_{in} \wedge \neg X_{in}) \wedge \neg X_{out}, \end{aligned}$$

where $X_* = L_* \wedge \neg C_*$.

This predicate may be transformed into a probability calculation by defining conditional distributions. We set

$$\begin{aligned} \text{score}(g, s) &= P(G \mid d, \ell_{in}, c_{in}, \ell_{out}, c_{out}) \\ &= (1 - P(\neg D_{in} \mid d) P(\neg X_{in} \mid \ell_{in}, c_{in}))^\beta P(\neg X_{out} \mid \ell_{out}, c_{in}, c_{out})^{1-\beta} \end{aligned} \quad (3.2)$$

where

$$\begin{aligned} P(\neg D_{in} \mid d) &= (1 - e^{-d/\lambda})^\alpha \\ P(\neg X_{in} \mid \ell_{in}, c_{in}) &= (1 - \ell_{in} (1 - c_{in}))^{(1-\alpha)} \\ P(\neg X_{out} \mid \ell_{out}, c_{out}, c_{in}) &= 1 - \ell_{out} (1 - \max(c_{in}, c_{out})). \end{aligned}$$

This definition is loosely based on treating the boolean variables from our logical predicate as random and independent. λ is estimated from training data, with α, β both set based on experiments to 9/10.

Notice that each of the measurement variables is only defined for multiple strokes. It remains to handle single-stroke groups so that one-stroke symbols will be recognized. To do so, denote the score of a multi-stroke group g by

$$\text{score}(g) = \max\{\text{score}(g', s_i) : g' \cup \{s_i\} = g\}.$$

Often this score will coincide with the value $\text{score}(g', s_n)$ computed in the first half of Algorithm 1 (where $g = g' \cup \{s_n\}$). But because of the augmentation step in the second

half of the algorithm, it is possible for the same group to be obtained through different sequences of stroke addition, so the maximization is necessary.

Now for each individual stroke s , we simply create a default group g_s with score

$$\text{score}(g_s) = 1 - \max \{ \text{score}(g) : g \cap g_s \neq \{\} \}.$$

As each input stroke is received by the MathBrush recognizer, Algorithm 1 is used to augment the existing set of candidate symbols by incorporating the new stroke. All of the resulting groups, along with their scores, are forwarded to the symbol recognition system for subsequent processing and scoring.

3.3 Evaluation

The grouping algorithm is somewhat difficult to evaluate in isolation because it is designed to be used in conjunction with higher-level recognition systems like symbol recognizers. I have employed two correctness metrics:

1. **Basic:** a symbol is grouped correctly if its group score is non-zero.
2. **Strict:** a symbol is grouped correctly if its group score is higher than all other groups sharing strokes with the symbol.

These metrics respectively represent upper and lower bounds on the real-world performance one can expect from the grouping algorithm. The basic metric overestimates the algorithm’s performance because some groups with non-zero scores will be scored too low for higher-level recognizers to recover from; those groups will not be reported as symbols during regular use. But the strict metric underestimates the algorithm’s performance because of our expectation of false positives: if the correct group has a score close to an intersecting but incorrect group’s score, the symbol recognizer will very often prefer the lower-scoring group because the incorrect group has no reasonable symbol recognition results.

The grouping algorithm and score functions defined above were evaluated on a data set collected in 2009 [20] consisting primarily of randomly-generated math expressions transcribed by twenty undergraduate students. The data was randomly divided roughly in half, with one half used for training and the other for testing. The stroke groups corresponding to symbols in the training data were extracted and used to obtain the scoring function parameter λ , then the grouping algorithm was applied to the test files and the two metrics above were measured. This experiment was replicated five times with five different splits of the data. The evaluation results are shown in Table 3.2.

These results indicate that the grouping algorithm reports over 99% of symbols, and scores roughly 90% of actual groups higher than all false positives. We will see in Chapter 9 that, in the context of full math expression recognition, the grouping algorithm’s output results in about 97% of symbols being grouped correctly.

Metric	Split 1	Split 2	Split 3	Split 4	Split 5	Average
Basic	99.25%	99.56%	99.55%	99.58%	99.59%	99.51%
Strict	89.16%	89.61%	90.03%	90.01%	89.80%	89.72%

Table 3.2: Grouping evaluation results.

Chapter 4

Symbol recognition

The role of the symbol recognition system is to examine the input subsets identified by the grouping system as potential symbols, and to determine which symbols, if any, those groups represent. These results are, in turn, incorporated into higher-level systems which identify mathematical expressions in the input.

4.1 Recognition algorithms

All of the symbol recognition algorithms discussed in this chapter follow the same distance-based classification paradigm. In this scheme, one uses a distance function $d(I, M)$ to measure the difference between a set I of *input* strokes and each set M of *model* strokes taken from a library L of labeled symbol models. Then I is recognized by d as the label attached to $\operatorname{argmin} \{d(I, g) : g \in L\}$. This approach is easily extended to report ordered lists of the k best recognition candidates. It also easily handles writer-dependent training, as symbols drawn by a particular user may simply be added to the library to adapt recognition to the user's writing style.

For the first three (online) algorithms below, we define the set-wise distance function above in terms of an algorithm-specific stroke-wise distance function:

$$d(I, M) = \begin{cases} \infty & \text{if } |I| \neq |M| \\ \frac{\sum_{s \in I} d(s, f_{I,M}(s))}{|I|} & \text{otherwise} \end{cases} \quad (4.1)$$

where $f_{I,M}$ is a function mapping each input stroke in I to the corresponding model stroke in M . This mapping is necessary to handle situations in which the strokes comprising the input symbol are drawn in a different order than those comprising the model symbol. Its construction will be discussed in Section 4.1.1. For the final (offline) algorithm, the number of input strokes need not match the number of model strokes, so the case yielding an infinite distance is unnecessary.

In cases where the stroke-wise distance function is not symmetric ($d(s_1, s_2) \neq d(s_2, s_1)$), we average the results of both match directions so that the second case of Equation 4.1

becomes

$$\frac{1}{2} \left(\frac{\sum_{s \in I} d(s, f_{I,M}(s))}{|I|} + \frac{\sum_{s \in M} d(s, f_{M,I}(s))}{|M|} \right) \quad (4.2)$$

Prior to invoking the distance function, both I and M are normalized such that the largest side of their bounding box has length 1. Next we will describe each of the four distance functions used by the recognizer.

4.1.1 Feature vector norm

The first distance function is the simplest. We define several functions $f_1(s), f_2(s), \dots, f_n(s)$, each measuring some feature of a stroke as a real number, and take

$$d(a, b) = \sqrt{\sum_{i=1}^n (f_i(a) - f_i(b))^2},$$

that is, the 2-norm of the vector of feature differences. We employ two sets of features: one for use with small punctuation marks, and the other for the general case.

- **Small symbol features:** aspect ratio; distance between stroke endpoints.
- **Standard features:** bounding box position and size; first and last point; length along stroke.

The function $f_{I,M}$ matching up strokes between the input and model stroke sets is defined greedily using this feature vector distance function, as follows:

1. Pick a stroke $a \in I$.
2. Find the stroke $b \in M$ minimizing $d(a, b)$, where d is the feature vector distance function. Set $f_{I,M}(a) = b$ and remove a and b from consideration in future processing.
3. Repeat until no unmatched input strokes remain.

4.1.2 Greedy elastic matching

The use of elastic matching goes back to Tappert [31], who developed it as a technique for recognizing cursive writing. In the standard formulation, each point in the input stroke is matched with a point in the model stroke and a pointwise distance measure is applied to each matched pair. The stroke-wise distance function is then the sum of pointwise distances between matched pairs. Tappert’s algorithm finds the minimal such distance, subject to the three following constraints:

1. The first input point is matched to the first model point;

2. The last input point is matched to the last model point; and
3. If the i th input point is matched to the j th model point, then the $i + 1$ st input point is matched to the j th, $j + 1$ st, or $j + 2$ nd model point.

Let the input stroke be the sequence of points $\{(\hat{x}_i, \hat{y}_i) : i = 1, \dots, n\}$ and the model stroke be the sequence $\{(x_i, y_i) : i = 1, \dots, m\}$. Given a distance function $g(i, j)$ between the points (\hat{x}_i, \hat{y}_i) and (x_j, y_j) , the above constraints induce the dynamic program $D[i, j]$, representing the minimal distance between model points 1 through i and input points 1 through j , as follows:

$$\begin{aligned} D[1, j] &= g(1, j) + D[1, j - 1] \\ D[2, j] &= g(2, j) + \min \{D[2, j - 1], D[1, j - 1]\} \\ D[i, j] &= g(i, j) + \min \{D[i - k, j - 1] : k = 0, 1, 2\} \end{aligned}$$

Note that many distance functions g are possible. For cursive writing, Tappert uses

$$g(i, j) = \min \left\{ \left| \hat{\theta}_i - \theta_j \right|, \left| 360 - (\hat{\theta}_i - \theta_j) \right| \right\} + \alpha |\hat{y}_i - y_j|,$$

where θ_i is the tangent angle at (x_i, y_i) and α is chosen so that the angular and y -coordinate components of the sum have equal weight.

Tappert gave a quadratic-time dynamic programming algorithm to find the elastic matching distance between two strokes. Each dynamic programming table cell requires only constant time to compute, but there are $\mathcal{O}(nm) \approx \mathcal{O}(n^2)$ table cells to be computed and stored. In the context of math recognition, the symbol library is quite large because of the presence of Greek letters and mathematical symbols, and symbol recognition processes may be invoked several times if there are several ways to partition a large input into distinct symbols. We found that the quadratic matching time per stroke consumed a significant proportion of total processing time in MathBrush, prompting our development of a faster variant.

Greedy elastic matching

We motivate our algorithm by some straightforward observations about Tappert's constraints. Let I_1, I_2, \dots, I_n be the points comprising the input stroke, and M_1, M_2, \dots, M_m be similar for the model stroke. According to constraints 1 and 2, I_1 must be matched to M_1 and I_n must be matched to M_m . By constraint 3, I_2 must be matched to one of M_1, M_2, M_3 , and I_{n-1} must be matched to one of M_m, M_{m-1}, M_{m-2} . Similarly, supposing I_i is matched to $M_{f(i)}$, I_{i+1} must be matched to one of $M_{f(i)}, M_{f(i)+1}, M_{f(i)+2}$, and I_{i-1} must be matched to one of $M_{f(i)}, M_{f(i)-1}, M_{f(i)-2}$.

Tappert's dynamic program finds a globally-optimal matching satisfying these constraints. Our approximate version is to simply match endpoints to endpoints, then to greedily choose the locally-optimal matching from the available options for each intermediate point along the input stroke. To ensure endpoints are matched together, we perform a

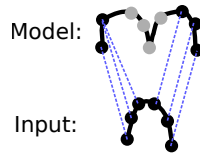


Figure 4.1: Many unmatched model points remain after matching every input point.

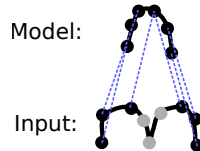


Figure 4.2: No more model points are available, but several input points remain to be matched.

two-sided match beginning at the start and end of the strokes and working simultaneously toward the middle.

There are two potential problems we must be aware of in this scheme, particularly if the number of points differs significantly between the input and model strokes. After matching all the input points to model points, there may be a large number of points in the middle of the model stroke which were never considered by the algorithm. Conversely, the algorithm may run out of model points available for matching before all of the input points have been considered. These situations are exemplified schematically by Figures 4.1 and 4.2, respectively. In the figures, dashed lines indicate pairs of matched points, and grey points are unmatched and problematic.

To account for these cases, we include the following two rules in our procedure:

1. After matching each input point to a model point, implicitly match the center-most input point to every second model point not yet considered for matching. (This process simulates skipping over model points, as permitted by Tappert's constraints.)
2. If there are no available model points to consider for matching, match all remaining input points to the center-most model point.

These rules immediately give an algorithm for approximate dynamic time warping, listed in Algorithm 2.

Regardless of the length of the strokes, the algorithm uses a fixed number of variables to track point indices, local and global match costs, and which local match choice was optimal. In each iteration of the main while loop (line 7), f_I is incremented and b_I is decremented, so only $n/2$ iterations are possible. Notice that the loop body requires only constant time, assuming g requires constant time. If the else clause at line 24 is invoked, then the loop at line 30 will not be entered; otherwise that loop will run at most $m/2$ times. The algorithm's runtime is thus linear in the number of input and model points.

Algorithm 2 Greedy approximate dynamic time warping.

Require: Input and model strokes of n, m points respectively; distance function $g(i, j)$ as in the previous section.

(Initialize indices to the start and end of strokes)

$f_I \leftarrow 1; b_I \leftarrow n; f_M \leftarrow 1; b_M \leftarrow m$

(Match endpoints)

$c_{f_0} \leftarrow g(f_M, f_I); c_{b_0} \leftarrow g(b_M, b_I)$

5: $c \leftarrow c_{f_0} + c_{b_0}$

$f_I \leftarrow f_I + 1; b_I \leftarrow b_I - 1$

while $f_I < b_I$ **do**

(Measure relevant local match costs)

$r \leftarrow b_M - f_M$

10: **if** $r > 0$ **then**

$c_{f_0} \leftarrow g(f_M, f_I); c_{b_0} \leftarrow g(b_M, b_I)$

$c_{f_1} \leftarrow g(f_M + 1, f_I); c_{b_1} \leftarrow g(b_M - 1, b_I)$

if $r > 1$ **then**

$c_{f_2} \leftarrow g(f_M + 2, f_I); c_{b_2} \leftarrow g(b_M - 2, b_I)$

15: **else**

$c_{f_2} \leftarrow \infty; c_{b_2} \leftarrow \infty$

(Choose minimum-cost match locally)

$i \leftarrow \operatorname{argmin} \{c_{f_k} : k = 0, 1, 2\}$

$j \leftarrow \operatorname{argmin} \{c_{b_k} : k = 0, 1, 2\}$

20: $c \leftarrow c + c_{f_i} + c_{b_j}$

(Advance to the next points under consideration)

$f_M \leftarrow f_M + i; b_M \leftarrow b_M - j$

$f_I \leftarrow f_I + 1; b_I \leftarrow b_I - 1$

else

25: *(Model exhausted; match remaining input points to last matched point)*

while $f_I < b_I$ **do**

$c \leftarrow c + g(f_M, f_I)$

$f_I \leftarrow f_I + 1$

(Input exhausted; match remaining model points to last matched point)

30: **while** $f_M < b_M$ **do**

$c \leftarrow c + g(f_M, f_I)$

$f_M \leftarrow f_M + 2$

return c

4.1.3 Functional curve approximation

Another approach, described by Golubitsky and Watt [9], is to approximate the model and input strokes by parametric functions and compute some norm on the difference between those approximations. They interpret a stroke $s = \{(x_i, y_i) : i = 1, \dots, n\}$ as a pair of functions $x, y : [0, 1] \rightarrow \mathbb{R}$ parameterizing the stroke by proportion of arclength. Each of these functions is approximated by a truncated series expansion of the form

$$\sum_{i=0}^k \alpha_i B_i(\lambda),$$

where the B_i are the orthogonal basis functions for the Legendre-Sobolev inner product; i.e., they satisfy

$$\langle B_i, B_j \rangle = \int_0^1 B_i(\lambda) B_j(\lambda) d\lambda + \mu \int_0^1 B_i'(\lambda) B_j'(\lambda) d\lambda = \delta(i - j), \quad (4.3)$$

where δ is the Kronecker delta function.

Finally, the distance function measuring the difference between input and model strokes is given by

$$\sqrt{\sum_{i=0}^k \left(\alpha_i^{(x)} - \beta_i^{(x)} \right)^2 + \left(\alpha_i^{(y)} - \beta_i^{(y)} \right)^2},$$

where the $\alpha_i^{(x)}$ and $\alpha_i^{(y)}$ are the series coefficients for the x - and y -coordinate functions of the input stroke, respectively, and the $\beta_i^{(x)}$ and $\beta_i^{(y)}$ are similar for the model stroke. That is, the stroke-wise distance is the two norm of the difference between the concatenated coefficient vectors.

The computation of the coefficients is interesting because it is designed for use in a streaming setting, requiring only $\mathcal{O}(k)$ operations per point plus $\mathcal{O}(k^2)$ for normalization (we use $k = 12$). It proceeds as follows.

First, approximate the moment integrals

$$m_i = \int_0^L \lambda^i f(\lambda) d\lambda,$$

for $f = x, y$, where $i = 0, \dots, k$, and L is the total arclength of the stroke. These are computed piecewise, extending the range of the integral by adding a new term

$$\int_{\ell_{j-1}}^{\ell_j} \lambda^i f(\lambda) d\lambda \approx \frac{\ell_j^{i+1} - \ell_{j-1}^{i+1}}{i+1} \times \frac{f(\ell_j) + f(\ell_{j-1})}{2}$$

for each point (x_j, y_j) in the stroke (ℓ_j denotes the stroke arclength up to point j).

After all points are processed, the moments are normalized to have domain $[0, 1]$ (so $m_i(f) = \int_0^1 \tau^i f(\tau) d\tau$, where $\tau = \lambda/L$), and the coefficients

$$\alpha_i(f) = \langle f(\tau), B_i(\tau) \rangle = \int_0^1 f(\tau) B_i(\tau) d\tau + \mu \int_0^1 f'(\tau) B_i'(\tau) d\tau$$

are found by integrating by parts:

$$\int_0^1 f(\tau)B_i(\tau) d\tau + \mu \left(f(\tau)B_i'(\tau)|_0^1 - \int_0^1 f(\tau)B_i''(\tau) d\tau \right)$$

and moving the sums outside the integrals:

$$\begin{aligned} & \sum_{j=0}^i [\tau_j] B_i(\tau) \int_0^1 \tau^j f(\tau) d\tau + \mu \left(f(\tau)B_i'(\tau)|_0^1 - \sum_{j=0}^{i-2} [\tau^j] B_i''(\tau) \int_0^1 \tau^j f(\tau) d\tau \right) \\ &= \sum_{j=0}^i [\tau_j] B_i(\tau)m_j(f) + \mu \left(f(\tau)B_i'(\tau)|_0^1 - \sum_{j=0}^{i-2} [\tau^j] B_i''(\tau)m_j(f) \right) \end{aligned}$$

where $[\tau^j] X$ denotes the coefficient of τ^j in X .

These operations yield a system of equations that is linear in the coefficients of the orthogonal basis polynomials, which are easily precomputed based on Equation 4.3.

Finally, the functional approximation is centered at the origin by setting the constant coefficients $\alpha_0(x)$ and $\alpha_0(y)$ of both the x and y series to zero, and the coefficient vector is normalized to have length one.

4.1.4 Hausdorff measure

The three preceding distance functions are all *online*. That is, they consider the input stroke to be a time-ordered sequence of points, as opposed to *offline* approaches which consider an unordered set of points (typically binarized image data). The ordering information is often quite useful, but is not applicable in all situations.

Two such cases are illustrated in Figure 4.3. On the left, the looped shape of the o or 0 symbol makes it difficult to tell whether the loop was drawn in a clockwise or counter-clockwise direction without tracing the curve. An online distance function may process the input stroke in one direction and the model stroke in another, leading to an unexpectedly large distance, whereas an offline measurement naturally obtains a smaller distance since point ordering is irrelevant. On the right, one a was drawn with one stroke while the other was drawn with two, precluding the use of online distance algorithms without sophisticated pre-processing. Again, an offline technique that considers only appearance without point order handles this case easily.



Figure 4.3: These symbols may be more easily recognized by offline algorithms than by online algorithms.

The offline distance used in the MathBrush recognizer is a variant of the distance used by Thammano and Rugkunchon [32] in their neural-network-based classifier. It is based

on the Hausdorff distance

$$d_H(X, Y) = \max \left\{ \sup_{x \in X} \inf_{y \in Y} d(x, y), \sup_{y \in Y} \inf_{x \in X} d(y, x) \right\} \quad (4.4)$$

between two subsets X and Y of a metric space (M, d) .

In particular, the input and model strokes are first rasterized onto grids having the same aspect ratio as the model stroke and 16 cells along their longest side. Each grid is then treated as a set of points $\{(x, y) : \text{cell } (x, y) \text{ of the grid is filled}\}$, and the Hausdorff distance can be taken using Euclidean distance as the pointwise distance function d .

We have found that summing, rather than maximizing, the distance over the filled grid cells gives better results in practice, so that Equation 4.4 becomes

$$d_S(X, Y) = \max \left\{ \frac{1}{|X|} \sum_{x \in X} \min_{y \in Y} d(x, y), \frac{1}{|Y|} \sum_{y \in Y} \min_{x \in X} d(y, x) \right\}. \quad (4.5)$$

4.2 Combining recognizers

It is often the case that, when one of the above techniques mis-classifies a symbol, another of them will classify the same symbol correctly (see the experiments below in Section 4.3). It is therefore worthwhile to consider combining all four techniques into a hybrid classifier.

Such combinations are often made by the so-called *voting method* or the related but more sophisticated Borda Count [34]. In this method, the model symbols are sorted into k lists (one for each classifier); each list is ordered by increasing distance from the input. Then each model symbol S is assigned $\sum_{i=1}^k \text{pos}_i(S)$ votes, where $\text{pos}_i(S)$ is the position at which S appears in the ordered list for classifier i . The final ranked list of classifier results is sorted in increasing order of votes.

The voting method works well when the set of input strokes is known to correspond to an actual symbol. But because it discards the underlying distance measurements, it cannot be used to compare recognition results amongst several alternative sets of input strokes. Such a situation is illustrated by the three leftmost strokes in Figure 4.4. Numbering the strokes from left to right as 1, 2, and 3, suppose the grouping system proposes all of $\{1\}$, $\{2, 3\}$, and $\{1, 2, 3\}$ as potential symbols. Then the rest of the recognition system must choose whether to interpret the input as one or two symbols.

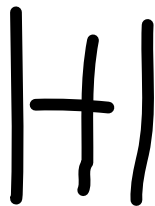


Figure 4.4: Voting cannot choose between alternative groupings.

In this situation, the voting method may easily fail. It is likely that most of the classifiers will rank H very highly for the potential symbol $\{1, 2, 3\}$, giving it a high final voting rank even though the distance function may have reported relatively large distances between the stroke sets. There is more ambiguity between competing symbols for the two smaller groups (1 vs. l and t vs. +, for example). It is therefore less likely for reasonable interpretations of those groups to be as highly ranked as H by the voting method, even though the underlying distance measurements may have been much smaller.

That is to say, we would like the hybrid classifier's numeric output to reflect the underlying distance measurements faithfully so that we can compare alternative subdivisions of the input and choose amongst them at higher levels of recognition. This suggests that some form of averaging of the results would be useful, leading to a second problem, namely that the results of each of the distance functions are incomparable with one another.

Each of the distance functions returns a single real number representing the difference between the input and model symbols. But the distribution of values returned is different for each function, as demonstrated in Figure 4.5 which shows histograms of the values obtained from each distance function when comparing the default MathBrush symbol library against itself. The various difference values therefore cannot be meaningfully combined arithmetically, as the resulting value would make no sense.

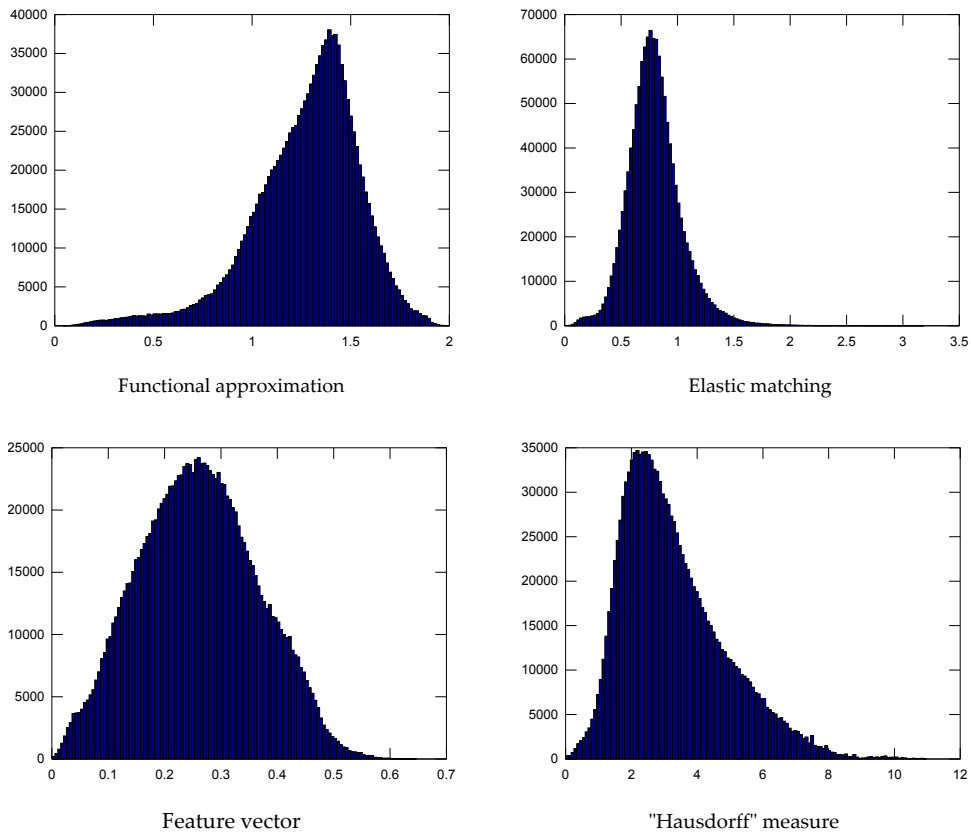


Figure 4.5: Distributions of the four distance measures.

We approximate these distributions in order to obtain comparable values from the four

distance metrics. Given distance functions d_1, \dots, d_n , we empirically model the quantile function Q_i of the distribution of outputs from d_i as a piecewise linear function. Then the distances between input set I and model set M are combined by the weighted sum

$$D(I, M) = \sum_{i=1}^n w_i Q_i(d_i(I, M)). \quad (4.6)$$

Let $L(\alpha)$ be the set of model symbols for a particular symbol α . Then the overall hybrid distance measurement for an input group I to be recognized as α is given by

$$D_\alpha(I) = \min \{D(I, M) : M \in L(\alpha)\}. \quad (4.7)$$

The quantile functions normalize the distances onto $[0, 1]$ in a way that accurately reflects the behaviour of the underlying distance functions. The weights w_i were found as the least-square error solution to an overdetermined system of equations, each of the form

$$\sum_{i=1}^n w_i Q_i(d_i(a_p, a_q)) = m(a_p, a_q).$$

In these equations, each a_i variable indicates a single-stroke symbol from the training set. Each pair of such strokes yields an equation of this form. We set $m(a_p, a_q) = 0$ if a_p and a_q have the same label, otherwise $m(a_p, a_q) = 1$.

4.3 Evaluation

The symbol classifier was evaluated using the same data set as the grouping system (see Section 3.3) using the same five divisions of the data into training and testing halves. Each of the training sets were further augmented with a smaller collection of training samples drawn from the MathBrush symbol library. During testing, each annotated symbol in each testing file was extracted and passed to the classifier, so no grouping ambiguity was possible.

After classification, each symbol in the testing data falls into one of three categories:

1. **Correct:** the symbol is classified correctly,
2. **Ranked:** it is classified incorrectly, but the correct symbol is in the three most highly-ranked results, or
3. **Incorrect:** the correct result is not included in the top three results.

As well as reporting results for each of the four distance-based classifiers and the hybrid classifier, we also measured the proportion of symbols for which all four distance-based classifiers were correct (“All”), and for which at least one of those classifiers was correct

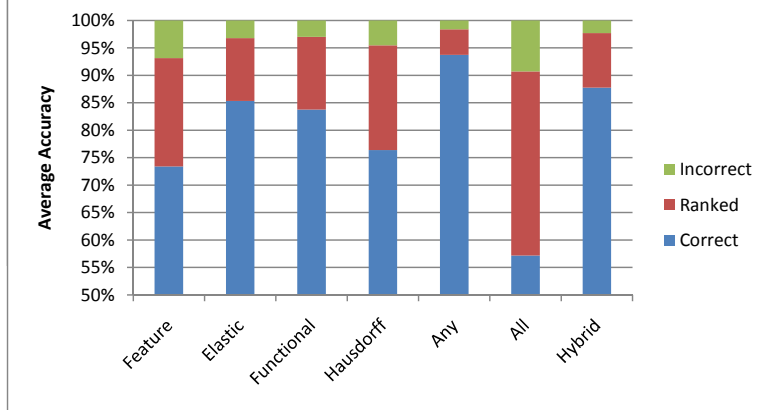


Figure 4.6: Symbol recognition accuracy.

(“Any”). These measurements roughly correspond to lower and upper bounds on the performance of the hybrid classifier. Figure 4.6 summarizes these results.

The elastic and functional approximation matchers are clearly the most accurate of the individual recognizers, obtaining average correctness rates of 85.3% and 83.8%, respectively, while the other two matchers only correctly recognized about 75% of the test symbols. Despite this gap, we found that removing either of the two weaker distance functions from the hybrid recognizer led to less accurate classification. While the elastic matcher attained a higher correctness rate, the functional approximation matcher’s ranked accuracy of 97% was slightly higher than that of the elastic matcher.

The hybrid recognizer is an effective combination of the individual classifiers. In these tests, it had a correctness rate of 87.8% (a 16% reduction in errors over the elastic matcher) and a ranked rate of 97.7% (a 23% reduction in ranked error over the functional approximation matcher). Yet more improvement is possible: a further 50% reduction in errors is necessary for the hybrid recognizer to equal the “Any” matcher which counts a classification as correct if any of the four individual classifiers were correct.

A modification that might further improve the accuracy of the hybrid classifier is to make the quantile combination weights a function of the symbol being matched against. For example, if a particular distance function is known to be especially effective for recognizing the symbol β , then it could be weighted higher for that particular symbol, but not for others. We currently lack sufficient data for training the hybrid recognizer to this level of detail.

Chapter 5

Relation classification

The role of the relation classification system is to determine which spatial relation, if any, joins two subsets of the input strokes. These relations are what connect symbols and subexpressions together to form larger mathematical structures. We use five relations, which we denote by \nearrow , \rightarrow , \searrow , \downarrow , and \odot . The arrows indicate a general direction of writing, while \odot denotes containment. For example, the symbols in the multiplication ab are joined by \rightarrow , in the exponentiation e^x by \nearrow , in the fraction $\frac{1}{2}$ by \downarrow , and in the square root \sqrt{x} by \odot .

Which relation applies to a pair of subexpressions depends primarily on the placement of one relative to the other. As such, many of the features of the input considered in this section will be based on the geometric bounding boxes of the subexpressions. But as well as these purely geometric features, our relation classifier takes into consideration the mathematical semantics of the subsets. For example, the relation joining the symbols in Figure 5.1 should be \nearrow if the p is lower-case, but \rightarrow if the P is upper-case. We currently limit consideration of subset semantics to primarily symbol-level information. The semantic interpretation of a subset is represented by an element of $\Sigma_R = \Sigma \cup C \cup \{\text{GEN}, \text{SYM}, \text{EXPR}\}$, where Σ is the set of recognizable math symbols (e.g. $a, 6, \gamma, +, f$), EXPR indicates a multi-symbol math expression, SYM indicates any single symbol, and GEN is a generic label which may be applied to any expression. C is a set of class labels, each of which corresponds to a family of symbols that share size and placement characteristics. Table 5.1 summarizes the current set of class labels, giving examples of each.



Figure 5.1: The correct choice of relation depends on symbol identity.

The semantic labels in Σ_R may be organized hierarchically with respect to specificity, as illustrated by Figure 5.2. Any single-symbol subexpression may be described not only by its symbol label, but also by any class labels in C that apply to that symbol, as well

Class name	Example symbols
Baseline	a c x
Ascender	2 A h
Descender	g y
Extender	()
Centered	+ =
i	i
j	j
Large-Extender	\int
Root	$\sqrt{\quad}$
Horizontal	-
Punctuation	. , '

Table 5.1: Relational class labels.

as SYM and GEN. Similarly, any multi-symbol subexpression may be described by either EXPR or GEN.

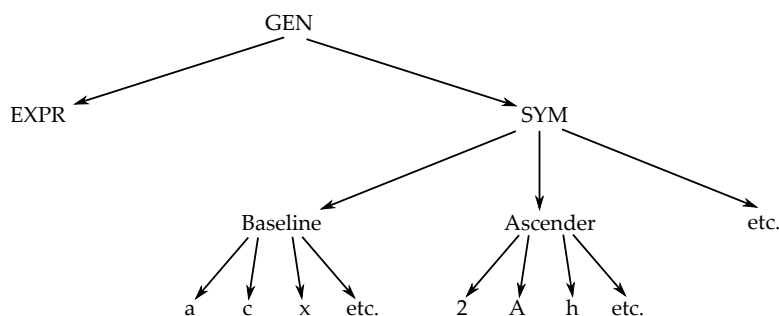


Figure 5.2: Hierarchical organization of semantic labels.

In general, we wish to classify relations using the most specific information available. But in some cases this may not be possible due to a lack of training data. (E.g., we may not have enough training examples of the relation \nearrow between A and θ to make a meaningful judgement about how well a pair of stroke subsets match that relation.) In these cases, we progressively fall back to less and less specific semantic labels until a pair is found for which there is sufficient training data. Labeling both subsets as GEN (i.e., as anything at all) is always permitted in case no combination of more specific labels has sufficient data.

The rest of this section will describe and evaluate four novel methods for classifying relations. Throughout, the pairs $I_1 = (S_1, e_1)$ and $I_2 = (S_2, e_2)$ represent the inputs to the relation classifier, where S_i is set of strokes, and e_i is the semantic label of S_i .

5.1 Rule-based approach

In the rule based approach, we treat the geometric relations as fuzzy relations and define a membership function $\mu_r(I_1, I_2)$ for each relation $r \in R$.

5.1.1 Fuzzy sets review

Recall that a *fuzzy set* \tilde{X} is a pair (X, μ) , where X is some underlying set and $\mu : X \rightarrow [0, 1]$ is a function giving the *membership grade in \tilde{X}* of each element $x \in X$. A *fuzzy relation* on X is a fuzzy set $(X \times X, \mu)$. The notions of set union, intersection, Cartesian product, etc. can be similarly extended to fuzzy sets. For details, refer to Zadeh [37]. To denote the grade of membership of a in a fuzzy set \tilde{X} , we will write $\tilde{X}(a)$ rather than referring explicitly to the name of the membership function, which is typically left unspecified. By $x \in \tilde{X} = (X, \mu)$, we mean $\mu(x) > 0$.

5.1.2 Relation membership functions

The membership function for the containment relation only considers the amount of overlap between the bounding boxes of S_1 and S_2 :

$$\odot((S_1, e_1), (S_2, e_2)) = \text{overlap}(S_1, S_2),$$

where $\text{overlap}(S_1, S_2)$, as in the grouping case, gives the intersection area of the bounding boxes of S_1 and S_2 divided by the area of the smaller of the two.

The membership functions for the other geometric relations incorporate the distance and angle between the bounding boxes of the input. They are all of the form

$$r((S_1, e_1), (S_2, e_2)) = \theta((S_1, e_1), (S_2, e_2)) \times d(S_1, S_2) \times \left(1 - \frac{1}{2} \text{overlap}(S_1, S_2)\right),$$

where θ is a scoring function based on angle and d is a distance-based penalty function.

The penalty function d ensures that inputs satisfying the relations are within a reasonable distance of one another. To compute d , the size of each of S_1 and S_2 is calculated as the average of bounding box width and height. Next, a distance threshold t is obtained as half the average of the two sizes, clamped to the range $[1/6, 1/3]$ (measured in inches). If the distance Δ between the bounding boxes is less than t , then $d = 1$. Otherwise, d decreases linearly to zero at $\Delta = 3t$. Note that this hard limit implies that the rule-based approach will not classify two sets of input strokes as related if their bounding box centroids are more than one inch apart.

To compute the angle-based function θ , we measure the angle φ between bounding-box centroids of I_1 and I_2 . Then θ is a triangular function with three parameters $\varphi_0, \varphi_1, \varphi_2$, as follows:

$$\theta(\varphi) = \begin{cases} 0 & \text{if } \varphi < \varphi_0 \\ \frac{\varphi - \varphi_0}{\varphi_1 - \varphi_0} & \text{if } \varphi_0 \leq \varphi \leq \varphi_1 \\ \frac{\varphi_2 - \varphi}{\varphi_2 - \varphi_1} & \text{if } \varphi_1 \leq \varphi \leq \varphi_2 \\ 0 & \text{if } \varphi > \varphi_2. \end{cases}$$

Figure 5.3 schematically illustrates the behaviour of θ . Table 5.2 lists the parameters for each relation in degrees. These values were selected based on manual examination of a small dataset. Note that the y -axis increases downward.

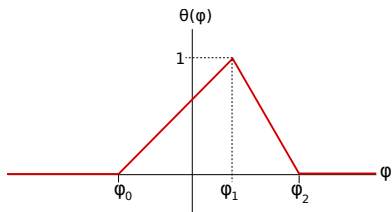


Figure 5.3: θ function component of relation membership grade.

Relation	φ_0	φ_1	φ_2
\rightarrow	-90	0	90
\nearrow	-90	-37.5	0
\searrow	-20	35	160
\downarrow	0	90	180

Table 5.2: Angle thresholds for geometric relation membership functions.

5.2 Naive Bayesian model

Generative probabilistic models generally model the relationship between a dependent variable X and several observable feature variables F_1, \dots, F_n by applying Bayes' theorem:

$$P(X | F_1, \dots, F_n) = P(F_1, \dots, F_n | X) \frac{P(X)}{P(F_1, \dots, F_n)}.$$

Naive Bayesian models further assume that the feature variables F_i are independent so that the RHS above may be factored as

$$\frac{P(X)}{P(F_1, \dots, F_n)} \prod_{i=1}^n P(F_i | X).$$

We created a naive Bayesian model for relation classification using the following feature variables. Let $\ell(S), r(S), t(S), b(S)$ denote the left-, right-, top-, and bottom-most coordinates of a set S of strokes. Then, given our two input subsets S_1 and S_2 , the features

are:

$$\begin{aligned}
 f_1 &= \frac{\ell(S_2) - \ell(S_1)}{N} \\
 f_2 &= \frac{r(S_2) - r(S_1)}{N} \\
 f_3 &= \frac{\ell(S_2) - r(S_1)}{N} \\
 f_4 &= \frac{b(S_2) - b(S_1)}{N} \\
 f_5 &= \frac{t(S_2) - t(S_1)}{N} \\
 f_6 &= \frac{t(S_2) - b(S_1)}{N} \\
 f_7 &= \text{overlap}(S_1, S_2)
 \end{aligned}$$

where $N = aR + (1 - a)C$ is a normalization scale based on the size

$$R = \max \{ \text{size}(S_1), \text{size}(S_2), 1/16 \}$$

of the input sets (measured in inches) and a constant $C = 1$ inch. The size of an input set S is given by

$$\text{size}(S) = \max (r(S) - \ell(S), b(S) - t(S)),$$

and we use $a = 1/2$.

For each relation $r \in R$ and each pair of semantic labels e_1, e_2 ,

$$P(X = r \mid e_1, e_2, f_1, \dots, f_7) \propto P(X = r \mid e_1, e_2) \prod_{i=1}^7 P(f_i \mid e_1, e_2, X = r). \quad (5.1)$$

We model each conditional distribution $f_i \mid e_1, e_2, X$ as Gaussian, while the relation class prior $X \mid e_1, e_2$ is categorical. The parameters of these distributions are estimated from training data. Note that the entire model is conditioned on semantic labels, so we have a different set of distributions for each combination of labels. The feature variables f_i were elided from the computation since $P(F_1, \dots, F_7 \mid e_1, e_2)$ is constant with respect to the relation variable X .

Given the input $(S_1, e_1), (S_2, e_2)$, we need only look up the appropriate set of distributions (based on e_1, e_2), and compute the product in Equation 5.1. If any of the distributions referenced in the product have less than 95% confidence that the true mean was within a small distance (0.05) from the estimated mean, then the calculation instead returns failure, and a less-specific pair of semantic labels will be used, as described above.

5.3 Discriminative model

Unlike generative models, discriminative models need not appeal to Bayes' theorem, as they model the probability $P(X \mid F_1, \dots, F_n)$ directly. We developed two discriminative

methods for relation classification, both using the same feature variables f_1 through f_7 as described in the previous section.

5.3.1 Discretization grid

The first discriminative model is a straightforward grid-based discretization strategy. The model was trained using the following process for each pair e_1, e_2 of semantic labels:

1. Fix a number N of grid cells per dimension. (We used $N = 5$ based on experiments, giving a total of $5^7 = 78125$ grid cells for our seven-dimensional data points.)
2. Divide the training set into outliers and non-outliers. (We used Tukey’s outlier test [11].)
3. Divide the inner $N - 2$ grid cells in each dimension equally-spaced between the minimum and maximum values appearing in non-outlier data points in each dimension. Reserve the two outer grid cells in each dimension for outliers.
4. Assign each input point to the grid cell containing it.

Then to determine $P(X = r \mid e_1, e_2, f_1, \dots, f_n)$, we need only find the cell C containing the point (f_1, \dots, f_n) in the grid corresponding to (e_1, e_2) , and compute the ratio

$$\frac{\text{number of points in } C \text{ corresponding to relation } r}{\text{number of points in } C}.$$

This method may be seen as a fast approximation of the k -nearest neighbour distribution. As in the generative case, a provision for insufficient data must be made so that a less-specific query may be made. In this case, the calculation simply returns failure if the grid cell C contains fewer than two points.

5.3.2 Margin trees

Margin trees are a machine-learning-inspired variant of the k -d tree data structure. They are a type of decision tree which partitions the feature space into hypercubes, each of which offers an approximation to the distribution $X \mid F_1, \dots, F_n$. k -d trees themselves are an extension of binary trees to k -dimensional data. Given a set P of points from \mathbb{R}^k (with each $p \in P$ indexed by dimension as (p_1, p_2, \dots, p_k)), Algorithm 3 constructs a k -d tree:

Algorithm 3 splits the input points at the median point along the first splitting dimension, then splits each of those halves at their median points along the second splitting dimension, and so on, forming a tree structure of the split points. Algorithm 4 demonstrates how to find the tree node covering a query point q .

To see how this applies to our probability computations, consider each training example as a tuple (r, f_1, \dots, f_7) , where $r \in R$ is a relation and the f_i are the geometric features

Algorithm 3 k -d tree construction.

Require: input point set $P = \{p^{(1)}, \dots, p^{(n)}\}$ as above, current tree depth D , and a set $S \subseteq \{1, \dots, k\}$ of splitting dimensions

if $n = 1$ **then**

return leaf node with point $p^{(1)}$

$d \leftarrow D \bmod |S|$

$s \leftarrow$ the d th-smallest element in S

$M \leftarrow$ median of $\{p_s^{(i)} : i = 1, \dots, n\}$

create an internal node N with split value M

recurse with $P = \{p \in P : p_s \leq M\}$, $D = D + 1$ to find left child of N

recurse on $P = \{p \in P : p_s > M\}$, $D = D + 1$ to find right child of N

return N

Algorithm 4 k -d tree cell search.

Require: k -d tree node N , query point $q = (q_1, \dots, q_k)$, current tree depth D , and a set $S \subseteq \{1, \dots, k\}$ of splitting dimensions

if N is a leaf node **then**

return N

$d \leftarrow D \bmod |S|$

$s \leftarrow$ the d th-smallest element in S

if $q_s \leq$ the split value for N **then**

 recurse on the left child of N with $D = D + 1$

else

 recurse on the right child of N with $D = D + 1$

described in Section 5.2. Consider constructing a k -d tree for these tuples, splitting along the dimensions corresponding to the geometric features.

After the tree is constructed, the search algorithm (Algorithm 4) may be queried with a point $F \in \mathbb{R}^7$ containing features extracted from an input. As the search algorithm is running, an approximation to $P(F)$ may be obtained at any tree node N that the algorithm visits by counting the number of children rooted at N and dividing by the number of nodes in the entire tree. The discretization scale becomes finer and finer as the search procedure descends into the tree. Similarly, an approximation to $P(r | F)$ can be obtained at any visited node N by counting the number of children rooted at N labeled with relation r in the appropriate tuple dimension, then dividing by the number of nodes in the subtree rooted at N .

Every node in the k -d tree thus approximates the distribution $X | F_1, \dots, F_7$ in the hypercube neighbourhood covered by the node. This observation is one of the two key ideas on which the margin tree data structure is based.

The second key idea behind margin trees addresses a potential mismatch between the nature of the data being represented and the data partitioning strategy used in the k -d tree algorithms. By splitting the data at the median point, those algorithms achieve optimal runtime efficiency, and cycling through the splitting dimensions provides a simple default behaviour. However, this strategy is not always best for generating accurate function approximations.

Intuitively, we expect each of the tuples in our database to fall into one of several clusters, each associated with a particular combination of relation and semantic variables. I.e., we expect most of the tuples arising from a \rightarrow relation between symbols 2 and a to be near one another, and rather different from those arising from a \downarrow relation between symbols x and $-$, which form their own cluster. Some of the clusters might overlap significantly; for example, the expressions b^x and px have similar bounding box profiles. But nevertheless many clusters will be quite distinct.

This distinction is ignored by the standard k -d tree algorithms above, which can lead to clusters being split across different tree nodes. For example, the leftmost illustration of Figure 5.4 shows two clusters, coloured orange and blue. The standard median-based k -d tree construction algorithm (Alg. 3) would split the blue cluster across both halves of the data, as shown in the central illustration of the Figure. A more appealing subdivision of the data which preserves the structure of its clusters is shown in the right-most illustration.

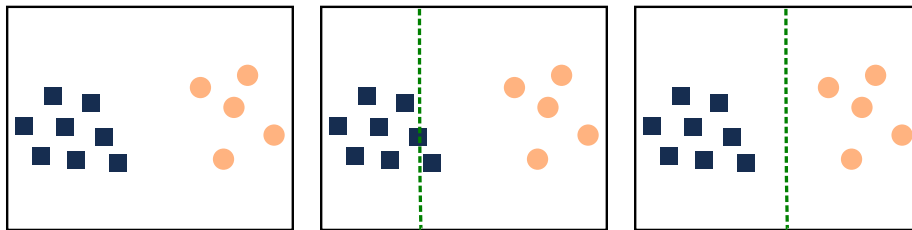


Figure 5.4: Data clusters and splitting possibilities.

Preserving the cluster-based structure of the data segregates distinct clusters into different tree nodes. Since the tree node split points determine the neighbourhoods in which we compute different approximations of the probability distributions of interest, this segregation effectively divides up the search space into neighbourhoods that more accurately reflect the natural groupings of data points, rather than the arbitrary neighbourhoods obtained by splitting the data in half.

To accomplish this cluster separation, we rely on a basic principle from machine learning: margin maximization. Given two sets of data points X and Y , and a curve dividing all the points in X from those in Y , the margin is defined as the minimal distance from any data point to the curve. Choosing the curve with the largest margin yields the “best” classifier in the sense that the classifier has minimal generalization error: under certain theoretical conditions, it makes the fewest errors when classifying previously-unseen data.

Therefore, rather than splitting the data at the median point along an alternating dimension, we shall split the data in such a way that some quantity resembling the margin is maximized at each step in the algorithm. This does not strictly constitute a maximum-margin classifier since we are not explicitly segregating all of the distinctly-labeled data into distinctly-labeled clusters (this is often impossible with the relation data we are using). However, we are subdividing the data in such a way that a query point is likely to be mapped to a tree node containing data from exactly the clusters to which the query point is most likely to belong, and not data from some other clusters which were included in the tree node by chance.

Margin tree algorithms

Having described the intuition behind margin trees: hierarchical max-margin-based partitioning of approximation neighbourhoods, we can now give explicit algorithms for constructing and querying margin trees. Algorithm 5 describes the construction process, while Algorithm 6 shows how to compute a probability of the form $P(X | f_1, \dots, f_n)$. The notation $\text{size}(N)$ denotes the number of margin tree nodes rooted at N (including N itself).

The constants C_1 and C_2 in Algorithm 6 control the granularity of probability approximations. If a tree node was built from fewer than C_1 training samples, it is deemed too sparsely populated and is not used. Cells constructed from between C_1 and C_2 samples are deemed to be sufficiently fine-grained and well-populated, and are used for probability estimation. Cells containing more than C_2 samples are considered too coarse-grained and are recursively searched for more finely-grained approximation neighbourhoods. In our experiments, we used $C_1 = 16$ and $C_2 = 96$.

To put these algorithms to practical use, we create a margin tree for each pair (e_1, e_2) of semantic labels appearing in the training set; each tree is constructed from all the relevant tuples (r, f_1, \dots, f_7) extracted from the training data, splitting on the feature variables, but not the relation. To compute a probability $P(X = r | e_1, e_2, f_1, \dots, f_7)$, Algorithm 6 is invoked for the tree corresponding to the semantic labels e_1, e_2 using query point (r, f_1, \dots, f_7) , with query dimension set to the relation.

Algorithm 5 Margin tree construction.

Require: An input point set $P = \{p^{(1)}, \dots, p^{(n)}\}$, and a set $S \subseteq \{1, \dots, k\}$ of splitting dimensions

if $n = 1$ **then**

return leaf node with point $p^{(1)}$

Choose a splitting dimension $d \in S$ and a split point s such that $p_d^{d,s+1} - p_d^{d,s}$ is maximal, where $p^{i,j}$ is the j th point in P when ordered along dimension i

$M \leftarrow (p_d^{d,s} + p_d^{d,(s+1)}) / 2$

Create an internal node N with split dimension d and point $p^{d,s}$

Recurse with $P = \{p \in P : p_d < M\}$ to construct the left child of N

Recurse on $P = \{p \in P : p_d > M\}$ to construct the right child of N

return N

Algorithm 6 Margin tree query.

Require: A k -d tree node N , query point $q = (q_1, \dots, q_k)$, a set $S \subseteq \{1, \dots, k\}$ of splitting dimensions, and a set $Q \subseteq \{1, \dots, k\} \setminus S$ of query dimensions

if $\text{size}(N) < C_1$ **then**

return “insufficient data”

if $\text{size}(N) < C_2$ **then**

 Let m be the number of nodes rooted at N with data points matching q on all query dimensions

return $m/\text{size}(N)$

Let d be the splitting dimension for N

if $q_d \leq$ the split value for N **then**

 Recurse on the left child of N

else

 Recurse on the right child of N

This query will either return an approximation to $P(X = r \mid e_1, e_2, f_1, \dots, f_7)$ in a neighbourhood of the features judged to be appropriate, or it will fail, in which case the query may be repeated with a less specific pair of semantic labels.

The problem of outliers

As constructed by Algorithm 5, margin trees do not handle outlier data points appropriately. Suppose the set of n input points includes a point p which lies far from the rest of the points along dimension d . Then the large gap between p and the other points will cause the algorithm to partition p into its own tree node splitting the data along dimension d , and will recurse on the remaining $n - 1$ points. This construction causes a significant slowdown in the query algorithm, which must search through chains of linearly-structured nodes arising from outliers before reaching nodes that correspond to true data clusters.

Unlike the grid-based method, a margin tree has no natural way of accounting for outliers. So rather than filtering and segregating outliers in their own discretization bin, we adjust the split selection criteria in Algorithm 5 so that it balances between choosing the max-margin and median points. In particular, we replace the maximization of $p_d^{d,s+1} - p_d^{d,s}$ with the maximization of a weight function $w(s/n, p^{d,s}, p^{d,s+1}, d)$. Based on experimentation, we chose

$$w(x, p^{(1)}, p^{(2)}, d) = \omega(p^{(1)}, p^{(2)}) (1 - 4(x - 1/2)^2) (p_d^{(2)} - p_d^{(1)}).$$

The third factor is the same as in Algorithm 5. The second weights each potential split point based on its position in the sorted point list; the median point is assigned the highest weight, and weights tail off to zero at the list’s extremities. The first factor $\omega(p^{(1)}, p^{(2)})$ is another weight function measuring the difference in the query variables (i.e., the dimensions not used for splitting – for our application, the clustering variables r, e_1, e_2):

$$\omega(p^{(1)}, p^{(2)}) = \frac{1 + \sum_{i \notin S} 1 - \delta(p_i^{(1)}, p_i^{(2)})}{1 + k - |S|}.$$

ω assigns higher weights to pairs of points belonging to “more different” clusters, and lower weights to points belonging to the same cluster.

The weight function w thus prefers to split the data at the maximum margin position, by weighting in favour of large gaps from different data clusters. But it balances this preference against a second preference to split the data at the median point for runtime efficiency.

5.4 Evaluation

To evaluate the efficacy of the models proposed above, we trained each of them on one half of the 2009 data and tested them on the other half, using the same five random splits as in

the grouping and symbol recognition evaluations. For each test, the relation maximizing each model’s scoring function was chosen as the model’s classification result. If any of the relation queries for a model reported failure due to insufficient data, the semantic labels e_1 and e_2 were degraded to less specific values, and the queries were retried. Each test is therefore correct or incorrect based on whether the reported relation matches the ground-truth relation, with no partial correctness possible.

Figure 5.5 shows the correctness rates of each classifier on the 2009 data. To account for potential overtraining, we also evaluated each classifier on the Tablet PC data collected in a 2011 study [21], using the same five training sets as in the first experiment. Figure 5.6 shows the classifiers’ correctness rates on that data set.

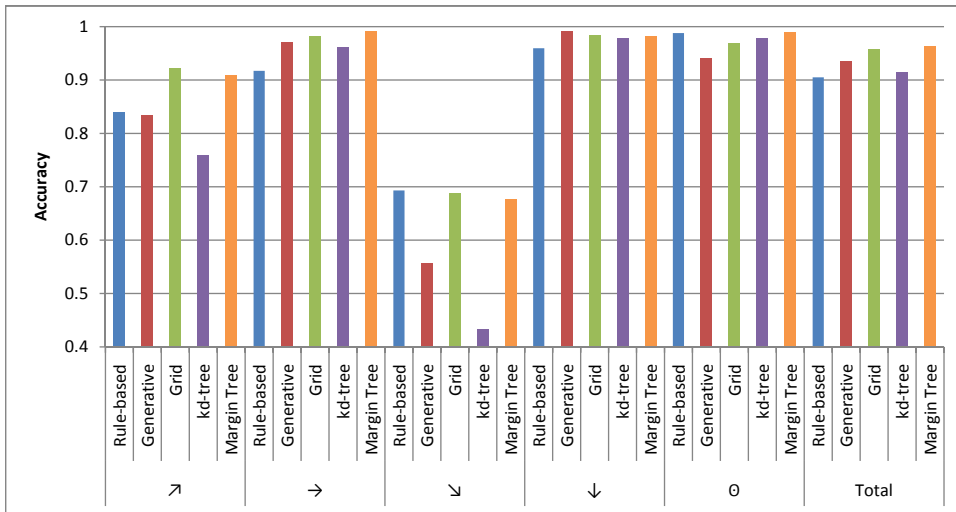


Figure 5.5: Relation classifier accuracy on 2009 data set.

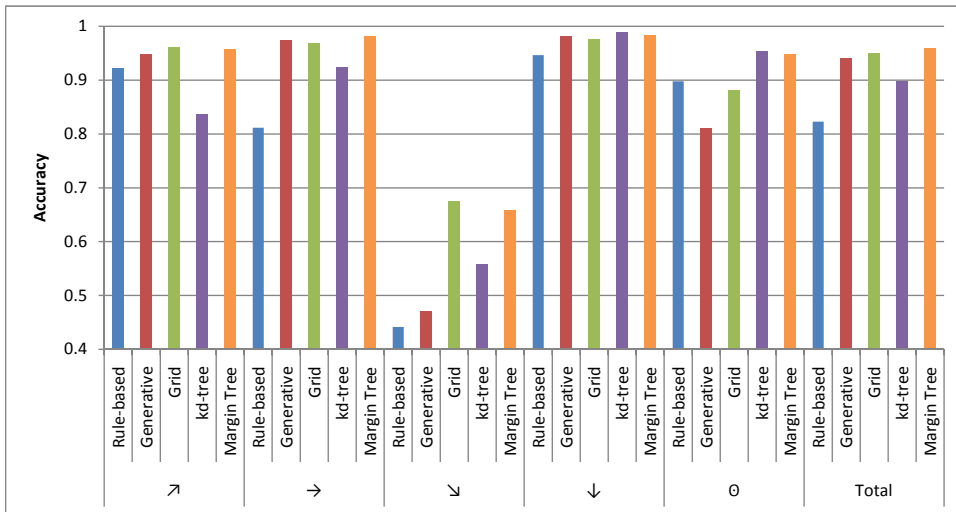


Figure 5.6: Relation classifier accuracy on 2011 data set.

The margin tree and grid-based implementations were the most accurate in both data sets, and neither seemed to suffer from overtraining on the 2009 data set. The generative

model was also quite accurate, but had significantly poorer performance for the \searrow and \odot relations in the 2011 data set, as compared with the 2009 data; however, its overall accuracy was higher on the 2011 data due to improved accuracy on the more common \nearrow relation.

Note that, although the discriminative models were most accurate in this isolated testing scenario, we use the Naive Bayes classifier in practice in the MathBrush recognizer because it performs better under more realistic conditions (see Section [8.2.4](#)).

Chapter 6

Relational grammars

The relational grammar formalism described in this chapter is the central component governing how the MathBrush recognizer assigns precise mathematical meaning to its noisy, ambiguous input. The recognition grammar is designed in such a way that particular productions correspond to particular real-world mathematical semantics. When the recognizer finds a plausible parse of a given production on an input subset, it tentatively associates the production’s semantics with that subset. This tentative process and how the semantic associations are made specific will be discussed in detail in the next chapter. This chapter describes the formal grammar model developed for the MathBrush recognizer.

In previous work, we described a fuzzy relational grammar formalism which explicitly modeled recognition as a process by which an observed, ambiguous input is interpreted as a certain, structured expression [22]. In this work, because we wish to apply both fuzzy and probabilistic scoring techniques to the recognition system, the grammar is defined more abstractly so that it only describes syntactic layout and has no intrinsic support for one scoring technique or another.

6.1 Preliminary grammar definition

We divide our grammar definition into two parts. Definition 1 outlines the primary structure of a relational context-free grammar (RCFG).

Definition 1. *A relational context-free grammar G is a tuple (Σ, N, S, O, R, P) , where*

- Σ is a set of terminal symbols,
- N is a set of non-terminal symbols,
- $S \in N$ is the start symbol,
- O is a set of observables,
- R is a set of relations on I , the set of interpretations of G (defined below),

- P is a set of productions, each of the form $A_0 \xrightarrow{r} A_1 A_2 \cdots A_k$, where $A_0 \in N$, $r \in R$, and $A_1, \dots, A_k \in N \cup \Sigma$,

This definition is similar to that of a traditional one-dimensional CFG. After discussing the differences, we will augment the definition with detailed linkage parameters which serve primarily as parsing aids.

6.1.1 Observables and partitions

The set O of *observables* represents the set of all possible concrete inputs. Each $o \in O$ is a set of ink strokes, each tracing out a particular curve in the (x, y) plane.

Denote by $T(o)$ the set of *recursive partitions* of an observable $o \in O$, defined as follows.

If $|o| = 1$, then $T(o) = \{\{o\}\}$. Otherwise, $T(o)$ is defined inductively as

$$T(o) = \bigcup_{n=1}^{|o|} T_n(o),$$

where $T_n(o)$ is the set of recursive partitions of o having an n -ary top-level partition:

$$T_n(o) = \bigcup_{\substack{o_1 \cup \cdots \cup o_n = o \\ o_i \cap o_j = \{\}, i \neq j}} \{(t_1, \dots, t_n) : t_i \in T(o_i)\}.$$

Each recursive partition $t \in T(o)$ represents a tree in which each node corresponds to a subset o' of o and has links to nodes which together form a partition of o' . In many cases, we will use a recursive partition t of o and o itself interchangeably. In these situations, t is understood to represent o ; i.e., the recursive disjoint union of the components of t . Finally, $T = \bigcup_{o \in O} T(o)$ is the set of all possible recursive partitions.

6.1.2 Relations

The set R contains relations that give structure to expressions by modeling the relationships between subexpressions. These relations are the same as those described in Chapter 5. Recall that the relations use semantic labels as well as geometric information so that expressions like the one shown in Figure 6.1 may be interpreted differently based on the identity of its constituent symbols. In the context of RCFGs, we say that the relations act on interpretations (i.e., geometric input paired with semantic labels), allowing context-sensitive statements to be made about recognition in an otherwise context-free setting.

AP

Figure 6.1: An expression in which the optimal relation depends on symbol identity.

6.1.3 Set of interpretations

While typical context-free grammars deal with *strings*, we call the objects derivable from RCFGs *expressions*. Any terminal symbol $\alpha \in \Sigma$ is an expression. An expression e may also be formed by concatenating a number of expressions e_1, \dots, e_k by a relation $r \in R$. Such an *r-concatenation* is written $e_1 r e_2 r \dots r e_k$.

The *representable set of G* is the set E of all expressions derivable from the nonterminals in N using productions in P . It may be constructed inductively as follows:

For each terminal $\alpha \in \Sigma$, $E_\alpha = \{\alpha\}$. For each production p of the form $A_0 \xrightarrow{r} A_1 \dots A_k$,

$$E_p = \{e_1 r \dots r e_k : e_i \in E_{A_i}\}.$$

For each nonterminal A ,

$$E_A = \bigcup_{p \in P \text{ having LHS } A} E_p.$$

Finally,

$$E = \bigcup_{A \in N} E_A.$$

An *interpretation* is a pair (e, t) , where $e \in E$ is an expression and t is a recursive partition in $T(o)$ for some observable $o \in O$. An interpretation is essentially a parse tree of o where the structures of e and t encode the hierarchical structure of the parse. The set of interpretations I referenced in the RCFG definition is just the set of all these interpretations:

$$I = \bigcup_{o \in O} \{(e, t) : e \in E, t \in T(o)\}.$$

6.1.4 Productions

The productions in P are similar to context-free grammar productions in that the left-hand element derives the sequence of right-hand elements. The relation r appearing above the \Rightarrow production symbol indicates a requirement that r is satisfied by adjacent elements of the RHS. Formally, given a production $A_0 \xrightarrow{r} A_1 A_2 \dots A_k$, if o_i denotes an observable interpretable as an expression e_i derivable from A_i (i.e., $e_i \in E_{A_i}$ and $(t_i, e_i) \in I$), then for $\bigcup_{i=1}^k t_i$ to be interpretable as $(e_1 r \dots r e_k)$ requires $((e_i, t_i), (e_{i+1}, t_{i+1})) \in r$ for $i = 1, \dots, k - 1$.

6.2 Examples

The following examples demonstrate how RCFG productions may be used to model the structure of mathematical writing.

1. Suppose that [ADD] and [EXPR] are nonterminals and + is a terminal. Then the production $[ADD] \xrightarrow{\rightarrow} [EXPR] + [EXPR]$ models the syntax for infix addition: two expressions joined by the addition symbol, ordered from left to right.
2. Similarly, the production $[PAREN] \xrightarrow{\rightarrow} ([EXPR])$ models the syntax for parenthesized expressions.
3. The production $[FRAC] \xrightarrow{\downarrow} [EXPR] \text{---} [EXPR]$ models a fraction: a numerator, fraction bar, and denominator ordered from top to bottom.
4. The production $[SUP] \xrightarrow{\nearrow} [EXPR] [EXPR]$ models superscript syntax. Interpreted as exponentiation, the first RHS token represents the base of the exponentiation, and the second represents the exponent. The tokens are connected by the \nearrow relation, reflecting the expected spatial relationship between subexpressions.
5. The following pair of productions models the syntax of definite integration:

$$\begin{aligned}
 [\text{LIMITS}] &\xrightarrow{\downarrow} [EXPR] \int [EXPR] \\
 [\text{INTEGRAL}] &\xrightarrow{\rightarrow} [\text{LIMITS}] [EXPR] d[\text{VAR}]
 \end{aligned}$$

Definite integrals are drawn using two writing directions. The limits of integration and integration sign itself are written in a vertical stack, while the integration sign, integrand, and variable of integration are written from left to right.

6.3 Linkage parameters

It is often convenient to specify what portion of an expression a relation should apply to. For example, in the expression

$$\int_0^{\infty} e^{-x^2} dx,$$

we may want the \rightarrow relation between the limits of integration and the integrand to apply to the symbols \int and e , and the \nearrow relation between e and its exponent to apply to e and $-x$. Yet in

$$\int_0^1 \frac{x^2 + 1}{\sqrt{x^2 - 1}} dx,$$

we may want the \rightarrow relation between the limits of integration and the integrand to apply to the \int symbol and the fraction as a whole, and the \downarrow relations linking the numerator to the

fraction bar and the fraction bar to the denominator to apply to complete subexpressions as well.

This flexibility is made possible by augmenting grammar productions with *head*, *tail*, and *linkage* parameters. Define the set $S = \{\text{SYM}, \text{GRP}, \text{DEF}\}$ of linkage specifiers corresponding to the notions “symbol”, “group” and “default”, respectively. Then we augment the grammar definition by adding the following components to the tuple:

1. $H : P \rightarrow \mathbb{N}$ is a function mapping each production to the RHS element which is its *head*,
2. $T : P \rightarrow \mathbb{N}$ is a function mapping each production to the RHS element which is its *tail*,
3. $L : P \times \mathbb{N} \rightarrow S^2$ is a function defining the *linkage specifier* at each position of each production, and
4. $L_0 : P \rightarrow S$ is a function specifying the *parent linkage* of each production.

Let p be a production $A_0 \xrightarrow{r} A_1 \cdots A_k$. Then p has a *tail* $T(p)$ which is the RHS component from which outgoing relations are tested, and a *head* $H(p)$ which is the component to which incoming relations are tested.

For example, in the production for addition above, we might choose the first [EXPR] on the RHS as the expression’s head, and the second (final) [EXPR] as the expressions tail. But in the production of [ILIMITS], we might choose the integral sign as both the head and tail so that when a parse of [ILIMITS] is included in a parse of [INTEGRAL], the \rightarrow relation joining the limits of integration to the integrand is tested against the integral sign and not against some other part of that subexpression.

The linkage function $L(p, i)$ gives a pair (ℓ_i, ℓ_{i+1}) of linkage specifiers controlling how a relation between A_i and A_{i+1} should be tested. Consider a relation $r((e_i, t_i), (e_{i+1}, t_{i+1}))$ as being *from* (e_i, t_i) and *to* (e_{i+1}, t_{i+1}) . The head and tail pointers of the productions used to obtain those two interpretations may be followed recursively until individual symbols are reached. These symbols are called the head and tail symbols of the expression.

The from and to components with which the relation r is queried are chosen based on the linkage specifiers as follows:

1. If $\ell_i = \text{SYM}$, then the from component is given by the tail symbol of the interpretation (e_i, t_i) . If $\ell_{i+1} = \text{SYM}$, then the to component is given by the head symbol of the interpretation (e_{i+1}, t_{i+1}) .
2. If $\ell_i = \text{GRP}$, then the from component is the entire interpretation (e_i, t_i) ; similarly if $\ell_{i+1} = \text{GRP}$, then the to component is the entire interpretation (e_{i+1}, t_{i+1}) .
3. If $\ell_i = \text{DEF}$, then the from component is selected based on the parent linkage $L_0(p_i)$ of the production p_i with LHS A_i used to obtain the interpretation (e_i, t_i) . Similarly, if $\ell_{i+1} = \text{DEF}$, then the to component is selected based on the parent linkage $L_0(p_{i+1})$ of the production p_{i+1} with LHS A_{i+1} used to obtain the interpretation (e_{i+1}, t_{i+1}) .

The parent linkage $L_0(p)$ of a production p determines how a relation from or to an expression derived from p is tested in the case where that expression appears as a subexpression in some larger expression using the DEF linkage specifier (i.e., item 3 above). As such, at any given point in a parse, the parent linkage of p determines either the from or to component of a relation query, as follows:

1. If $L_0(p) = \text{SYM}$, then the tail (head) symbol of the interpretation of p is used as the from (to) component.
2. If $L_0(p) = \text{GRP}$, then the entire interpretation of p is used as the from (to) component.
3. If $L_0(p) = \text{DEF}$, then the from (to) component is obtained recursively based on the parent linkage $L_0(p^*)$, where p^* is the production with LHS $A_{T(p)}$ ($A_{H(p)}$) used to obtain the interpretation of the tail (head) nonterminal of p .

By default, the head of a production $A_0 \xrightarrow{\tau} A_1 \cdots A_k$ is 1 (i.e., its first RHS element) and the tail is k (i.e., its last RHS element). If linkage parameters are not specified by the grammar, they default to DEF.

6.3.1 Examples

For example, assume the following parameters for the productions in section 6.2.

- the head and tail of the [ADD] and [ILIMITS] productions are as described above, and the parent linkage of [ILIMITS] is SYM;
- the parent linkage of the [FRAC] production is GRP, and within the production, the linkage from [EXPR] to — is (GRP, SYM) and the linkage from — to [EXPR] is (SYM, GRP); and
- the parent linkage of [SUP] is GRP, and the linkage from [EXPR] to [EXPR] within the production is (SYM, GRP).

When deriving the expected parse of the expression in Figure 6.2, relations will be tested as follows:

- The \downarrow relation will be tested between 1 and \int and between \int and 0 when parsing [ILIMITS], as usual.
- When parsing the exponential, the \nearrow relation will be tested between the right parenthesis $)$ and the entire subexpression $x^2 + 1$ because the relevant linkage specifier is (SYM, GRP), and $)$ is the tail symbol of the subexpression (ae) .
- When parsing the fraction, the \downarrow relation will be tested between the entire numerator $(ae)^{x^2+1}$ and the fraction bar, and between the fraction bar and the denominator $x - 1$ because the linkage specifiers are (GRP, SYM) and (SYM, GRP), respectively.

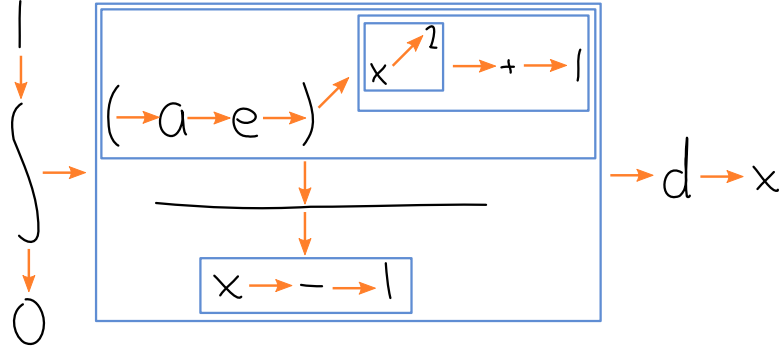


Figure 6.2: An expression demonstrating a variety of subexpression linkages.

- When parsing the entire expression as [INTEGRAL], the \rightarrow relation will be tested between the integral sign and the entire fraction. Because no explicit linkage was set between those two nonterminals in the [INTEGRAL] production, the default specifier will lookup the parent linkages. For [LIMITS], the parent linkage is SYM, so the tail symbol \int is used. For the fraction, the parent linkage is GRP, so the entire subexpression is used.

As these examples demonstrate, the linkage parameters permit very flexible yet precise control over which subexpressions are selected for relation queries during parsing.

6.4 Set of interpretations as a directed set

Given a particular input observable $o \in O$, define the set of interpretations of o as the subset of the grammar's set of interpretations that concern o :

$$I_o = \{(e, t) \in I : t = o\}.$$

It is useful to define notation for further subsets of I_o matching the structure of the representable set of G . For cleaner notation, assume that the grammar productions are in a normal form such that each production is either of the form $A_0 \Rightarrow \alpha$, where $\alpha \in \Sigma$ is a terminal symbol, or of the form $A_0 \xrightarrow{r} A_1 \cdots A_k$, where all of the A_i are nonterminals. This normal form is easily realized by, for each $\alpha \in \Sigma$, introducing a new nonterminal X_α , replacing all instances of α in existing productions by X_α , and adding the production $X_\alpha \Rightarrow \alpha$.

Then, for every observable o , define the following notations.

Definition 2 (Interpretations of o).

1. For every production p of the form $A \Rightarrow \alpha$, define

$$I_o^p = \{(\alpha, t)\}$$

for t the trivial recursive partition $\{o\}$.

2. For every nonterminal A , define

$$I_o^A = \bigcup_{p \text{ having LHS } A} I_o^p.$$

3. For every production p of the form $A_0 \xrightarrow{r} A_1 \cdots A_k$, define

$$I_o^p = \{(e_1 r \cdots r e_k, (t_1, \dots, t_k)) : (e_i, t_i) \in I_{t_i}^{A_i}, i = 1, \dots, k; \\ (t_1, \dots, t_k) \in T_k(o); \\ ((e_i, t_i), (e_{i+1}, t_{i+1})) \in r, i = 1, \dots, k - 1\}.$$

Given a scoring function $\text{sc}(e, t) : I \rightarrow [0, 1]$ on interpretations, we treat these sets of interpretations as directed sets using the preorder $(e_1, t_1) \leq (e_2, t_2) \iff \text{sc}(e_1, t_1) \leq \text{sc}(e_2, t_2)$. (Recall that the preorder for a directed set is reflexive and transitive, but not necessarily antisymmetric; that is $(e_1, t_1) \leq (e_2, t_2)$ and $(e_2, t_2) \leq (e_1, t_1)$ do not together imply that the two interpretations are identical.)

Using these definitions, recognizing an input o can be seen as a process which reports elements of I_o^S (for S the grammar's start symbol) from greatest to least.

6.5 Semantic expression trees and textual output

RCFG productions model the two-dimensional syntax of mathematical expressions. To represent mathematical semantics, each production is also associated with rules for generating textual output and math expression trees with semantic information. For example, consider again the production $[\text{ADD}] \xrightarrow{\rightarrow} [\text{EXPR}] + [\text{EXPR}]$. A rule to generate a MathML string would be written in our grammar format as $\langle \text{mrow} \rangle \%1 \langle \text{mo} \rangle + \langle \text{mo} \rangle \%3 \langle \text{mrow} \rangle$, where the “% n ” notation indicates that the string representation of the n th RHS element should be inserted at that point. A rule to generate a semantic expression tree would be written $\text{ADD}(\%1, \%3)$. This rule would generate a tree with the root node labelled with addition semantics (“ADD”) and two subtrees. Similarly to the string case, the % n notation indicates that the tree representation of the n th RHS element should be used as a subtree. Hence, the first child tree corresponds to the left-hand operand of the addition expression, and the second child tree corresponds to the right-hand operand.

Chapter 7

Two-dimensional parsing

There are two particular properties of RCFGs that make parsing difficult. Like other relational grammars, the languages they generate are multi-dimensional. This prevents the straightforward application of common parsing techniques like Earley's algorithm, which assume a simply ordered sequence of input tokens. Multi-dimensionality also complicates the task of deciding which subsets of the input may contain a valid parse. Furthermore, because our input is ambiguous, we require a parser to report all recognizable parse trees. Since there may be exponentially many trees, some effort is required to ensure a reasonable running time.

7.1 Shared parse forests

Each of the three recognition subsystems described in Chapters 3, 4 and 5 may introduce ambiguity into parsing, reflecting the ambiguity present in the user's handwritten input. The ambiguity takes one of three forms based on its origin:

- A particular subset of the input may or may not correspond to a distinct symbol. The output of the grouping system measures the plausibility that a given subset is in fact a symbol.
- The identity of a given symbol candidate is ambiguous. The output of the symbol recognizer assigns scores to each possible identity.
- The spatial relationship between input subsets, whether symbols or subexpressions, is ambiguous. The output of the relation classifier assigns scores to each possibility.

Because of these ambiguities, the number of potential parses of a given input is, in general, exponential in the number of input strokes. It is therefore infeasible to generate all possible parses and pick one to report to the user. This problem is similar to that of parsing ambiguous languages, in which the same input may be represented by many parse trees. (Indeed, the language of math expressions is ambiguous even in the absence

of syntactic fuzziness due to the semantic ambiguities identified in Chapter 1). Parsing ambiguous languages is a well-studied problem; we adopt the shared parse forest approach of Lang [15], in which all recognizable parses are simultaneously represented by an AND-OR tree.

For example, consider the expression shown in Figure 7.1 along with the following toy grammar:

$$\begin{aligned}
 [\text{ST}] &\Rightarrow [\text{ADD}] \mid [\text{TRM}] \\
 [\text{ADD}] &\overset{\rightarrow}{\Rightarrow} [\text{TRM}] + [\text{ST}] \\
 [\text{TRM}] &\Rightarrow [\text{MUL}] \mid [\text{SUP}] \mid [\text{CHR}] \\
 [\text{MUL}] &\overset{\rightarrow}{\Rightarrow} [\text{SUP}] [\text{TRM}] \mid [\text{CHR}] [\text{TRM}] \\
 [\text{SUP}] &\overset{\nearrow}{\Rightarrow} [\text{CHR}] [\text{ST}] \\
 [\text{CHR}] &\Rightarrow [\text{VAR}] \mid [\text{NUM}] \\
 [\text{VAR}] &\Rightarrow a \mid b \mid \dots \mid z \\
 [\text{NUM}] &\Rightarrow 0 \mid 1 \mid \dots \mid 9
 \end{aligned}$$

Ax + b

Figure 7.1: An ambiguous mathematical expression.

Figure 7.2 depicts a shared parse forest representing some possible interpretations of Figure 7.1. In the figure, the boxed arrows are AND nodes in which the arrows indicate the relation that links derived subexpressions. The ovals are OR nodes representing derivations of nonterminals on particular subsets of the input. The circles relate subsets of the input with terminal symbols from the grammar. Simple productions of the form $[\text{CHR}] \Rightarrow [\text{VAR}]$, for example, have been omitted for clarity. Any tree rooted at the $[\text{ST}]$ node that has exactly one path to each input element is a valid parse tree. This shared parse forest captures, for example, the expressions $Ax + b$, $AX + 6$, $A^x + 6$, A^Xtb , etc. If an expression is incomplete (e.g., $(x + y$ without the closing parenthesis), then no parse will exist for the correct interpretation. However, other parses using different interpretations of the input may exist (e.g., $lx + y$ or C_xty).

Parsing an RCFG may be divided into two steps: forest construction, in which a shared parse forest is created that represents all recognizable parses of the input, and tree extraction, in which individual parse trees are extracted from the forest in decreasing score order. We describe each of these steps in turn.

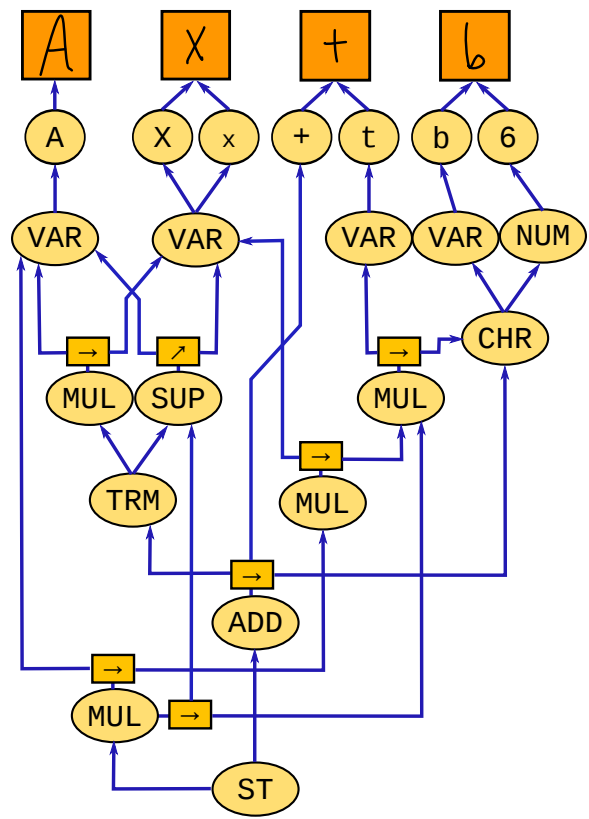


Figure 7.2: Shared parse forest for Figure 7.1.

7.2 Shared parse forest construction

Because the symbols appearing in a two-dimensional math expression cannot be simply ordered, one would naively have to parse every subset of the input in order to obtain all possible parses. To develop a practical parsing algorithm, we introduce constraints on how the input may be subdivided into individual symbols and subexpression so as to limit how many input subsets must be considered. The constraints are based on the two-dimensional structure of mathematical notation.

7.2.1 The ordering assumption and rectangular sets

Define two total orders on observables: $<_x$ orders observables by minimum x-coordinate from left to right, and $<_y$ orders observables by minimum y-coordinate from top to bottom. (We take the y axis to be oriented downward.) Associate each relation $r \in R$ with one of these orders, denoted $\text{ord } r$. $\text{ord } r$ is the dominant writing direction used for a particular relation. For math recognition, we use $\text{ord } \rightarrow = \text{ord } \nearrow = \text{ord } \searrow = \text{ord } \odot = <_x$, and $\text{ord } \downarrow = <_y$.

Informally, we assume that each relation $r \in R$ is embedded in either $<_x$ or $<_y$. Thus, we may treat any grammar production as generating either horizontal or vertical concatenations of subexpressions, making the partition-selection problem much simpler.

More formally, denote by $\min_d t$ the element $a \in t$ such that $a <_d b$ for all $b \in t$ aside from a , and define $\max_d t$ similarly.

Assumption 1 (Ordering). *Let t_1, t_2 be observables, and let e_1, e_2 be representable expressions. We assume that $((e_1, t_1), (e_2, t_2)) \notin r$ whenever $\max_{\text{ord } r} t_1 \geq_{\text{ord } r} \min_{\text{ord } r} t_2$.*

The ordering assumption says that, for a parse to exist on $t_1 \cup t_2$, the last symbol of t_1 must begin before the first symbol of t_2 along the dominant writing direction of the expression being parsed. For example, in Figure 7.1, to parse $A^x + b$ in the obvious way requires that the A begins before the x , and the $+$ begins after the x but before the b , when the symbols are considered from left to right (i.e., ordered by $<_x$).

Similarly, we could formulate a production for fractions as $[\text{FRAC}] \xrightarrow{\downarrow} [\text{EXPR}] - [\text{EXPR}]$. Then to parse a fraction would require that the bottom symbol of the numerator began before the fraction bar, and the fraction bar began before the top symbol of the denominator, when considered from top to bottom (i.e., ordered by $<_y$).

Liang et al. [18] proposed *rectangular hulls* as a subset-selection constraint for two-dimensional parsing. A very similar constraint that we call *rectangular sets* is implied by the ordering assumption.

Definition 3 (Rectangular set/partition). *Call a subset o' of an observable o rectangular in o if it satisfies*

$$o' = \left\{ a \in o : \min_x o' \leq a \leq \max_x o' \right\} \cap \left\{ a \in o : \min_y o' \leq a \leq \max_y o' \right\}.$$

Call a recursive partition t of o rectangular if every component in t is rectangular in o .

From the definition of $<_x$ and $<_y$, a set o' that is rectangular in o must include all strokes from o whose left edge lies between the left-most left edge of an element in o' and the right-most left edge *and* whose top edge lies between the top- and bottom-most top edges of elements of o' .

Proposition 1. *Let $o \in O$ be an observable, and let p be a production of the form $A_0 \xrightarrow{r} A_1 \cdots A_k$. Under the ordering assumption, if $(e_1 r \cdots r e_k, (t_1, \dots, t_k)) \in I_o^p$, then the recursive partition $t = (t_1, \dots, t_k)$ of o is rectangular.*

Proof. Let $d = \text{ord } r$, and choose any component t_i of t . We must show that

$$t_i = \left\{ a \in o : \min_x t_i \leq_x a \leq_x \max_x t_i \right\} \cap \left\{ a \in o : \min_y t_i \leq_y a \leq_y \max_y t_i \right\}.$$

It is clear that t_i is a subset of the RHS, so suppose that there is some $a' \in o$ in the RHS put into $t_j \neq t_i$ by the partition of t . If $j < i$, then

$$((t_j, e_j), (t_{j+1}, e_{j+1})), \dots, ((t_{i-1}, e_{i-1}), (t_i, e_i)) \in r.$$

By the assumption,

$$\max_d t_j <_d \min_d t_{j+1} \leq \max_d t_{j+1} <_d \cdots <_d \min_d t_i.$$

But $\min_d t_j \leq_d a' \leq_d \max_d t_j$ since $a' \in t_j$, and $\min_d t_i \leq_d a' \leq_d \max_d t_i$ since a' is in the RHS, so $\min_d t_i \leq_d a' \leq_d \max_d t_j$, a contradiction. A similar contradiction can be obtained in the case where $j > i$. \square

Rectangular sets are the natural two-dimensional generalization of contiguous substrings in one-dimensional string parsing. This definition could be generalized to arbitrary dimension, giving “hypercube sets” of input elements.

Following Liang et al., notice that any rectangular set $u \subseteq o$ can be constructed by choosing any four strokes in o and taking them to be represent the left, right, top, and bottom boundaries of the set. There are therefore $\mathcal{O}(|o|^4)$ rectangular subsets of o . If we instead naively parsed every subset of the input, there would of course be $2^{|o|}$ subsets to process. The ordering assumption thus yields a substantial reduction in the number of subsets that must be considered for parsing.

Liang et al. define a rectangular hull of a set of input elements to be their geometric bounding box, and they parse only those sets whose rectangular hulls have a null intersection. That is, no bounding box of a subset selected for parsing can intersect that of any other parsed set. This formulation causes problems for containment notations like square roots, as well as for somewhat crowded or messy handwriting styles, which our rectangular set formulation avoids. For example, consider the square root expression on the left of Figure 7.3. The rectangular hulls of the root symbol and its argument are shown as solid

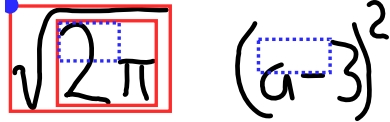


Figure 7.3: Expressions with overlapping symbol bounding boxes.

boxes and are the (union of) geometric bounding boxes of the symbols. Note that the rectangular hull of the square root symbol intersects (in fact contains) that of its contents. The argument cannot be separated from the operator into non-intersecting hulls.

When considering input subsets as rectangular sets, we do not use the natural geometric bounding box of the strokes, as Liang et al. do. Instead, we take only the minimal x and y coordinates of each stroke, and consider the bounding box (or rectangular hull) of those points. The “boundary” of the root symbol in Figure 7.3 is thus the single point at the top-left of its bounding box, and the boundary of the rectangular set representing the argument 2π is shown as a dotted box. By using minimal coordinates instead of complete bounding boxes, the rectangular set boundaries do not intersect.

Similarly, in the expression on the right of Figure 7.3, the rectangular hull of opening parenthesis intersects the hull of the a symbol, and that of the closing parenthesis intersects the hulls of both the 3 and the exponent 2. As before, the expression cannot be partitioned such that the required subsets’ hulls are non-intersecting. But the boundary of the rectangular set representing $a - 3$ (indicated by a dotted box) extends only to the left edge of the 3 symbol. The boundaries of the parentheses and the exponent are, as for the root sign, single points at the top-left corner of their bounding boxes. This small change – taking the left- and top-most coordinates of strokes as their representative points, “spaces out” overlapping writing and facilitates non-intersecting partitions.

Figure 7.4 illustrates schematically the recursive rectangular partitioning of an expression, following the expected parse of $\sum_{i=1}^{n-1} \frac{i^2}{n-i}$. The whole expression is a rectangular set. The central dotted vertical line indicates a rectangular partition into the sum symbol (with limits) and the summand. In the summand, for example, the two horizontal dashed lines indicate a rectangular partition into the numerator, fraction bar, and denominator, and the numerator and denominator are further partitioned into rectangular sets, each containing a single symbol. Note that the resulting boxes in the figure are meant to emphasize the hierarchical structure of the partition. They do not indicate the geometric bounding boxes of the rectangular sets.

7.2.2 Parsing algorithms

Using the restriction to rectangular partitions derived above, we develop bottom-up and top-down algorithms for constructing the shared parse forest of an input. The output of each of those techniques is a table B of parse links indexed by a nonterminal grammar symbol $A \in N$ and a rectangular subset o of the input observable. Table entry $B(A, o)$ represents the set I_o^A of interpretations of A of the observable o .

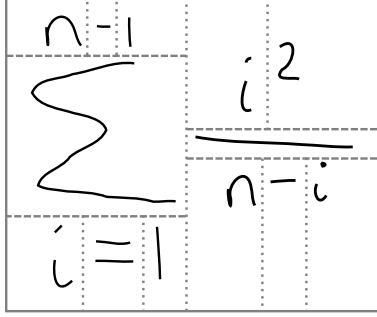


Figure 7.4: Recursive rectangular partitions in an expression.

Bottom-up parsing

Algorithm 7 generalizes to two dimensions the well-known CYK algorithm for CFG parsing. It uses dynamic programming to parse each rectangular subset of the input from smallest to largest. As in the CYK algorithm, the grammar is assumed to be in Chomsky Normal Form. That is, each production is either of the form $A_0 \Rightarrow \alpha$ for some $\alpha \in \Sigma$, or of the form $A_0 \xrightarrow{r} A_1 A_2$, where $r \in R$ and $B, C \in N$.

Algorithm 7 Bottom-up RCFG parser.

Require: An input observable o .

for every rectangular subset $o' \subset o$ of size $|o'| = 1, 2, \dots, |o|$ **do**

for each production p of the form $A \Rightarrow \alpha$ **do**

if o' is recognizable as α by the symbol recognizer **then**

$B(A, o') \leftarrow B(A, o') \cup \{(p; (o'))\}$

for each production p of the form $A_0 \xrightarrow{r} A_1 A_2$ **do**

$d \leftarrow \text{ord } r$

for $x \in (o' \setminus \{\min_d o'\})$ **do**

$o_1 \leftarrow \{a \in o' : a <_d x\}$

$o_2 \leftarrow \{a \in o' : a \geq_d x\}$

if $B(A_1, o_1) \neq \{\}$ and $B(A_2, o_2) \neq \{\}$ **then**

$B(A, o') \leftarrow B(A, o') \cup \{(p; (o_1, o_2))\}$

Any rectangular subset of an observable o may be constructed by following Algorithm 8. We can thus pre-compute all rectangular subsets of o so that they are accessible to Algorithm 7 in constant time. This precomputation step requires $\mathcal{O}(|o|^5 \log |o|)$ operations using standard sorting techniques. The algorithm proper requires $\mathcal{O}(|P||o|)$ operations per rectangular subset, for a total runtime of $\mathcal{O}(|o|^5 (|P| + \log |o|))$.

Top-down parsing

The bottom-up parsing algorithm above is straightforward but has two downsides. First, the requirement to re-write the grammar in Chomsky normal form complicates the implementation of linkage parameters in the grammar. By re-organizing the productions, the

Algorithm 8 Rectangular subset extraction.

Let d be one of x, y and let d' be the other.

List the elements of o in increasing order under $<_d$.

Extract a contiguous subsequence. (*We now have a set $o' \subseteq o$ satisfying $o' = \{a \in o : \min_d o' \leq_d a \leq_d \max_d o'\}$.*)

Re-order the remaining elements into increasing order under $<_{d'}$.

Extract a contiguous subsequence. (*The subsequence elements comprise a rectangular subset of o .*)

correspondence between non-terminals and input subsets is changed – the group of strokes referred to by a GRP linkage parameter in the CNF grammar may be different from the group referred to by the same parameter in the original grammar. Second, experiments with the CYK approach showed that, while all rectangular sets must be enumerated and parsed, relatively few actually contribute to valid parse trees. A similar observation was made by Grune and Jacobs [10] in the case of ambiguous languages. The algorithm’s runtime, while predictable, is always the worst-case time.

Instead of CYK, we therefore adapted to RCFGs a tabular variant of Unger’s method for CFG parsing [33]. In this approach, we assume that the grammar is in the normal form described in Section 6.4. At a high level, the algorithm parses a production p on an input subset o as follows:

1. If p is a terminal production, $A_0 \Rightarrow \alpha$, then check if o is recognizable as α according to the symbol recognizer. If it is, then add the parse link $(p; (o))$ to table entry $B(A_0, o)$; otherwise parsing fails.
2. Otherwise, p is of the form $A_0 \xrightarrow{r} A_1 \cdots A_k$. For every rectangular partition (t_1, \dots, t_k) of o , parse each nonterminal A_i on t_i . If any of the sub-parses fail, then fail on the current partition. Otherwise, add $(p; (t_1, \dots, t_k))$ to table entry $B(A_0, o)$. If parsing fails on every partition, then parsing fails on o .

One drawback of this algorithm is that its runtime is exponential in the size of the RHS of a grammar production, since case 2 above iterates over $\binom{|o|}{k-1}$ partitions. This bound is obtained by sorting the input by $<_{\text{ord } r}$ and choosing $k-1$ split points to induce a rectangular partition. In the worst case, then (i.e., when parses exist on every partition), our algorithm must consider every grammar production on every rectangular set, giving a complexity of $\mathcal{O}(n^{3+k}pk \log n)$, where n is the number of elements in the input observable, p is the number of grammar productions, and k is the number of RHS tokens in the largest production. The extra factor of k arises from writing up to k partition subsets into a parse table entry in case 2 above. Importantly, k can be controlled by the designer of a grammar as a tradeoff between RHS size and number of grammar productions. For example, if the grammar is written in Chomsky Normal Form (CNF), then the complexity is $\mathcal{O}(n^5p)$. Note that the general bound is asymptotically tight to the worst-case size of the parse forest, since there may be $\mathcal{O}(n^4p)$ table entries (counting each production as a distinct entry), each of which may link to $\mathcal{O}(n^{k-1}k)$ other table entries.

Instead of writing our grammar in CNF, we allow arbitrary production lengths and use the following three optimizations to reduce the number of partitions that must be considered. The first two are typical when using Unger’s method [10]; the third is specific to fuzzy r-CFG parsing.

1. *Terminal symbol milestones.* The symbol recognizer may be used to guide partitioning. Suppose p is $A_0 \xrightarrow{r} A_1 \cdots A_{i-1} \alpha A_{i+1} \cdots A_k$, where α is a terminal symbol. Then for a parse to exist on the partition (t_1, \dots, t_k) , the symbol α must have been reported as a candidate for the subset t_i . In general, given a partition, any subset corresponding to a terminal symbol in the grammar production must be recognizable as that terminal symbol. We therefore “seed” the parse table with symbol recognition results, and limit the enumeration of rectangular partitions to those for which the appropriate terminal symbols are already present in the parse table. Such seeding also facilitates recognition of typeset symbols, which are simply inserted into the parse table with their known identities prior to invocation of the parsing algorithm.
2. *Minimum nonterminal length.* If there are no empty productions in the grammar, then each nonterminal must expand to at least one terminal symbol. Moreover, given a minimum number of strokes required to recognize a particular symbol (e.g., at least two strokes may be required to form an F), one can compute the minimal number of input elements required to parse any sequence of nonterminals $A_1 \cdots A_k$. These quantities further constrain which partitions are feasible.

For example, consider the expression in Figure 7.1. Suppose we are parsing the production $[\text{ADD}] \xrightarrow{r} [\text{TRM}] + [\text{ST}]$ and we know that the $+$ symbol must be drawn with exactly two strokes. Then $[\text{ADD}]$ cannot be parsed on fewer than 4 input strokes, and the input must be partitioned into subsets t_1, t_2 , and t_3 such that $|t_1| \geq 1, |t_2| = 2, |t_3| \geq 1$. Furthermore, from the previous optimization, t_2 must be chosen such that it is recognizable as the symbol $+$. In this particular case, only one partition is feasible.

3. *Spatial relation test.* Just because an input subset can be partitioned into rectangular sets does not mean that those sets satisfy the geometric relations specified by a grammar production. Unfortunately, the grammar relations act on expressions as well as observables, so they cannot be tested during parsing because expressions are not explicitly constructed. As there may be exponentially many expressions, they *cannot* be constructed, and therefore we cannot evaluate the grammar’s spatial relations, which vary with expression identity. Still, we can speed up parsing by testing whether relations are satisfied which approximate the grammar relations.

Namely, we use the rule-based relation classifier described in Section 5.1 with semantic labels both set to the generic label GEN. If the classifier reports a score of zero when tested on adjacent partition subsets t_i and t_{i+1} , then the entire partition is discarded. The rule-based variant is not as accurate as the other relation classifiers, but it is fast to compute and accepts inputs with a wide variance of features. For this filtering stage, it is better to accept false positives than to reject false negatives.

In the algorithms below, we denote by \hat{r} the rule-based relation corresponding to grammar relation r .

Parsing a production $A_0 \xrightarrow{r} A_1 \cdots A_k$ on an observable o proceeds by two mutually recursive procedures. The first procedure, **PARSE-NT-SEQ**, parses a sequence $A_1 \cdots A_k$ of nonterminals on an observable o , as follows:

1. If $k = 1$, then parse the nonterminal A_1 on o . Fail if this sub-parse fails.
2. Otherwise, for every rectangular partition of o into two subsets, o_1 and o_2 , such that $|o_1| \geq \text{minlen}(A_1)$ and $|o_2| \geq \text{minlen}(A_2 \cdots A_k)$, parse A_1 on o_1 , and recursively parse the nonterminal sequence $A_2 \cdots A_k$ on o_2 . Fail if either parse fails.

The second procedure, **PARSE-SEQ**, parses a general sequence $A_1 \cdots A_k$ of nonterminals and terminals on o , as follows. Let $d = \text{ord } r$, and let i be minimal such that A_i is a terminal symbol. Then each rectangular $o' \subseteq o$ that is recognizable as A_i according to the symbol recognizer induces a rectangular partition of o into o_1, o', o_2 , where $\max_d o_1 <_d \min_d o'$ and $\max_d o' <_d \min_d o_2$. For each of these partitions satisfying $|o_1| \geq \text{minlen}(A_1 \cdots A_{i-1})$ and $|o_2| \geq \text{minlen}(A_{i+1} \cdots A_k)$, parse $A_1 \cdots A_{i-1}$ on o_1 , and parse $A_{i+1} \cdots A_k$ on o_2 . Fail if either sub-parse fails.

So, to parse a production, we just check whether its RHS contains terminals or not, and invoke the appropriate procedure. To clarify the outline of the algorithm, we have omitted the relational tests described in optimization 3 above. These details are included in the pseudocode below.

Algorithm 9 **PARSE-NT-SEQ**: Top-down parser for nonterminal sequences.

Require: A sequence $A_1 \cdots A_k$ of nonterminal symbols, a set o of input strokes, a grammar relation r , a leading set o_L , and a trailing set o_R .

```

 $B' \leftarrow \{\}$ 
 $d \leftarrow \text{ord } r$ 
if  $k = 1$  then
  (Verify that  $o$  fits in via  $\hat{r}$  with the leading and trailing sets)
  if  $((o_L, o) \in \hat{r}$  or  $o_L = \{\}$ ) and  $((o, o_R) \in \hat{r}$  or  $o_R = \{\}$ ) then
    return  $\text{PARSE}(A_1, o)$ 
for  $L = \text{minlen}(A_1), \dots, |o| - \text{minlen}(A_2 \cdots A_k)$  do
  Let  $o_1 \subseteq o$  be the first  $L$  elements of  $o$  ordered by  $<_d$ .
  if  $(o_L, o_1) \in \hat{r}$  or  $o_L = \{\}$  then
     $B_1 \leftarrow \text{PARSE}(A_1, o_1)$ 
    if  $B_1 \neq \{\}$  then
       $B_2 \leftarrow \text{PARSE-NT-SEQ}(A_2 \cdots A_k, o \setminus o_1, r, o_1, o_R)$ 
       $B' \leftarrow B' \cup \{b_1, b_2 : b_1 \in B_1, b_2 \in B_2\}$ 
return  $B'$ 

```

These functions comprise the two-dimensional parsing algorithm that we use in practice for MathBrush. Each production $A_0 \xrightarrow{r} A_1 \cdots A_k$ is parsed recursively by parsing $A_2 \cdots A_k$

Algorithm 10 PARSE-SEQ: Top-down parser for general production sequences.

Require: A sequence $A_1 \cdots A_k$ of grammar symbols from $N \cup \Sigma$, a set o of input strokes, a grammar relation r , and a leading set o_L .

Let i be minimal such that A_i is a terminal symbol

$B' \leftarrow \{\}$

$d \leftarrow \text{ord } r$

for every rectangular subset o' of o recognizable as the terminal symbol A_i **do**

$o_1 \leftarrow \{a \in o : a <_d \min_d o'\}$

$o_2 \leftarrow \{a \in o : \max_d o' <_d a\}$

if $\text{minlen}(A_1 \cdots A_{i-1}) \leq |o_1|$ **and** $\text{minlen}(A_{i+1} \cdots A_k) \leq |o_2|$ **then**

$B_1 \leftarrow \text{PARSE-NT-SEQ}(A_1 \cdots A_{i-1}, o_1, r, o_L, o')$

if at least one of A_{i+1}, \dots, A_k is a terminal symbol **then**

$B_2 \leftarrow \text{PARSE-SEQ}(A_{i+1} \cdots A_k, o_2, r, o')$

else

$B_2 \leftarrow \text{PARSE-NT-SEQ}(A_{k+1} \cdots A_k, o_2, r, o', \{\})$

$B' \leftarrow B' \cup \{b_1, o', b_2 : b_1 \in B_1, b_2 \in B_2\}$

return B'

Algorithm 11 PARSE: Top-down parser entry point.

Require: A nonterminal A_0 and a set o of input strokes.

if $B(A_0, o)$ is not marked “parsed” **then**

for each production p of the form $A_0 \xrightarrow{r} A_1 \cdots A_k$ **do**

if at least one of A_1, \dots, A_k is a terminal symbol **then**

$B' \leftarrow \text{PARSE-SEQ}(A_1 \cdots A_k, o, r, \{\})$

else

$B' \leftarrow \text{PARSE-NT-SEQ}(A_1 \cdots A_k, o, r, \{\}, \{\})$

Mark $B(A_0, o)$ “parsed”

$B(A_0, o) \leftarrow \{(p; x) : x \in B'\}$

return B'

for each valid subset choice for A_1 , aborting a recursive branch when no valid subset choices exist. This approach may be seen as a depth-first search in the space of valid partitions. But the optimizations described above constraint the search space and significantly speed up the parsing process. The technique is fast enough that recognition results may be updated and displayed in real-time as the user is writing.

7.3 Extracting interpretations

Recall from Chapter 6 that I_o^S is the set of interpretations of an observable o with respect to a given RCFG. Our goal is to extract elements of I_o^S and report them to the user in decreasing score order. The parse forest constructed using the algorithms in the previous section is a representation of I_o^S from which individual parse trees, each corresponding to an interpretation, may be extracted. In this section, we develop algorithms for performing

this tree extraction task efficiently.

7.3.1 The monotone assumption

Define the *rank* of an element x of a directed set (X, \leq) as the number of elements $y \in X$ such that $x \geq y$. Applying this definition to the set I_o^S of interpretations, with \leq defined in terms of the scoring function sc as described in Section 6.4, the top-scoring interpretation will have rank 1, the interpretation with second highest score will have rank 2, and so on. Denote the element of a directed set X with rank n by $[n]X$.

Consider the problem of finding $[1]I_o^p$, the highest-ranked parse of a production p on the observable o . Supposing, as usual, that p has the form $A_0 \xrightarrow{r} A_1 \cdots A_k$, every interpretation (e, t) in I_o^p includes t a recursive partition (t_1, \dots, t_k) of o and e an r -concatenation of the form $e_1 r \cdots r e_k$, where each subexpression e_i is an element $[m_i]I_{t_i}^{A_i}$ for some m_i . Since we have not constrained how $\text{sc}(e, t)$ is calculated, the only way to determine $[1]I_o^p$ is to iterate over all combinations of m_i and find the highest-scoring expression.

This exhaustive algorithm is impractical. To develop a tractable algorithm, we must constrain the behaviour of the scoring function. Its exact structure will be detailed in the next chapter, but for now we assume that the score of a terminal interpretation is a function of a symbol recognition score $S(\alpha, o)$ related to recognizing the observable o as α and some stroke grouping score $G(o)$ giving the plausibility that o constitutes a symbol:

$$\text{sc}(\alpha, t) = g(S(\alpha, o), G(o)),$$

for t the trivial partition $\{o\}$ of o .

Similarly, the score of interpreting the recursive partition $t = (t_1, \dots, t_k)$ as the r -concatenation $e = e_1 r \cdots r e_k$ is assumed to be a function of symbol recognition, stroke grouping, and relation classification scores, decomposable as

$$\text{sc}(e, t) = f_k(\text{sc}(e_1, t_1), \dots, \text{sc}(e_k, t_k), R_r((e_1, t_1), (e_2, t_2)), \dots, R_r((e_{k-1}, t_{k-1}), (e_k, t_k))),$$

where $R_r(a, b)$ is a scoring function for the relation r applied to the interpretations a and b .

Given this assumption on the general structure of sc , we further assume that the combination functions g and f_i increase monotonically with their arguments, subject to some constraints. Let $\text{cl}(e)$ denote the semantic label associated with an expression e . (Recall from Chapter 5 that the semantic labels group together symbols with similar bounding-box profiles.)

Assumption 2 (Monotone). *Assume*

1. If $S(t, \beta) > S(t, \alpha)$, then $g(S(t, \beta), G(t)) > g(S(t, \alpha), G(t))$.
2. If $\text{cl}(e_1) = \text{cl}(e'_1)$ and $\text{cl}(e_2) = \text{cl}(e'_2)$, then $R_r((t_1, e_1), (t_2, e_2)) = R_r((t_1, e'_1), (t_2, e'_2))$.

3. If $\text{sc}(t_i, e_i) \geq \text{sc}(t_i, e'_i)$ and $\text{cl}(e_i) = \text{cl}(e'_i)$ for $i = 1, \dots, k$, then

$$f_k(\text{sc}(t_1, e_1), \dots, \text{sc}(t_k, e_k), R_r((t_1, e_1), (t_2, e_2)), \dots, R_r((t_{k-1}, e_{k-1}), (t_k, e_k))) \geq f_k(\text{sc}(t_1, e'_1), \dots, \text{sc}(t_k, e'_k), R_r((t_1, e'_1), (t_2, e'_2)), \dots, R_r((t_{k-1}, e'_{k-1}), (t_k, e'_k))).$$

7.3.2 Extracting elements of I_o^S

Using the monotone assumption, we develop an algorithm for extracting elements of I_o^S in rank order. The basic idea of our algorithm is to first extract the most highly-ranked expression, $[1]I_o^S$, and then to recursively maintain an expanding “frontier” of potential next-best expressions, from which subsequent expressions are obtained.

In the following, we use the following refinements of the sets of interpretations in Definition 2. For each nonterminal A and semantic label c , let $I_o^A(c) = \{(e, t) \in I_o^A : \text{cl}(e) = c\}$ restrict I_o^A to expressions with label c . For a recursive partition $t \in T(o)$, let $I_t^p = \{(e, t) : (e, t) \in I_o^p\}$ restrict I_o^p to interpretations using the partition t . Finally, for semantic labels c_1, \dots, c_k , let $I_t^p(c_1, \dots, c_k) = \{(e_1 r \dots r e_k, t) \in I_t^p : \text{cl}(e_i) = c_i, i = 1, \dots, k\}$ restrict I_t^p to interpretations in which each subexpression has a particular semantic label.

Number the semantic labels derivable from A_0 as c_1, \dots, c_N . It is clear that $[1]I_o^{A_0} = \text{argmax} \{\text{sc}(e) : e = [1]I_o^{A_0}(c_i), i = 1, \dots, N\}$. Suppose we have determined $[j]I_o^{A_0}$ up to $j = n$, yielding a partial set of expressions $I = \{[j]I_o^{A_0} : j = 1, \dots, n\}$. Then $[n+1]I_o^{A_0}$ is given by the following result.

Proposition 2. *Let $I = \bigcup_i \{[j]I_o^{A_0}(c_i) : 1 \leq j \leq n_i\}$. That is, the n_i most highly-ranked elements of each $I_o^{A_0}(c_i)$ have already been extracted from $I_o^{A_0}$. Then*

$$[n+1]I_o^{A_0} = \text{argmax} \{\text{sc}(e) : e = [n_i+1]I_o^{A_0}(c_i), i = 1, \dots, N\}.$$

In this proposition, the indices n_i represent the expanding frontier of potential next-best expressions. The actual next-best expression is just the best expression from the frontier. After extraction, the frontier must be updated by incrementing the appropriate n_i .

Expressions may be extracted in ranked order from the set I_o^p by similarly maximizing over a frontier of potential “next-best” expressions, based on the initial maximizations

$$[1]I_o^p = \text{argmax} \{\text{sc}(e) : e = [1]I_t^p, t \in T_k(o)\}$$

and

$$[1]I_t^p = \text{argmax} \{\text{sc}(e) : e = [1]I_t^p(c_1, \dots, c_k); c_1, \dots, c_k \text{ are semantic labels}\}.$$

However, this strategy must be generalized somewhat to extract expressions from the sets $I_t^p(c_1, \dots, c_k)$. For brevity, we denote an r -concatenation $[n_1]I_{t_1}^{A_1}(c_1)r \dots r [n_k]I_{t_k}^{A_k}(c_k)$ in a given set $I_t^p(c_1, \dots, c_k)$ by its indices as (n_1, \dots, n_k) .

Part 3 of the monotone assumption implies that the score of an interpretation using the expression $n = (n_1, \dots, n_k)$ is at least that of an interpretation of the same input using expression $m = (m_1, \dots, m_k)$ whenever $n_i \leq m_i$ for all i (written $n \leq m$; we similarly define other comparison relations component-wise). Proposition 3 uses this observation to show how to extract expressions from $I_t^p(c_1, \dots, c_k)$ in ranked order. But first, we introduce the idea of successor sets, which describe how the frontier of expressions expands.

Definition 4. Let $e \in I_t^p(c_1, \dots, c_k)$ be (m_1, \dots, m_k) for some indices m_i . We define the successor set of e to be

$$\text{succ}(e) = \{ (m_1 + 1, m_2, \dots, m_k), \\ (m_1, m_2 + 1, \dots, m_k), \dots, \\ (m_1, m_2, \dots, m_k + 1) \}.$$

Thus, $\text{succ}(e)$ contains all of the expressions obtainable from $I_t^p(c_1, \dots, c_k)$ by incrementing exactly one of the extraction indices m_i of e . Given a subset I of $I_t^p(c_1, \dots, c_k)$ already extracted in ranked order, the next-best expression is found in the successor set of an expression on the frontier of I . The following result makes this idea precise.

Proposition 3. Suppose we have determined $[j] I_t^p(c_1, \dots, c_k)$ up to $j = n$, yielding a partial set of expressions I . Define the boundary of I to be $\text{bd } I = \{e \in I : \text{succ}(e) \not\subseteq I\}$. Let $X = I_t^p(c_1, \dots, c_k)$. Then,

1. If I is empty, then $[1] X = (1, 1, \dots, 1)$.
2. Otherwise, let $S = (\bigcup_{e \in \text{bd } I} \text{succ}(e)) \setminus I$. Then

$$[n + 1] X = \underset{e \in S}{\text{argmax}} \text{sc}(e)$$

Proof. 1. Suppose to the contrary that $[1] X = (m_1, \dots, m_k)$. This immediately contradicts the monotone assumption since $(m_1, \dots, m_k) \geq (1, \dots, 1)$.

2. Suppose to the contrary that $[n + 1] X = e' = (m_1, \dots, m_k)$ with $e' \notin S$. Wlog, suppose that m_1 is the largest entry in the tuple, and consider the expression represented by $e^* = (m_1 - 1, m_2, \dots, m_k)$. By the monotone assumption, this expression is scored higher than e' . Inductively, since $[n + 1] X = e'$, e^* must already be in I . But e' is in the successor set of e^* , a contradiction.

□

Figure 7.5 illustrates graphically an example of applying Proposition 3 for $k = 2$. The extraction process can be thought of as growing a staircase-type shape out from the origin in a k -dimensional lattice.

The following algorithms make these results concrete with respect to the parse graph B produced by the algorithms in the previous section. The details of iterating over semantic labels are omitted for clarity, but are discussed following the pseudocode listings.

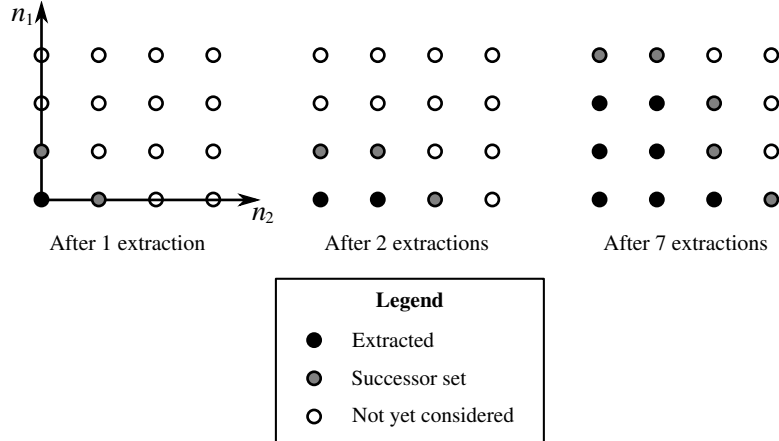


Figure 7.5: Illustration of expression extraction for a two-token production.

We divide the problem into two parts: extracting interpretations from a particular branch $(p; (t_1, \dots, t_k))$, and extracting interpretations from a nonterminal node $B(A, o)$. These two parts are implemented as mutually-recursive procedures invoked according to the structure of the grammar productions.

Algorithms 12 and 14 implement the first part, while Algorithms 13 and 15 implement the second part. They essentially translate the consequences of the monotone assumption into parse graph operations. Note that each algorithm uses data structures local to the point in B from which it is extracting expressions. One can think of these algorithms as being associated directly with each node and branch in the parse graph. The process is initialized by calling `BEST-NONTERMINAL-EXPRESSION(S, o)` with S the start symbol and o the entire input. `NEXT-NONTERMINAL-EXPRESSION(S, o)` may then be called repeatedly to iterate over all parses of o .

Furthermore, two distinct modes of extraction are supported. In the `EXHAUSTIVE` mode, expressions are extracted exactly as suggested by the monotone assumption. However, this leads to an overwhelming number of expressions being available to the user. A second mode, called `SEMANTICS`, is more restrictive, and is the default mode for expressions larger than a single terminal symbol.

In `SEMANTICS` mode, all reported expressions must either be derived from different productions (and thus represent different parse structures), or, if derived from the same production, must partition the input differently into subexpressions. This restriction effectively constrains the number of alternatives available, while still allowing all possible parse results to be obtained by examining the alternatives for different subexpressions. It is easily implemented by extracting only one expression per branch.

It is difficult to precisely quantify the complexity of these algorithms. However, we can characterize them in terms of the size of B . The initialization step, yielding $[1]I_t$, follows every possible branch in $B(S, t)$ for S the start symbol and t the entire input, visiting each node once. Extracting $[n + 1]I_t$ entails a visit to only one branch per nonterminal visited, but to k nonterminals per branch $(p; (t_1, \dots, t_k))$ visited, where k is the number of RHS tokens in the production p . Note, though, that each such token corresponds to a

Algorithm 12 BEST-LINK-EXPRESSION: Extract the most highly-ranked expression from a branch.

Require: A branch $(p; (t_1, \dots, t_k))$
 $cache \leftarrow \{\}$ (*Initialize priority queue local to the branch*)
if p is a terminal production $A_0 \xrightarrow{r} \alpha$ **then**
 $e^* \leftarrow \alpha$
else
(p is of the form $A_0 \xrightarrow{r} A_1 \cdots A_k$)
for $i = 1, \dots, k$ **do**
 $e_i \leftarrow$ BEST-NONTERMINAL-EXPRESSION(A_i, t_i)
 $e^* \leftarrow (e_1 r \cdots r e_k)$
 $[0] I_{t_1 \cup \dots \cup t_k}^p \leftarrow e^*$
return e^*

Algorithm 13 NEXT-LINK-EXPRESSION: Extract the next most highly-ranked expression from a branch.

Require: A branch $(p; (t_1, \dots, t_k))$
if extraction mode is SEMANTICS **then**
return NONE
if p is a terminal production $A_0 \xrightarrow{r} \alpha$ **then**
return NONE
(p is of the form $A_0 \xrightarrow{r} A_1 \cdots A_k$)
Suppose n expressions have already been extracted from this branch.
Let $(e_1 r \cdots r e_k) = [n] I_{t_1 \cup \dots \cup t_k}^p$ be the last expression extracted here
Let m_i be such that $e_i = [m_i] I_{t_i}^{A_i}$ for $i = 1, \dots, k$
for $i = 1, \dots, k$ **do**
if $m_i = |I_{t_i}|$ **then**
 $\hat{e}_i \leftarrow$ NEXT-NONTERMINAL-EXPRESSION(A_i, t_i)
else
 $\hat{e}_i \leftarrow [m_i + 1] I_{t_i}^{A_i}$
Add $e = (e_1 r \cdots r e_{i-1} r \hat{e}_i r e_{i+1} r \cdots r e_k)$ to $cache$ with priority $I_i^p(e)$
if $cache$ is empty **then**
return NONE
Pop e^* from $cache$
 $[n + 1] I_{t_1 \cup \dots \cup t_k}^p \leftarrow e^*$
return e^*

Algorithm 14 BEST-NONTERMINAL-EXPRESSION: Extract the most highly-ranked expression derivable from a nonterminal.

Require: A nonterminal A and an observable t .

$cache \leftarrow \{\}$ Initialize priority queue local to (A, t)

for every branch $(p; x) \in B(A, t)$ **do**

 Add $e = \text{BEST-LINK-EXPRESSION}(p; x)$ to $cache$ with priority $I_t^p(e)$

Pop e^* from $cache$

$[0] I_t^A \leftarrow e^*$

return e^*

Algorithm 15 NEXT-NONTERMINAL-EXPRESSION: Extract the next most highly-ranked expression derivable from a nonterminal.

Require: A nonterminal A and an observable t .

Suppose n expressions have already been extracted from $B(A, t)$.

Let $e = [n] I_t^A$ be the last expression extracted here

Let $(p; (t_1, \dots, t_k))$ be the branch from which e was extracted

Add $\hat{e} = \text{NEXT-LINK-EXPRESSION}(p; (t_1, \dots, t_k))$ to $cache$ with priority $I_t^p(\hat{e})$

if $cache$ is empty **then**

return NONE

Pop e^* from $cache$

$[n + 1] I_t^A \leftarrow e^*$

return e^*

subexpression in the parse. The amount of work performed in this case is therefore directly proportional to the number of nodes in a parse tree representing $[n]I_t$.

To include semantic labels in these algorithms requires some modifications of Algorithms 12 and 13. Whenever a terminal expression is extracted by a recursive call in one of the algorithms (say from the nonterminal A on the trivial recursive partition t), that recursive call is repeated to extract *all* of the terminal interpretations of A on t . All the interpretations arising from those extractions are added to the priority queue. In this way, all combinations of semantic labels are accounted for. Because of this exhaustive extraction of terminal expressions, the first part of the monotone assumption is unnecessary in practice.

In the worst case, if all possible combinations of labels are populated, this adds an additional factor of C^k to the complexity of the tree extraction algorithm, where C is the number of semantic labels and k is the number of RHS elements of p . Importantly, both C and k are controlled by the grammar designer. Furthermore, recall that only different terminal symbols are given meaningfully different semantic labels; all larger expressions share the generic labels `EXPR` and `GEN`. This restriction is important because it limits this extra iteration to only those expressions directly involving terminal symbols (i.e., near the leaves of the parse forest). No extra iteration is invoked for larger expressions because they do not possess any non-trivial semantic labels.

The restriction of semantic labels to terminal expressions also simplifies data organi-

zation. In general, to iterate over interpretations of a production $A_0 \xrightarrow{\tau} A_1 \cdots A_k$ on a recursive partition t_1, \dots, t_k whose subexpressions have semantic labels c_1, \dots, c_k requires iteration over interpretations of each A_i on t_i to be restricted to semantic label c_i . If different nonterminal expressions possessed a variety of semantic labels, this semantic restriction would add significant complexity to the structure of the parse forest as well as Algorithms 14 and 15.

7.3.3 Handling user corrections

As mentioned in the introduction, we wish to provide to users a simple correction mechanism so that they may select their desired interpretation in case of recognition errors or multiple ambiguous interpretations. Our recognizer facilitates such corrections by allowing *locks* to be set on any subset of the input.

Two types of locks are supported:

1. *Expression locks*, which fix the interpretation of a particular subset of the input to be a specified expression, and
2. *Semantic locks*, which force interpretations of a particular subset of the input to be derived from a specified nonterminal.

Each of these lock types is useful in different circumstances. For example, if $x + a$ was recognized as $X + a$, then an expression lock may be applied to the portion of the input recognized as the X symbol, forcing it to be interpreted as a lower-case x instead. If $x + a$ was recognized as $x + a$, then a semantic lock may be applied to the entire input, forcing an addition expression to be derived. If recognition gave $X + a$, then the lock types may be combined.

Consider extracting an expression from input o . If o is locked to an expression e by an expression lock, then we consider I_o to contain only one element, e . If o is locked to a nonterminal A_L by a semantic lock, then we take $I_o^{A'}$ to be empty for all nonterminals $A' \neq A_L$, except those for which a derivation sequence $A' \Rightarrow^* A_L$ exists, in which case we take $I_o^{A'} = I_o^{A_L}$.

MathBrush allows users to navigate ranked interpretations on any subset of their input, as shown in Figure 7.6. In case recognition is not accurate, the user may select the correct interpretation and an appropriate lock is applied to the parse graph. The corrections are maintained in subsequent parse results as the user continues to write. In this way, correction is simple and relatively painless – certainly easier than erasing the expression and rewriting it from scratch.

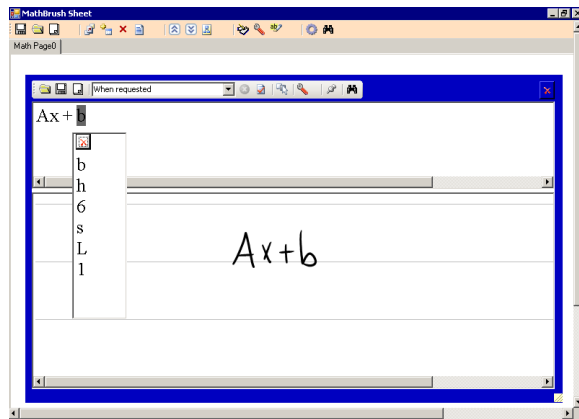
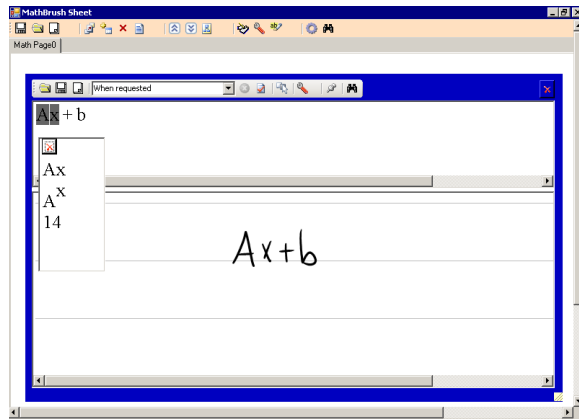
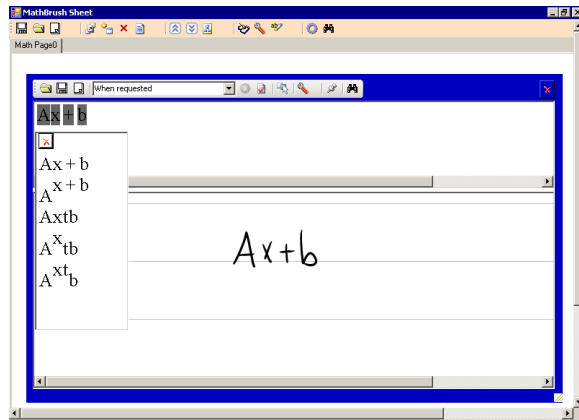


Figure 7.6: Interface for displaying and selecting alternative interpretations.

Chapter 8

Scoring interpretations

The previous chapter described how a scoring function on interpretations, $sc(e, t)$, may be used to efficiently organize and report all parses of an input observable in order of decreasing score. The algorithms we developed assumed that sc was either:

1. A function $sc(\alpha, t) = g(S(\alpha, t), G(t))$ of underlying symbol and stroke grouping score functions for a terminal expression α , or
2. A function

$$sc(e_1 r \cdots r e_k, (t_1, \dots, t_k)) = f_k(sc(e_1, t_1), \dots, sc(e_k, t_k), R_r((e_1, t_1), (e_2, t_2)), \dots, R_r((e_{k-1}, t_{k-1}), (e_k, t_k)))$$

of sub-interpretation scores and a relation scoring function, in the general case.

In either case, we require the combination functions g and f_k to be monotonic increasing in all of their arguments, as described in the previous chapter.

In this chapter, we develop two concrete scoring functions satisfying these assumptions, respectively based on fuzzy logic and probability theory. In the next chapter, the systems resulting from combining these scoring functions with the parsing techniques described previously will be evaluated and compared with an earlier version of the MathBrush recognizer as well as other recognition systems.

8.1 Fuzzy logic-based scoring

In the fuzzy case, we treat the set I_o of interpretations of o as a fuzzy set, and treat sc as its membership function. Although the classical membership function for conjunctive variable combinations (e.g., “the strokes of o correspond to a symbol *and* represent the symbol α ”) is the min function, Zhang et al [41] found that using multiplication when computing

membership grades helped to prevent ties. We therefore compute the membership grade in the terminal case as

$$\text{sc}(\alpha, t) = \sqrt{S(\alpha, o)G(t)},$$

where $S(\alpha, o) = D_\alpha(o)^{-2}$ for D_α the hybrid symbol distance function from Equation 4.7, and $G(t)$ is the stroke grouping score from Chapter 3.

In the general case of an r -concatenation, we compute the membership grade as

$$\text{sc}(e_1 r \cdots r e_k, (t_1, \dots, t_k)) = \left(\left(\prod_{i=1}^k \text{sc}(e_i, t_i) \right) \left(\prod_{i=1}^{k-1} \mu_r((e_i, t_i), (t_{i+1}, e_{i+1})) \right) \right)^{\frac{1}{2k-1}},$$

where μ_r is the membership function from Section 5.1 treating each relation r as fuzzy.

The geometric averaging used in these scoring functions preserves the tie-breaking properties of multiplication while normalizing for expression size.

8.2 Probabilistic scoring

The probabilistic case requires significantly more care to develop than the fuzzy case. None of the underlying scoring functions described above naturally produce probability distributions, so some transformation of those functions is required. However, probabilistic formalisms do offer straightforward and systematic ways in which to represent “common-sense” information such as symbol and subexpression co-occurrence frequencies (e.g., preferring *sin* over *sjn*), and we wish to include such details in our model.

8.2.1 Model structure

Given an input observable $\hat{o} \in O$, we define the following families of variables:

- An expression variable $E_o \in \{e : (e, t) \in I_o \text{ for some } t\} \cup \{\text{NIL}\}$ from the set of interpretations of each $o \subseteq \hat{o}$. $E_o = \text{NIL}$ indicates that the subset o has no meaningful interpretation.
- A symbol variable $S_o \in \Sigma \cup \{\text{NIL}\}$ for each $o \subseteq \hat{o}$ indicating what terminal symbol o represents. $S_o = \text{NIL}$ indicates that o is not a symbol. (o could be a larger subexpression, or could have $E_o = \text{NIL}$ as well.)
- A relation variable $R_{o_1, o_2} \in R \cup \{\text{NIL}\}$ indicating which grammar relation joins the subsets $o_1, o_2 \subset \hat{o}$. $R_{o_1, o_2} = \text{NIL}$ indicates that the subsets are not directly connected by a relation.
- A vector of grouping-oriented features g_o for each $o \subseteq \hat{o}$.
- A vector of symbol-oriented features s_o for each $o \subseteq \hat{o}$.

- A vector of relation-oriented features f_o for each $o \subset \hat{o}$.
- A “symbol-bag” variable B_o distributed over $\Sigma \cup \{\text{NIL}\}$ for each $o \subseteq \hat{o}$.

This is a large group of variables: there are $2^{2^{|\hat{o}|}}$ relation variables alone! Fortunately, we may restrict ourselves to variable assignments which correspond to valid parse trees, which significantly reduces the model’s complexity through the following deterministic constraints.

- The set of subsets $\{o : E_o \neq \text{NIL}\}$ with non-NIL expressions must correspond exactly to a recursive partition of \hat{o} , otherwise the expressions do not combine to form a parse tree.
- If $E_o = \text{NIL}$, then $S_o = \text{NIL}$, and all $R_{o,*}$ and $R_{*,o}$ are also NIL.
- E_o is a terminal expression iff $S_o \neq \text{NIL}$.

Additionally, the ordering assumption of Section 7.2.1 allows us to also specify $E_o = \text{NIL}$ if o is not a rectangular set. This immediately reduces the number of potential non-NIL expression variables to $\mathcal{O}(|\hat{o}|^4)$ and the number of relation variables to $\mathcal{O}(|\hat{o}|^8)$.

That is still a considerable number of variables, though, so direct assessment of the joint distribution is infeasible. Instead, we rely on the algorithms from the previous chapter to propose parse trees. Each parse tree corresponds to an assignment to the capitalized variables defined above in which the constraints just given are known to be satisfied. We compute a score for each such assignment that is proportional to the joint probability

$$P \left(\bigwedge_{o \subseteq \hat{o}} E_o, \bigwedge_{o \subseteq \hat{o}} S_o, \bigwedge_{o_1, o_2 \subset \hat{o}} R_{o_1, o_2}, \bigwedge_{o \subseteq \hat{o}} g_o, \bigwedge_{o \subseteq \hat{o}} s_o, \bigwedge_{o \subset \hat{o}} f_o, \bigwedge_{o \subseteq \hat{o}} B_o \right).$$

To do so, first factor the joint distribution as the following Bayesian network:

$$\begin{aligned}
& P \left(\bigwedge_{o \subseteq \hat{o}} E_o, \bigwedge_{o \subseteq \hat{o}} S_o, \bigwedge_{o_1, o_2 \subset \hat{o}} R_{o_1, o_2}, \bigwedge_{o \subseteq \hat{o}} g_o, \bigwedge_{o \subseteq \hat{o}} s_o, \bigwedge_{o \subset \hat{o}} f_o, \bigwedge_{o \in \hat{O}} B_o \right) \\
&= \prod_{o' \subset \hat{o}} P \left(E_{o'} \mid \bigwedge_{o \subset o'} E_o, \bigwedge_{o \subseteq o'} S_o, \bigwedge_{o_1, o_2 \subset o'} R_{o_1, o_2}, \bigwedge_{o \subseteq o'} g_o, \bigwedge_{o \subseteq o'} s_o, \bigwedge_{o \subset o'} f_o \right) \\
&\times \prod_{o \subseteq \hat{o}} P(S_o \mid g_o, s_o) \\
&\times \prod_{o_1, o_2 \subset \hat{o}} P(R_{o_1, o_2} \mid E_{o_1}, E_{o_2}, f_{o_1}, f_{o_2}) \\
&\times \prod_{o' \subseteq \hat{o}} P \left(B_{o'} \mid \bigwedge_{o \subseteq \hat{o}} S_o \right) \\
&\times \prod_{o \subseteq \hat{o}} g_o \prod_{o \subseteq \hat{o}} s_o \prod_{o \subset \hat{o}} f_o.
\end{aligned} \tag{8.1}$$

Note that the probabilities of the feature variables g_o, s_o , and f_o are functions only of the input and do not change with the other variables. They may thus be omitted from calculation as we wish only to compare the relative scores of different parse trees.

Next, we will describe each of the conditional distributions before showing how to calculate their product.

8.2.2 Expression variables

The distribution of expression variables,

$$E_{o'} \mid \bigwedge_{o \subset o'} E_o, \bigwedge_{o \subseteq o'} S_o, \bigwedge_{o_1, o_2 \subset o'} R_{o_1, o_2}, \bigwedge_{o \subseteq o'} g_o, \bigwedge_{o \subseteq o'} s_o, \bigwedge_{o \subset o'} f_o,$$

is deterministic and based on the grammar productions. In a valid joint assignment (i.e., one derived from a parse tree), each non-NIL conditional expression variable E_o is a subexpression of $E_{o'}$, and the non-NIL relation variables indicate how these subexpressions are joined together. This information completely determines what expression $E_{o'}$ must be, so that expression is assigned probability 1.

In the case where $E_{o'}$ is a terminal expression α , it must be the case that $E_o = \text{NIL}$ in the joint assignment for all $o \subset o'$, and similarly all the relation variables dealing with subsets of o' must be NIL. Only the symbol variable S_o takes a non-nil value, and it must be α . So again we have the deterministic distribution

$$P \left(E_{o'} = \alpha \mid S_o = \alpha, \bigwedge_{o \subset o'} (E_o = \text{NIL}), \bigwedge_{o_1, o_2 \subset o'} (R_{o_1, o_2} = \text{NIL}), \bigwedge_{o \subseteq o'} g_o, \bigwedge_{o \subseteq o'} s_o, \bigwedge_{o \subset o'} f_o \right) = 1.$$

8.2.3 Symbol variables

Each grouping-oriented feature vector g_o is a 5-tuple with elements given by the measurements $d, \ell_{in}, c_{in}, \ell_{out}$, and c_{out} (respectively “distance”, “in-group overlap”, “in-group containment notation score”, “out-of-group overlap”, and “out-of-group containment notation score”) described in Section 3.2 on the grouping algorithm. Let N_G be the grouping score given by Equation 3.2.

Each symbol-oriented feature vector s_o contains an element $s_{o,\alpha}$ for each $\alpha \in \Sigma$ with $s_{o,\alpha} = D_\alpha(o)$, the hybrid symbol recognition distance from Equation 4.7 in Section 4.2. For a given observable o , let $p_\alpha = s_{o,\alpha}^{-2}$ for each α , and let $N_S = \max_{\alpha \in \Sigma} 1/s_{o,\alpha}$. Then the probability that the strokes in o do not correspond to a symbol is defined to be

$$P(S_o = \text{NIL} \mid g_o, s_o) = 1 - \frac{N}{N+1},$$

where $N = \log(1 + N_S N_G)$.

The remaining probability mass, corresponding to the case where o is indeed a symbol, is distributed amongst the terminal symbols such that

$$P(S_o = \alpha \mid g_o, s_o) \propto \frac{p_\alpha}{\sum_{\alpha \in \Sigma} p_\alpha}.$$

8.2.4 Relation variables

Each relation-oriented feature vector f_o encodes the bounding box of o . From a pair f_{o_1}, f_{o_2} of such vectors, the bounding-box features f_1 through f_7 described in Section 5.2 are easily computed.

For each relation $r \in R$, let $p_r = P(X = r \mid e_1, e_2, f_1, \dots, f_7)$ from Equation 5.1 (the naive Bayesian probability of relation r holding between expressions e_1, e_2 with bounding box features f_1, \dots, f_7), and let $N = \log(1 + \max_{r \in R} p_r)$. Then, similarly to the symbol case, we set

$$P(R_{o_1, o_2} = \text{NIL} \mid E_{o_1}, E_{o_2}, f_{o_1}, f_{o_2}) = 1 - \frac{N}{N + 1},$$

and distribute the remaining probability mass amongst the relations so that

$$P(R_{o_1, o_2} = r \mid E_{o_1}, E_{o_2}, f_{o_1}, f_{o_2}) \propto \frac{p_r}{\sum_{r \in R} p_r}.$$

This distribution only applies when $E_{o_1}, E_{o_2} \neq \text{NIL}$. If either expression is NIL, then the relation must be NIL, so

$$P(R_{o_1, o_2} = \text{NIL} \mid E_{o_1} = \text{NIL}, E_{o_2}, f_{o_1}, f_{o_2}) = P(R_{o_1, o_2} = \text{NIL} \mid E_{o_1}, E_{o_2} = \text{NIL}, f_{o_1}, f_{o_2}) = 1.$$

Although the naive Bayesian classifier did not perform as well as the grid- and margin-tree-based relation classifiers in the isolated relation classification task from Chapter 5, we found that the performance of those classifiers dropped significantly in the more complex environment of math recognition. In the evaluation, every test example consisted of two known subexpressions with an unknown but non-NIL relation joining them. But in the full context of recognition, there are a large number of potential subexpressions which may or may not be joined by relations. The discriminative classifiers, because they rely exclusively on frequency counts of positive examples, are not well-suited for distinguishing between the NIL and non-NIL cases. So, although they performed well in the isolated classification task, we use the naive classifier in the context of full expression recognition.

8.2.5 Symbol-bag variables

The role of the symbol-bag variables B_o is to apply a prior distribution to the terminal symbols appearing in the input, and to adapt that distribution based on which symbols are currently present. Because those symbols are not known with certainty, such adaptation may only be done imperfectly.

We collected symbol occurrence and co-occurrence rates from the Infty project corpus [30] as well as from the L^AT_EX sources of University of Waterloo course notes for introductory algebra and calculus courses. The symbol occurrence rate $r(\alpha)$ is the number of expressions in which the symbol α appeared and the co-occurrence rate $c(\alpha, \beta)$ is the number of expressions containing both α and β .

In theory, each B_o variable is identically distributed and satisfies

$$P\left(B_o = \text{NIL} \mid S_o = \text{NIL}, \bigwedge_{o' \subseteq \hat{o}} S_{o'}\right) = 1$$

and

$$P\left(B_o = \alpha \mid S_o = \delta, \bigwedge_{o' \subseteq \hat{o}} S_{o'}\right) \propto r(\alpha) + \sum_{\beta \in \Sigma} P(S_{o'} = \beta \text{ for some } o' \subseteq \hat{o}) \frac{c(\alpha, \beta)}{r(\beta)},$$

for any non-NIL $\delta \in \Sigma$ (recall that \hat{o} is the entire input observable).

The trivial NIL case allows us to remove all terms with $B_{o'} = \text{NIL}$ from the product $\prod_{o' \subseteq \hat{o}} P(B_{o'} \mid \bigwedge_{o'' \subseteq \hat{o}} S_{o''})$ in our scoring function.

In the non-NIL case, note that for any subset o' of the input for which the grouping score $G(o')$ is 0, we have $S_{o'} = \text{NIL}$ with probability one. Thus in the $P(S_{o'} = \beta \text{ for some } o' \subseteq \hat{o})$ term above, we need only consider those subsets of o such that $G(o') > 0$. Rather than assuming independence and iterating over all subsets with non-zero grouping score, or developing a more complex model, we simply approximate this probability by

$$\max_{o' \subseteq \hat{o}, G(o') > 0} P(S_{o'} = \beta \mid g_{o'}, s_{o'}).$$

Evaluation of these symbol-bag probabilities proceeds as follows. Starting with an empty input, only the $r(\alpha)$ term is used. The distribution is updated incrementally each time symbol recognition is performed on a candidate stroke group. The set of available candidate groups is updated each time a new stroke is drawn by the user. When a group corresponding to an observable o is identified, the symbol recognition and stroke grouping processes induce the distribution of S_o , which we use to update the distribution of the B_* variables by checking whether any of the max expressions need to be revised. Intuitively, B_* treats the “next” symbol to be recognized as being drawn from a bag full of symbols. If a previously-unseen symbol β is drawn, then more symbols are added to the bag based on the co-occurrence counts of β .

This process is similar to updating the parameter vector of a Dirichlet distribution, except that the number of times a given symbol appears within an expression is irrelevant. We are concerned only with how likely it is that the symbol appears at all.

8.2.6 Calculating the joint probability

Based on the discussion above, it is clear that many of the variables in our Bayesian model must be set to NIL in order for the joint assignment to correspond to a valid parse tree.

In particular, a variable is set to NIL unless it corresponds exactly to some meaningful part of the parse tree, whether that is a symbol name, a subexpression, or a relation between subsets. We will use this fact to dramatically simplify the calculation of the joint probability given in Equation 8.1.

Let

$$Z = \prod_{o \subseteq \hat{o}} P(S_o = \text{NIL} \mid g_o, s_o) \times \prod_{o_1, o_2 \subset \hat{o}} P(R_{o_1, o_2} = \text{NIL} \mid E_{o_1}, E_{o_2}, f_{o_1}, f_{o_2}).$$

Then, under the assumption that Z is constant with respect to the E_* variables, we divide Equation 8.1 by Z to remove all of the NIL variables, and divide by g_o, s_o, f_o as described earlier, since they are functions only of the input:

$$\begin{aligned} & P \left(\bigwedge_{o \subseteq \hat{o}} E_o, \bigwedge_{o \subseteq \hat{o}} S_o, \bigwedge_{o_1, o_2 \subset \hat{o}} R_{o_1, o_2}, \bigwedge_{o \subseteq \hat{o}} g_o, \bigwedge_{o \subseteq \hat{o}} s_o, \bigwedge_{o \subset \hat{o}} f_o, \bigwedge_{o \subseteq \hat{o}} B_o \right) \\ & \propto \prod_{o' \subset \hat{o}, E_{o'} \neq \text{NIL}} P \left(E_{o'} \mid \bigwedge_{o \subseteq o'} E_o, \bigwedge_{o \subseteq o'} S_o, \bigwedge_{o_1, o_2 \subset o'} R_{o_1, o_2}, \bigwedge_{o \subseteq o'} g_o, \bigwedge_{o \subseteq o'} s_o, \bigwedge_{o \subset o'} f_o \right) \\ & \times \prod_{o \subseteq \hat{o}, S_o \neq \text{NIL}} \frac{P(S_o \mid g_o, s_o)}{P(S_o = \text{NIL} \mid g_o, s_o)} \\ & \times \prod_{o_1, o_2 \subset \hat{o}, R_{o_1, o_2} \neq \text{NIL}} \frac{P(R_{o_1, o_2} \mid E_{o_1}, E_{o_2}, f_{o_1}, f_{o_2})}{P(R_{o_1, o_2} = \text{NIL} \mid E_{o_1}, E_{o_2}, f_{o_1}, f_{o_2})} \\ & \times \prod_{o' \subseteq \hat{o}, S_{o'} \neq \text{NIL}} P \left(B_{o'} \mid \bigwedge_{o \subseteq o'} S_o \right) \end{aligned} \tag{8.2}$$

The RHS of Equation 8.2 as the scoring function for the parse tree extraction algorithms from the previous chapter. The division removes from consideration any variables not directly involved with the parse tree in question; we need only divide each individual symbol and relation probability by the corresponding probability that that variable would be NIL. As a result, only linearly many variables are used in the multiplication, since parse tree size is linear in the number of input strokes. Note that such a division is not required for the expression variables E_o because of their deterministic nature: if o is not a component of the recursive partition used by the parse tree, then $E_o = \text{NIL}$ with probability 1 and so need not be included in the calculation.

A caveat of this division is that, to meaningfully compare the scores of different parse trees, the probability

$$P(R_{o_1, o_2} = \text{NIL} \mid E_{o_1}, E_{o_2}, f_{o_1}, f_{o_2})$$

must not depend on the values of E_{o_1}, E_{o_2} , as mentioned above. Otherwise the constant elided by the proportionality symbol would vary by changing the parse tree. In practice, when computing N for the relation NIL probability described in Section 8.2.4, we fix $e_1 = e_2 = \text{GEN}$ when querying the relation classifier to obtain the p_r values.

Chapter 9

System evaluation

We evaluated the MathBrush recognizer on two data sets using different methodologies. In the first, the system was tested on the data collected during our 2009 study [20]. These tests allow us to compare the accuracy of the recognizer’s current incarnation with that of previous versions via a user-oriented accuracy metric which is described below. In the second scenario, we test the recognizer on the data from 2011 and 2012 Competitions on Recognition of Handwritten Mathematical Expressions (CROHME), and measure its accuracy using the CROHME accuracy metrics. This methodology allows us to evaluate our system on a subsystem-by-subsystem basis, and to compare our system against several others created by researchers around the world.

9.1 Evaluation on Waterloo corpus

9.1.1 Correction count metric

Devising objective metrics for evaluating the real-world accuracy of a recognition system is difficult. Several authors have proposed accuracy metrics (e.g., [39, 5, 16, 8, 27]) based on implementation details specific to their particular recognizers. It is therefore difficult to directly compare their evaluation results to one another, or to apply their methodologies to our evaluation.

Some authors have proposed recognizer-independent evaluation schemes that rely only on the output of a recognizer, and treat its implementation as a black box. Awal et al. [3] proposed an approach in which the accuracy of symbol segmentation, symbol recognition, relation, and expression recognition are reported separately. A positive feature of this approach is that it makes clear, in the case of incorrect expression recognition, which recognizer subsystem failed, and a version of it was used for the CROHME competition (no relation accuracy measurements were used). Sain et al. [28], following the intuition of Garain and Chaudhuri [8] that errors far from the dominant baseline are less important than those on or near the main baseline, proposed a scheme called EMERS. In it, the edit distance between parse trees representing recognized and ground-truth expressions

measures the accuracy of a recognition result. Edits are weighted so that those more deeply-nested in the parse tree have less cost.

These approaches are both valuable in that they are easily implemented, permit objective comparisons between recognizers, and provide valuable feedback to recognizer developers. But they both measure the amount by which a recognized expression deviates from its ground-truth, and do not consider whether the ground-truth was achievable by the recognizer at all. If a recognizer consistently fails to recognize a particular structure or symbol, the deviation from ground-truth may be small, even though the recognizer is effectively useless for that recognition task.

Because our recognizer is designed specifically for real-time, interactive use within the MathBrush pen-math system, we believe that a user-oriented accuracy model provides the best way to assess its performance. So as well as asking “Is the recognition correct?”, we want to ask not “How many symbols were wrong?” or “How many transformation steps give the correct answer”, but “Is the desired result attainable?”, and “How much effort must the user expend to get it?” To a user, it is not necessarily the case that an error on the main baseline (say, recognizing $a + b$ as atb) is more incorrect than one on a secondary baseline (say, recognizing $n^{2^{1-\epsilon}}$ as $n^{2^{t-\epsilon}}$).

To answer these questions, we count the number of corrections that a user would need to make to a recognition result in order to obtain the correct parse. If an input is recognized correctly, then it requires zero corrections. Similarly, if it is recognized “almost correctly”, it requires fewer corrections than if the recognition is quite poor. This 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 math recognizer, for instance, count the number of required corrections, and obtain comparable measurements. Our evaluation scheme thus provides an abstract way to compare the performance of recognition systems without direct reference to their implementation details.

Liu et al. [19] also devised a user-based correction cost model for evaluating a diagram recognition system. They measured the physical time required to correct the errors in recognized diagrams. This approach would also be useful for evaluating our system, but the size of the expression corpus makes it impractical to manually test and correct the recognition results.

Instead, we have automated the evaluation process. We developed a testing program that simulates a user interacting with the recognizer. The program passes ink representing a math expression to the recognizer. Upon receiving the recognition results, the program compares them to the ground-truth associated with the ink. If the highest-ranked result is not correct, then the testing system makes corrections to the recognition results, as a user would, to obtain the correct interpretation. That is, the system browses through lists of alternative interpretations for subexpressions or symbols, searching for corrections matching the symbols and structures in the ground-truth. It returns the number of corrections required to obtain the correct expression.

Algorithm 16 outlines this process. The recognizer uses a “context” to refer to any node in the shared parse forest. So, as the algorithm descends into an expression tree, it can

request alternatives for any particular subexpression, and having any particular semantics. For example, if an expression intended to be $a + c$ was instead recognized as $a + C$, the algorithm would detect the correct top-level structure, and the correct expression on the left side of the addition sign. On the right side, it would request alternatives “in context”; that is, using only the ink related to the c symbol, and being feasible for the right side of an addition expression, as determined by the grammar.

Algorithm 16 $cc(g, e)$: Count the number of corrections required to match recognition results to ground-truth.

Require: A recognizer R , a ground-truth expression g , and the first recognition alternative e from R .

if $e = g$ **then**
 return 0

$errorhere \leftarrow 0$ (*Indicates whether an error appears in the top level of the expression tree*)

while e has different top-level semantics from g or uses a different partition of the input **do**
 $errorhere \leftarrow 1$
 $e \leftarrow$ the next alternative from R in this context
 if e is null **then**
 (*R provides no more alternatives*)
 return ∞

for each pair of subexpressions e_i, g_i of e, g **do**
 $n_i \leftarrow$ recurse on R, g_i, e_i
 if $n_i = \infty$ **then**
 return ∞

return $errorhere + \sum n_i$

The correction count produced by this algorithm is akin to a tree edit distance, except that it is constrained so that the edits must be obtainable through the recognizer’s output. They cannot be arbitrary terminal or subtree substitutions.

9.1.2 Methodology and results

To facilitate meaningful comparison with previous results, we have replicated the experiments performed in a previous publication [22] as closely as possible. These tests used the probabilistic scoring function.

The correction count metric provides accuracy scores for both correct and incorrect recognition results, but there are some types of recognition errors that it cannot account for. For example, if an expression is recognized correctly except for one symbol, for which the correct alternative is not available, then the correction count will be ∞ , even though the expression is “nearly correct”.

To reduce the number of these types of results, we tested the recognizer in two scenarios. In the first, called *default*, we divided the 3674 corpus transcriptions into training and

testing sets. The training set contained all 1536 transcriptions having fewer than four symbols. The remaining 2138 transcriptions formed the testing set. These transcriptions contained between four and 23 symbols each, with an average of seven. All of the symbols were extracted from the training set and used to augment our pre-existing symbol database. The pre-existing database contained samples of each symbol written by one to three writers, and is unrelated to the evaluation data set. It was used to ensure baseline coverage over all symbol types. Because the training transcriptions contained at most a few symbols, they did not yield sufficient data to adequately train the relation classifiers using semantic labels. We therefore augmented the relation classifier training data with transcriptions from a 2011 collection study [21]. (The symbol recognizer was not trained on this data.)

The second scenario, called *perfect*, evaluated the quality of expression parsing in isolation from symbol recognition. In it, the same training and testing sets were used as in the default scenario, but the terminal symbols were extracted directly from ground truth, bypassing both the stroke grouping and symbol recognition modules. There were thus no alternative symbols for the parser to choose between, and no ambiguities in stroke grouping.

Each recognized transcription may be placed into one of the following categories:

1. *Correct*: No corrections were required. The top-ranked interpretation was correct.
2. *Attainable*: The correct interpretation was obtained from the recognizer after one or more corrections.
3. *Incorrect*: The correct interpretation could not be obtained from the recognizer.

In the original experiment, the “incorrect” category was divided into both “incorrect” and “infeasible”. The infeasible category counted transcriptions for which the correct symbols were not identified by the symbol recognizer, making it easier to distinguish between symbol recognition failures and relation classification failures. In the current version of the system, many symbols are recognized through a combination of symbol and relation classification, so the distinction between feasible and infeasible is no longer useful. We have therefore merged the infeasible and incorrect categories for this experiment.

Figure 9.1 shows the recognizer’s accuracy and average correction count in the default scenario for both the current version and the version used in our previous experiments [22]. Figure 9.2 shows similar information for the perfect scenario.

In the default scenario, just over 33% of transcriptions were recognized correctly, a further 47% were attainable with corrections (about 0.85 corrections per attainable transcription, on average), and about 20% were incorrectly recognized. The rates of attainability and correctness were both significantly higher for the probabilistic recognizer than for our earlier fuzzy variant (80% vs. 67% and 33% vs. 19%, respectively). The average number of corrections required to obtain the correct interpretation was also lower for the new recognizer.

In the perfect scenario, about 96% of transcriptions were attainable (requiring about 0.11 corrections on average) with 85% being correct. The previous recognizer achieved

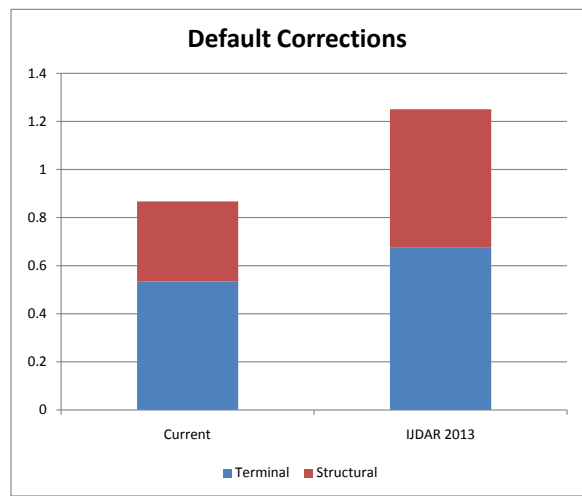
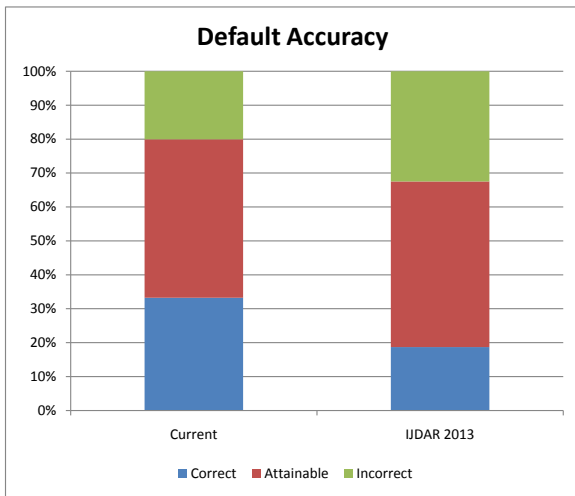


Figure 9.1: Default scenario results.

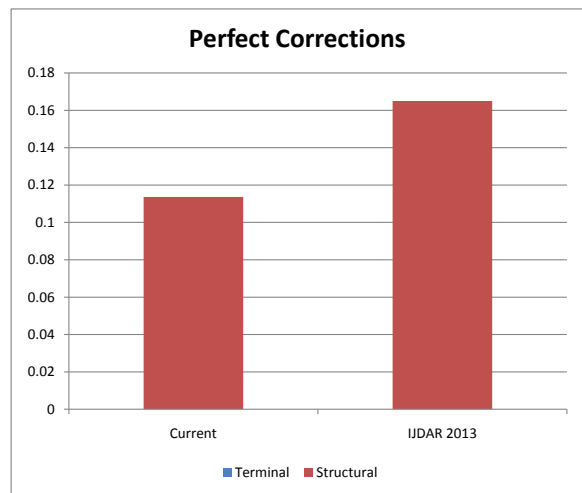
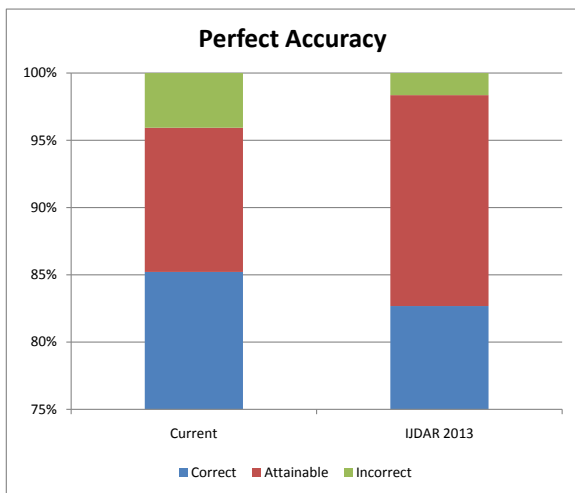


Figure 9.2: Perfect scenario results.

comparable rates of 98% attainable (just over 0.16 corrections on average) and 83% correct. The current version again achieved a somewhat higher correctness rate, but in this scenario had a slightly lower attainability rate. Rather than this indicating a problem with the probabilistic recognizer, it likely indicates that the hand-coded rules of the fuzzy relation classifier were too highly tuned for the 2009 training data. The large changes in the accuracy rates of the rule-based relation classifier between the 2009 and 2011 testing data in Section 5.4 support this point.

9.2 Evaluation on the CROHME corpora

Since 2011, the CROHME math recognition competition has invited researchers to submit recognition systems for comparative accuracy evaluation. In previous work, we compared the accuracy of a previous version of the MathBrush recognizer against that of the 2011 entrants [22]. We also participated in the 2012 competition, placing second behind a corporate entrant [26]. In this section we will evaluate the current MathBrush recognizer described in this thesis against its previous version as well as the other CROHME entrants.

The 2011 CROHME data is divided into two parts, each of which includes training and testing data. The first part includes a relatively small selection of mathematical notation, while the second includes a larger selection. For details, refer to the competition paper [25]. The 2012 data is similarly divided, but also includes a third part, which we omit from this evaluation as it included notations not used by MathBrush (namely notations from boolean logic such as overbar negations, etc.)

For each part of this evaluation, we trained our symbol recognizer on the same base data set as in the previous evaluation and augmented that training data with all of the symbols appearing in the appropriate part of the CROHME training data. The relation and grouping systems were trained on the 2011 Waterloo corpus. We used the the grammars provided in the competition documentation by converting them into the file format recognized by our parser.

The accuracy of our recognizer was measured using a perl script provided by the competition organizers. In this evaluation, only the top-ranked parse was considered. There are four accuracy measurements. Stroke reco. indicates the percentage of input strokes which were correctly recognized and placed in the parse tree. Symbol seg. indicates the percentage of symbols for which the correct strokes were properly grouped together. Symbol reco. indicates the percentage of symbol recognized correctly, out of those correctly grouped. Finally, expression reco. indicates the percentage of expressions for which the top-ranked parse tree was exactly correct (corresponding to a “correct” result in our classification scheme for the previous evaluation).

In Tables 9.1 and 9.2, the first two rows show the test results for the current version of MathBrush using the fuzzy and probabilistic scoring funtions. In the first (2011) table, the third row shows evaluation results the version of our recognizer used for CROHME 2012, the fourth row shows results from a previous publication [22], and the remainder of

the table is reproduced from the CROHME 2011 report [25]. In the second (2012) table, all rows but the first two are reproduced from the CROHME 2012 report [26].

Part 1 of corpus				
Recognizer	Stroke reco.	Symbol seg.	Symbol reco.	Expression reco.
MathBrush prob.	93.16	97.64	95.85	67.40
MathBrush fuzzy	92.20	96.54	95.65	62.43
MathBrush (2012)	88.13	96.10	92.18	57.46
MathBrush (2011)	71.73	84.09	85.99	32.04
Rochester Institute of Tech.	55.91	60.01	87.24	7.18
Sabanci University	20.90	26.66	81.22	1.66
University of Valencia	78.73	88.07	92.22	29.28
Athena Research Center	48.26	67.75	86.30	0.00
University of Nantes	78.57	87.56	91.67	40.88
Part 2 of corpus				
Recognizer	Stroke reco.	Symbol seg.	Symbol reco.	Expression reco.
MathBrush prob.	91.64	97.08	95.53	55.75
MathBrush fuzzy	88.85	94.99	95.08	48.28
MathBrush (2012)	87.08	95.47	92.20	47.41
MathBrush (2011)	66.82	80.26	86.11	20.11
Rochester Institute of Tech.	51.58	56.50	91.29	2.59
Sabanci University	19.36	24.42	84.45	1.15
University of Valencia	78.38	87.82	92.56	19.83
Athena Research Center	52.28	78.77	78.67	0.00
University of Nantes	70.79	84.23	87.16	22.41

Table 9.1: Evaluation on CROHME 2011 corpus.

These results indicate that the MathBrush recognizer is significantly more accurate than all other entrants except the Vision Objects recognizer, which is much more accurate again. (Vision Objects is a French company based in Nantes). They are of more interest for our purpose because they allow us to roughly compare different versions of the MathBrush recognizer on a subsystem-by-subsystem basis. The comparison can only be approximate because correctness is evaluated based on the single top-ranked interpretation of the input, which depends on a complex combination of the individual subsystems.

The accuracy of the stroke grouping system is roughly equivalent to the 2012 version of MathBrush. This is expected as the algorithms used are very similar. The symbol recognizer in the 2012 was a preliminary version of the hybrid recognizer described here with a simpler version of the offline classifier; the changes have led to a modest improvement in accuracy (roughly 3% in absolute terms, but a 40-50% reduction in errors). It is difficult to determine exactly how much of the improvement in the expression recognition rate is due to symbol recognition as compared with the probabilistic relation classifier or the overall probabilistic scoring function, both of which were not present in the 2012 recognizer, which used the rule-based relation classifier and the fuzzy scoring function. But given the magnitude of some of the improvements, especially in Part 1 of these corpora, it seems unlikely that all of the improvement was due to increased symbol recognition accuracy.

Part 1 of corpus				
Recognizer	Stroke reco.	Symbol seg.	Symbol reco.	Expression reco.
MathBrush prob.	91.54	96.56	94.83	66.36
MathBrush fuzzy	90.72	95.56	93.96	56.07
MathBrush (2012)	89.00	97.39	91.72	51.85
University of Valencia	80.74	90.74	89.20	35.19
Athena Research Center	59.14	73.31	79.79	8.33
University of Nantes	90.05	94.44	95.96	57.41
Rochester Institute of Tech.	78.24	92.81	86.62	28.70
Sabancı University	61.33	72.11	87.76	22.22
Vision Objects	97.01	99.24	97.80	81.48
Part 2 of corpus				
Recognizer	Stroke reco.	Symbol seg.	Symbol reco.	Expression reco.
MathBrush prob.	92.74	95.57	97.05	55.18
MathBrush fuzzy	89.82	94.49	95.73	42.47
MathBrush (2012)	90.71	96.67	94.57	49.17
University of Valencia	85.05	90.66	91.75	33.89
Athena Research Center	58.53	72.19	86.95	6.64
University of Nantes	82.28	88.51	94.43	38.87
Rochester Institute of Tech.	76.07	89.29	91.21	14.29
Sabancı University	49.06	61.09	88.36	7.97
Vision Objects	96.85	98.71	98.06	75.08

Table 9.2: Evaluation on CROHME 2012 corpus.

Chapter 10

Conclusion

This thesis described the theory and implementation details underlying the MathBrush math recognizer. Building from low-level distance metrics and classification routines, we detailed the components and combination strategies underlying our sophisticated and accurate recognition system. This chapter summarizes the thesis, pointing out primary contributions and offering thoughts on future research directions.

10.1 MathBrush requirements

Chapter 1 listed the three requirements placed upon the recognizer by the MathBrush system in which it is used:

- it must be reasonably fast,
- it must be trainable, and
- it must support user corrections.

We made significant efforts to ensure that the recognizer is fast enough for real-time use, including the linear-time variant of elastic matching (Section 4.1.2), the ordering assumption and corresponding restriction to rectangular sets (Section 7.2.1), and the optimizations applied to Unger’s algorithm (Section 7.2.2). In practice, the recognizer is quick enough to use comfortably on the Tablet PC platform for most inputs. However, on less powerful tablet devices (such as Android tablets and the iPad) and for large expressions, there is often a delay after adding a stroke to the input. User studies would help to determine the impact of such delays on the input process. If they are a significant impediment to effective use of MathBrush, it would be worthwhile to investigate additional optimizations, including:

- Judicious use of pruning to avoid invoking the parser on input subsets or non-terminals which are unlikely to yield useful results.

- A fast pre-processing step which could avoid invoking the full parsing algorithm by first considering a small number of likely parses incorporating a new input stroke.
- Using a single symbol recognition algorithm (rather than four), and only invoking the hybrid classifier in case of ambiguity. This could be driven by data collected about recognizer behaviour.

To facilitate user training, our symbol recognizer easily accomodates new symbol styles by adding them to the symbol library, as described in Section 4.1. This process is made fairly simple by the MathBrush interface, which prompts the user to write the symbol they wish to train in a few on-screen ink collection areas. The training process for the relation classifier is much less straightforward and is not exposed to users. However, the corrections which users make to change erroneous recognition results are a rich and currently un-tapped source of training information. Future research should investigate how those corrections can be used to create custom-trained writer-dependent recognizers, as well as how those corrections may be pooled to improve the writer-independent baseline recognizer.

The correction process itself is made possible by our use of parse forests (Section 7.1), the tree extraction algorithms described in Section 7.3, and the scoring models of Chapter 8. Because the parse forest generally represents exponentially many trees in the input size, the “semantics mode” of tree extraction is vital for keeping the number of displayed results manageable for users. All the same, the correction count employed in Section 9.1 as a user-centric accuracy metric was measured automatically, rather than by observing real users. User studies would indicate to what extent our correction system is truly an improvement over erasing and re-writing when input is mis-recognized.

10.2 Low-level classifiers

Chapters 3, 4, and 5 respectively discussed our approaches to stroke grouping, symbol recognition, and relation classification. We developed a novel stroke-grouping system based on translating logic- and rule-based heuristics into the language of binary random variables. This system reported false negatives only about 0.5% of the time, was correct by the strictest definition about 90% of the time, and obtained the correct grouping result in a practical context in about 96% of cases.

We proposed a hybrid symbol classification method using four underlying distance-based classifiers, including a novel time- and space-optimized variant of the well-known elastic matching algorithm. The hybrid method attained a correctness rate of about 88% on isolated symbols, an improvement on the correctness rates of the underlying classifiers, which ranged from about 75-85%. During testing on the CROHME data set, the recognizer reached even higher correctness rates of 95-97%; however these results were on a smaller set of symbols than is present in the Waterloo corpus.

The correctness rates of both of these systems must be improved in order to improve overall recognition quality. As well as developing new recognition algorithms or adapting existing methods to a mathematical context, improvement may be possible through

combining the grouping and recognition modules more closely. Whether a collection of strokes should be considered a symbol certainly depends on whether those strokes actually look like a known symbol, yet that information is not taken into account in our system (except in the case of square root signs to determine whether stroke overlap is acceptable or not). The main challenge in combining the two systems is to maintain real-time performance. Symbol recognition is an expensive step, and applying it to every stroke subset the grouping system considers is not practical.

Several techniques were proposed and evaluated for relation classification, including a rule-based heuristic approach and three probabilistic techniques: a naive Bayesian model using Gaussian distributions, a grid-based discretization strategy, and a second discretization method using the margin tree data structure which we developed. Both the discretization techniques performed very well in isolated testing with correctness rates of about 95%. But in the full context of math recognition, they proved less able to distinguish between cases where a relation existed and did not exist, so we use the naive Bayesian approach in practice. The majority of relation classification errors occur on the horizontally-oriented relations \nearrow , \rightarrow , and \searrow because of the natural ambiguity of bounding box features. Improvements using the same feature set should rely on more context-sensitive models (which in turn require more training data). Given the effectiveness of the quantile-based method used for combining symbol classifiers, it seems worthwhile to apply that technique to the relation classifiers as well. Also, the present classifiers use only information local to the stroke groups in question. Considering how the expressions resulting from each candidate relation would fit into the larger picture may also yield improvements. During our experiments with fuzzy sets, estimating global baselines and measuring how well a given relation matched its expected placement with respect to the baseline yielded modest improvements in relation classification accuracy. Finally, the hybrid symbol recognizer described in Chapter 4 was more accurate than any of the underlying individual recognizers. A similar quantile-based combination method may improve accuracy for the relation classification task as well.

10.3 Grammars and parsing

Chapter 6 developed a novel relational grammar formalism offering sophisticated control over subexpression combination through linkage parameters. The linkage parameters are quite powerful, and we observed a significant increase in recognition accuracy after incorporating them into our grammar model. We adapted familiar notions such as strings and languages to our model, and described how the grammar formalism relates to the recognition process. Chapter 7 built efficient parsing algorithms for this grammar model, inspired by classic CFG parsing algorithms. By introducing some natural assumptions on the structure of the input and scoring algorithms, we attained acceptable performance for real-time use. The algorithms maintain a user-centric design by making available all recognizable interpretations of the input and organizing those results hierarchically so that the user may select alternative recognition results rather than re-writing their expression if recognition is incorrect.

The ordering assumption is reasonable in theory, but is sometimes violated in practice,

even on expressions that are easily readable by people. It would be beneficial to eliminate or weaken the assumption as much as possible while still achieving fast recognition speed and avoiding consistent worst-case behaviour as in the CYK parsing algorithm. A well-specified and efficient method of identifying all “reasonable” combinations of rectangular sets which cover the input would improve accuracy without unduly affecting performance.

10.4 Scoring methods

Finally, Chapter 8 detailed two scoring functions on which to base the rank-based reported scheme of Chapter 7. The fuzzy scoring function simply treats the sets of interpretations introduced in Chapter 6 as fuzzy sets and define a membership function on them using the geometric average. The probabilistic scoring function treats the expressions and symbols recognized on input subsets as random variables and organizes them in a Bayesian model. An algebraic technique dramatically reduces the number of variables it is necessary to consult when evaluating the score of a given parse tree.

In our evaluation, the probabilistic model was more effective than the fuzzy model. This is not surprising, given its greater sophistication and inclusion of more details (e.g., symbol co-occurrence counts, NIL probabilities, etc.). The relative ease with which such details can be systematically included in an expression’s score is a real strength of the probabilistic approach, and the addition of more real-world data would likely further improve performance. We have developed a preliminary implementation that assigns a prior distribution to the expression variables based on known subexpression occurrences and so on. (E.g., addition expressions are more likely to contain subscripts than function names, etc.) But we currently lack sufficient data for such a model to be useful. The number of potential subexpression combinations is very large, and the resulting look-up tables are so sparsely populated that they give nonsense answers to queries even when using smoothing. Nonetheless, this remains a promising idea for the future, when larger collections of math expressions may be available.

References

- [1] Francisco Álvaro, Joan-Andreu Sánchez, and José-Miguel Benedí. Recognition of printed mathematical expressions using two-dimensional stochastic context-free grammars. In *Proc. of the Int'l. Conf. on Document Analysis and Recognition*, pages 1225–1229, 2011.
- [2] Robert H. Anderson. *Syntax-Directed Recognition of Hand-Printed Two-Dimensional Mathematics*. PhD thesis, Harvard University, 1968.
- [3] A.-M. Awal, H. Mouchère, and C. Viard-Gaudin. The problem of handwritten mathematical expression recognition evaluation. In *Proc. of the Int'l. Conf. on Frontiers in Handwriting Recognition*, pages 646–651, 2010.
- [4] Dorothea Blostein. Math-literate computers. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Calcuemus/MKM*, volume 5625 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2009.
- [5] Kam-Fai Chan and Dit-Yan Yeung. Error detection, error correction and performance evaluation in on-line mathematical expression recognition. In *in On-Line Mathematical Expression Recognition,*” *Pattern Recognition*, 1999.
- [6] Kam-Fai Chan and Dit-Yan Yeung. Error detection, error correction and performance evaluation in on-line mathematical expression recognition. *Pattern Recognition*, 34(8):1671 – 1684, 2001.
- [7] U. Garain and B.B. Chaudhuri. Recognition of online handwritten mathematical expressions. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(6):2366–2376, Dec. 2004.
- [8] Utpal Garain and B. Chaudhuri. A corpus for ocr research on mathematical expressions. *Int. J. Doc. Anal. Recognit.*, 7(4):241–259, 2005.
- [9] Oleg Golubitsky and Stephen M. Watt. Online computation of similarity between handwritten characters. In *Proc. Document Recognition and Retrieval XVI*, 2009.
- [10] Dick Grune and Cerial J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer, 2 edition, 2008.

- [11] David Hoaglin, Frederick Mostaller, and John Tukey (editors). *Understanding Robust and Exploratory Data Analysis*. John Wiley & Sons, 1983.
- [12] Jesse Hull. Recognition of mathematics using a two-dimensional trainable context-free grammar. Master’s thesis, Massachusetts Institute of Technology, 1996.
- [13] G. Labahn, E. Lank, S. MacLean, M. Marzouk, and D. Tausky. MathBrush: A system for doing math on pen-based devices. *The Eighth IAPR Workshop on Document Analysis Systems (DAS)*, pages 599–606, 2008.
- [14] G. Labahn, E. Lank, M. Marzouk, A. Bunt, S. MacLean, and D. Tausky. MathBrush: A case study for interactive pen-based mathematics. *Fifth Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM)*, June 11-13 2008.
- [15] Bernard Lang. Towards a uniform formal framework for parsing. In *Current Issues in Parsing Technology*, pages 153–171. Kluwer Academic Publishers, 1991.
- [16] Joseph J. Laviola, Jr. *Mathematical sketching: a new approach to creating and exploring dynamic illustrations*. PhD thesis, Brown University, 2005.
- [17] Joseph J. LaViola, Jr. and Robert C. Zeleznik. Mathpad2: a system for the creation and exploration of mathematical sketches. In *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, pages 432–440, New York, NY, USA, 2004. ACM.
- [18] Percy Liang, Mukund Narasimhan, Michael Shilman, and Paul Viola. Efficient geometric algorithms for parsing in two dimensions. In *ICDAR ’05: Proceedings of the Eighth International Conference on Document Analysis and Recognition*, pages 1172–1177, Washington, DC, USA, 2005. IEEE Computer Society.
- [19] W. Liu, L. Zhang, L. Tang, and D. Dori. Cost evaluation of interactively correcting recognized engineering drawings. *Lecture Notes in Computer Science*, 1941:329–334, 2000.
- [20] S. MacLean, G. Labahn, E. Lank, M. Marzouk, and D. Tausky. Grammar-based techniques for creating ground-truthed sketch corpora. *Int’l. J. Document Analysis and Recognition*, 14:65–74, 2011.
- [21] S. MacLean, D. Tausky, G. Labahn, E. Lank, and M. Marzouk. Is the iPad useful for sketch input?: a comparison with the Tablet PC. In *Proceedings of the Eighth Eurographics Symposium on Sketch-Based Interfaces and Modeling, SBIM ’11*, pages 7–14, 2011.
- [22] Scott MacLean and George Labahn. A new approach for recognizing handwritten mathematics using relational grammars and fuzzy sets. *International Journal on Document Analysis and Recognition (IJDAR)*, 16(2):139–163, 2013.
- [23] Christopher D. Manning and Hinrich Schuetze. *Foundations of Statistical Natural Language Processing*. The MIT Press, 1999.

- [24] Erik G. Miller and Paul A. Viola. Ambiguity and constraint in mathematical expression recognition. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 784–791, 1998.
- [25] H. Mouchère, C. Viard-Gaudin, U. Garain, D.H. Kim, and J.H. Kim. CROHME2011: Competition on recognition of online handwritten mathematical expressions. In *Proc. of the 11th Int'l. Conference on Document Analysis and Recognition*, 2011.
- [26] H. Mouchère, C. Viard-Gaudin, U. Garain, D.H. Kim, and J.H. Kim. Competition on recognition of online handwritten mathematical expressions (CROHME 2012). In *Proc. of the 11th Int'l. Conference on Frontiers in Handwriting Recognition*, 2012.
- [27] I. Rutherford. Structural analysis for pen-based math input systems. Master's thesis, David R. Cheriton School of Computer Science, University of Waterloo, 2005.
- [28] K. Sain, A. Dasgupta, and U. Garain. Emers: a tree matching-based performance evaluation of mathematical expression recognition systems. *IJDAR*, 14(1):75–85, 2011.
- [29] Yu Shi, HaiYang Li, and F.K. Soong. A unified framework for symbol segmentation and recognition of handwritten mathematical expressions. In *Document Analysis and Recognition, Ninth International Conference on*, pages 854–858, 2007.
- [30] Masakazu Suzuki, Seiichi Uchida, and Akihiro Nomura. A ground-truthed mathematical character and symbol image database. *Document Analysis and Recognition, International Conference on*, pages 675–679, 2005.
- [31] Charles C. Tappert. Cursive script recognition by elastic matching. *IBM Journal of Research and Development*, 26(6):765–771, 1982.
- [32] Arit Thammano and Sukhumal Rugkunchon. A neural network model for online handwritten mathematical symbol recognition. *Lecture Notes in Computer Science*, 4113, 2006.
- [33] Stephen H. Unger. A global parser for context-free phrase structure grammars. *Commun. ACM*, 11:240–247, April 1968.
- [34] Merijn van Erp and Lambert Schomaker. Variants of the borda count method for combining ranked classifier hypotheses. In L.R.B. Schomaker and L.G. Vuurpijl, editors, *Proceedings of the Seventh Int'l. Workshop on Frontiers in Handwriting Recognition*, pages 443–452, 2000.
- [35] H.-J. Winkler, H. Fahrner, and M. Lang. A soft-decision approach for structural analysis of handwritten mathematical expressions. In *Proc. International Conference on Acoustics, Speech and Signal Processing*, pages 2459–2462, 1995.
- [36] R. Yamamoto, S. Sako, T. Nishimoto, and S. Sagayama. On-line recognition of handwritten mathematical expression based on stroke-based stochastic context-free grammar. In *The Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

- [37] L.A. Zadeh. Fuzzy sets. *Information Control*, 8:338–353, 1965.
- [38] R. Zanibbi, D. Blostein, and J.R. Cordy. Baseline structure analysis of handwritten mathematics notation. In *Document Analysis and Recognition, 2001. Proceedings. Sixth International Conference on*, pages 768–773, 2001.
- [39] R. Zanibbi, D. Blostein, and J.R. Cordy. Recognizing mathematical expressions using tree transformation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(11):1455–1467, Nov 2002.
- [40] Robert Zeleznik, Timothy Miller, Chuanjun Li, and Joseph J. Laviola, Jr. Mathpaper: Mathematical sketching with fluid support for interactive computation. In *SG '08: Proceedings of the 9th international symposium on Smart Graphics*, pages 20–32, Berlin, Heidelberg, 2008. Springer-Verlag.
- [41] Ling Zhang, D. Blostein, and R. Zanibbi. Using fuzzy logic to analyze superscript and subscript relations in handwritten mathematical expressions. In *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*, pages 972–976 Vol. 2, 2005.