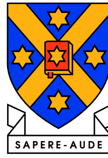
Library Declaration Form



University of Otago Library

Author's full name and year of birth: Steven Ross Smithies,
(for cataloguing purposes) 1976

Title of thesis: Freehand Formula Entry System

Degree: Master of Science

Department: Computer Science

Permanent Address: 4B Fairbanks Place, Glendene, Auckland 1008

I agree that this thesis may be consulted for research and study purposes and that reasonable quotation may be made from it, provided that proper acknowledgement of its use is made.

I consent to this thesis being copied in part or in whole for

i) a library

ii) an individual

at the discretion of the University of Otago.

Signature:

Date:

Freehand Formula Entry System

Steve Smithies

a thesis submitted for the degree of

Master of Science

at the University of Otago, Dunedin,

New Zealand.

May 18, 1999

Abstract

Current equation editing systems rely either on text-based equation description languages or on interactive construction by means of selecting and filling in templates. These systems are often tedious to use, even for experts, because the user is forced to determine the structure of the formula before entry.

This thesis describes a system that enables the freehand entry and editing of formulae using a pen and tablet. The raw input strokes are passed through a new algorithm for automatically segmenting them into symbols. It uses a character recogniser to evaluate different possible groupings. The user interface has tools for easy correction of the inevitable errors occurring in the grouping and recognition stage. A pop up menu offers character recognition alternatives while stroke segmentation errors are corrected by drawing a temporary line through the strokes that belong to a single character. The recognised symbols can be passed through a graph rewriting formula parser, producing a linear command string representation of the formula.

A user test was designed and conducted to evaluate the effectiveness of the user interface and the effectiveness of a graph rewriting parser for processing handwritten input. It was found that a usable pen-based formula entry system can be built and, more importantly, is preferable to use over existing template or command-string based systems, although the character recognition and formula processing modules in the system need improvement before it would be of commercial value.

Acknowledgements

I wish to thank my supervisor, Dr. Kevin Novins, for all the guidance, ideas, motivation and help he gave me while I was working on my Masters and writing this thesis. Second, thanks to everyone in the Graphics Lab for the fun, ideas and help that have gone into this system over the past year. Parts of Chapter 4 of this thesis was originally part of a paper submitted to Graphics Interface '99, so I would like to say thanks to my coauthors on that paper: Kevin Novins and Jim Arvo, for the work they did helping write the paper and, as a result, parts of Chapter 4. Last, but not least, I would like to thank my family and all my friends for providing the essential friendship, support and encouragement I needed over the year.

Contents

1 Introduction **1**
1.1 Background . . . 1

2 Literature Review **5**
2.1 Existing Formula Entry Methods . . . 6
2.1.1 Command Line Interfaces . . . 6
2.1.2 Template-style Editors . . . 8
2.1.3 Graphical Online Pen Entry Systems . . . 9
2.2 Issues In Formula Recognition . . . 10
2.2.1 Input . . . 11
2.2.2 Noise Versus Small Symbols . . . 12
2.2.3 Symbol Segmentation and Recognition . . . 12
2.2.4 Ambiguous Symbols . . . 13
2.2.5 Identifying Significant Spatial Relationships . . . 16
2.2.6 Ambiguity of Symbol Placement . . . 17
2.2.7 Little Redundancy . . . 17
2.2.8 Connected and Overlapping Symbols . . . 17
2.2.9 Ambiguity in the Formula . . . 18
2.2.10 Post-processing Error Correction Rules . . . 21
2.3 Formula Parsers . . . 21
2.3.1 Modified Grammars . . . 21
2.3.2 Box Languages . . . 22
2.3.3 Projection Profile Cutting . . . 26
2.3.4 Procedurally Coded Math Syntax . . . 27
2.3.5 Stochastic Grammars . . . 28
2.3.6 Graph Rewriting . . . 30
2.3.7 Data Driven and Knowledge Driven Modules . . . 31
2.4 Summary . . . 32

3 The Formula Processor **33**
3.1 Formula Processor Details . . . 35
3.1.1 Bounding Regions . . . 36
3.1.2 Input to the Formula Processor . . . 38
3.1.3 Building the Initial Graph . . . 39
3.1.4 Building the Arcs . . . 39
3.1.5 Initial Graph Preprocessing . . . 45

3.1.6 Main Processing . . . 46
3.1.7 Parser Implementation . . . 49
3.2 Formula Straightening . . . 50
3.3 Summary . . . 51

4 The Interface **53**
4.1 Aspects of User Interface Design . . . 53
4.2 Pen Based Computing . . . 55
4.3 A Pen Based Formula Entry System . . . 55
4.3.1 The Character Recogniser . . . 57
4.3.2 Basic Input . . . 58
4.3.3 Stroke Segmentation . . . 58
4.3.4 Online Annotation . . . 64
4.3.5 Stroke Regrouping . . . 65
4.3.6 Modify Characters . . . 66
4.3.7 Parsing and Preview . . . 68
4.3.8 Correcting Equation Parsing Errors . . . 71

5 User Testing **75**
5.1 Designing the Test . . . 76
5.2 Choosing Participants . . . 77
5.3 Ethical Considerations . . . 79
5.4 The Test Itself . . . 80
5.5 Post-test Analysis . . . 82
5.6 Usability Inspection . . . 83

6 Evaluation **85**
6.1 User Testing . . . 85
6.1.1 Working Styles . . . 86
6.1.2 Time Spent Entering and Correcting Formulae . . . 86
6.1.3 Parsing . . . 91
6.1.4 Comparative Timing Results . . . 93
6.1.5 Error Rates . . . 94
6.1.6 Evaluation of User Interface Features . . . 95
6.1.7 Evaluation of the User Testing . . . 99
6.2 Usability Inspection . . . 100
6.2.1 Visibility of System Status . . . 101
6.2.2 Match Between the System and the Real World . . . 101
6.2.3 User Control and Freedom . . . 101
6.2.4 Consistency and Standards . . . 102
6.2.5 Recognition Rather Than Recall . . . 102
6.2.6 Flexibility and Efficiency of Use . . . 102
6.2.7 Aesthetic and Minimalist Design . . . 102
6.2.8 Help Users Recognise, Diagnose, and Recover From Errors . . . 103
6.2.9 Error Prevention . . . 103
6.2.10 Help and Documentation . . . 103

6.2.11 Overall Degree of Usability . . . 104
6.3 The Overall System . . . 104
6.3.1 Positive . . . 104
6.3.2 Negative . . . 105
6.3.3 Overall . . . 105

7 Future Work **107**
7.1 The Formula Parser . . . 107
7.2 Keyboard Input . . . 108
7.3 Magic Hot-Spots . . . 109
7.4 Indication of Areas . . . 109
7.5 Training of the Character Recogniser . . . 109
7.6 Squiggle Select for Other Selecting Operations . . . 110
7.7 Morphing of Symbols . . . 110

8 Conclusion **113**

References **117**

A Accompanying CD-ROM **123**
A.1 Readme Text File . . . 123
A.2 The Thesis . . . 123
A.3 Quicktime Movie . . . 123
A.4 Tar File . . . 125
A.5 Graphics Interface '99 Paper . . . 125

B Ethical Statement **127**

C Participant Consent Form **131**

D Anonymous Questionnaire **135**

E Oral Questionnaire **139**

F Anonymous Responses **141**

G Oral Responses **147**

H Raw Data **153**
H.1 Error Rates . . . 153
H.2 Parsing Attempts . . . 154
H.3 Drawing and Correction Times . . . 155

List of Figures

2.1 Screenshot of Microsoft's Equation Editor. 9
2.2 Without the context of the surrounding symbols, the identity of a symbol sometimes can not be determined. In (a), it is not possible to determine whether it is an o (the letter o) or 0 (the digit zero). In (b) it is a 0, yet in (c) it is an o. 13
2.3 The relative location of symbols can be ambiguous. (a) represents a times x, and (c) represents a to the power of x, but what about (b)? . . 16
2.4 Overlapping symbols. 18
2.5 A box language summation template. 23
2.6 These y-centres of these symbols line up, although their bounding boxes do not. 24
2.7 Building a projection profile. 26

3.1 Sample rules from a graph grammar. 34
3.2 Construction of bounding regions. 36
3.3 These bounding boxes overlap, although the symbols do not. 37
3.4 Using centre points instead of bounding boxes for geometric tests. While the 1 is outside the fraction bar's bounding box, the 3 is not. 38
3.5 A simple formula. 39
3.6 Preprocessing and parsing of a simple formula. 40
3.7 Regions used by the geometric check. 42
3.8 Is "A" to the top-right of "B"? Yes. 43
3.9 As the length of the exponent grows, it moves from the "top-right" to the "right" region. 44
3.10 The shape of the "right" region that Lavirotte and Pottier use. 44
3.11 No arc is built between symbols that have other symbols between them. 45
3.12 The "nothing inside" test. If a grammar rule collapsed the $\frac{2}{4}$ to a single node, the centre point of the 3 would end up inside its new bounding region. Because of this, the application of the rule is not permitted. . . 48
3.13 A problem with the no-inside restriction. Collapsing the integral is forbidden due to the fraction bar ending up inside it. 49
3.14 A sloped or skewed formula can be difficult to process reliably. 50

4.1 A suggested undo gesture. 55
4.2 A formula that has been entered into, and parsed by, the system. . . . 56
4.3 Delays between strokes for writing the alphabet. 60
4.4 All groupings for four units. 64

4.5 A user beginning to enter a formula. The first three characters have been recognised, and the remaining two are still waiting to be recognised. 65
4.6 Modifying stroke groupings. 67
4.7 Correcting a misrecognised character. 69
4.8 A formula being processed. 70
4.9 The LaTeX preview window. 71
4.10 The display for a formula that the system was unable to parse. 72

5.1 The proportion of problems with a user interface found as the number of evaluators is increased. 78

6.1 Times for people entering Formula 1. 87
6.2 Times for people entering Formula 2. 87
6.3 Times for people entering Formula 3. 88
6.4 Times for people entering Formula 4. 88
6.5 Times for people entering Formula 5. 89
6.6 Times for people entering all the formulae. 89
6.7 Time spent by users entering and correcting formulae. 90
6.8 Part-way through parsing a formula, with an erroneous decision having been made by the parser. 92
6.9 A misinterpreted formula. 92
6.10 Misrecognition and misgrouping rates. 96
6.11 Parsing attempts for each formula. 96
6.12 The overlapping bounding boxes of the square-root symbol and the z are indistinguishable, making it hard to tell at glance if they are correctly grouped. 97

Chapter 1

Introduction

1.1 Background

Mathematical formulae have a two-dimensional nature. A lot of information is conveyed through the relative positions of symbols in a formula: the operation that is being applied to the symbols, or what the arguments to a function are. Original systems for doing mathematics on a computer were developed in the times of character-only interfaces (Littin, 1995). As a result, formulae were entered as a linear text string of "commands". Graphical entry was out of the question as the equipment required was either unavailable, or prohibitively expensive.

A linear command string can represent all the same formulas as a traditional two-dimensional notation. Turning a two-dimensional formula into a linear representation does not incur a loss of information, but it is necessary to know the overall structure before it can be linearised. In effect, the user has to do a mental "parse" of the formula to determine its structure before starting to enter it.

Even with the proliferation of graphic capable displays and pointing devices such as mice or pens and tablets, applications and systems are still produced with the option of, or even restricted to, the use of keyboard entry for formulae.

Entering a formula as a command string is not very difficult if you have the formula in front of you and are just transcribing it, assuming you have a knowledge of the command language you are using. It is only when you are trying to write a formula from a "mental image", or want to manipulate it afterwards, that it becomes difficult.

For example, LaTeX (Lamport, 1994) is a non-WYSIWYG system for creating and typesetting documents, using text commands for all layout and description of the structure of a document. The entry of formulae is also through a command language.

The formula:

$$\int_0^4 x^2 dx$$

is entered into LaTeX as `\int^{4}_{0}{x^2}dx`. While this is not complex, the structure of a formula becomes a lot harder to visualise and manipulate as the length of its command string grows. For example, the formula:

$$f(a) = \frac{1}{2\pi i} \int_0^{2\pi} \frac{f(e^{i\theta})}{T^{-1}(e^{i\theta})} \frac{d}{d\theta}(T^{-1}(e^{i\theta}))d\theta$$

is entered as:

```
f(a) = \frac{1}{2\pi i} \int^{2\pi}_{0}{
\frac{f(e^{i\theta})}{T^{-1}(e^{i\theta})} \frac{d}{d\theta}
(T^{-1}(e^{i\theta})) d\theta}
```

By looking at the command string for this formula, it is difficult to gain an idea of its overall structure. Furthermore, changing the layout of the components in the formula, or performing other major editing operations is very difficult.

The people who are most likely to benefit from the ability to carry out mathematics on a computer are also the ones who are least likely to be interested in typing in complex or unnatural notations for formulae. Mathematicians would have spent many years of their lives using pencil and paper, a method which does allow 2D input and display. In spite of the enormous potential offered by computers for mathematical computation and manipulation, a simple means of entry still does not exist.

While most computer algebra systems have graphical formula entry front-ends available, for example Mathematica (Wolfram, 1996), sometimes this front-end is unavailable for the particular package of interest, or difficult to use. Typically these are template based editors, where the user chooses a template for the operation they desire, and fills in the boxes provided by the user interface. While template systems are a step towards the goal of being able to enter formulae while keeping their two dimensional layout, they are still not as fluid to use as a pencil and paper and still require a mental pre-parse of formulae to determine their structure before entry.

An ideal computer-based mathematics system would be one that had the ease of use of pencil and paper but with the power of a computer behind it. A user would be able to enter their formula as they would write it on a piece of paper: drawing with a pen on a tablet and having their strokes appear on the screen. Once the user had drawn their formula, they would then be able to freely manipulate, add to and change the

formula as they desired. Once the formula was entered to their satisfaction, the system could then insert it into their word processor, LaTeX document, or other application. Should the formula entry system be part of a WYSIWYG system, a typeset version of the formula could be displayed. If it was part of a computer algebra system, they could evaluate or perform other operations such as expansion, simplification, or evaluation of it.

This thesis describes a new system that is a step towards the ideal system described above: the Freehand Formula Entry System (FFES). It allows the freehand entry and editing of formulae using a pen and tablet. Automatic handwritten formula recognition is then used to generate a LaTeX command string for the formula.

The system consists of a number of independent modules:

- *A handwriting recogniser* that recognises the strokes input by the user to determine the symbols written.
- *A formula processor* which takes the list of symbols and their positions, and returns the formula they represent.
- *A user interface* which provides transparent interaction with the handwriting recogniser and formula processor. The user interface provides easy entering and manipulation of formulae, along with the means to correct any errors made by the other components in the system.

The handwriting recognition module was developed as part of a larger system (Smithies, Novins and Arvo, 1999), and the formula processor is an implementation of published techniques (Lavirotte and Pottier, 1997). The synthesis and evaluation of these modules and the user interface are the main contribution of this thesis. The user interface includes a new method for automatically determining the correct grouping of the user's strokes into symbols, and a number of new user interface tools. The tools enable the user to easily correct the inevitable mistakes made by the character recogniser and automatic stroke grouping process.

This thesis also reports on user testing carried out on this system. It was found that a pen-based formula entry system is easier and more comfortable to use over existing formula entry systems, both for the initial entry and subsequent editing of formulae.

Chapter 2 reviews existing formula processors and formula entry systems. Each formula processor is analysed and its suitability for handwritten formula interpretation is discussed. Chapter 3 describes in detail the implementation of the formula processor, a subset of the system described by Lavirotte and Pottier (1997).

Chapter 4 describes the new user interface have designed and implemented for the freehand entry and editing of formulae, including the new techniques developed for automatically grouping a user's strokes, and correcting errors that arise as the user enters their formula.

Chapter 5 describes user testing and the user test of this system was conducted.. Chapter 5 also discusses usability inspections. Chapter 6 is an evaluation of the system, based on results of the user testing and an informal usability study. Chapter 7 discusses directions that further development of the system could take, in response to issues and ideas raised in Chapter 6.

The final chapter, Chapter 8, summarises the findings of this thesis, and suggests some future avenues of research.

Chapter 2

Literature Review

Since the late 1960s a lot of research has been done in the field of parsing mathematical formulae, for a range of purposes from the recognition of a person's scribings with a pen and tablet, through to the automatic interpretation and efficient storage of printed tables of integrals that have been scanned from books. Whatever the purpose, there are a number of approaches that can be used. Some are more suited to typeset while others are more suited to handwritten input.

The main difference between handwritten and typeset input is that typeset input has, at least for a single source, a much more rigorous and predictable nature with regard to its layout and appearance. Handwriting, on the other hand, exhibits a great deal of variability, even for a single author. There are unpredictable variations in the size of symbols, layout, and notational conventions used, which even vary within a single formula. When dealing with handwritten input, it is important to be able to interpret formulae reliably, in spite of minor disturbances in the positions and sizes of symbols.

A good source of general information on formula recognition is "General Diagram Recognition Methodologies" by Blostein (1996), and Chapter 22 of "Handbook of Character Recognition and Document Image Analysis", by Blostein and Grbavec (1996).

This chapter will discuss existing formula entry methods: command line interfaces, template-style editors, and graphical pen and tablet entry systems. For each of these, their strengths and weaknesses are discussed. This chapter then discusses issues in formula recognition that apply to both handwritten and typeset input. Existing formula parsing systems are then presented, with attention paid to their ability to process handwritten input.

2.1 Existing Formula Entry Methods

This section discusses existing methods for entering formulae into a computer. Take the following formula as an example:

$$\int_{10}^{20} \frac{4x^3}{\ln x} dx$$

Either the formula can be turned into some linear form, and then typed at the keyboard, or some system could be used where the 2D nature of the formula is preserved.

The paper by Kajler and Soiffer (1998) gives a good overview of the techniques and considerations involved in making interfaces for computer algebra systems. They predominantly discuss window based, template style entry systems, though they do have a section on alternative input methods, such as pen and tablet or voice.

2.1.1 Command Line Interfaces

Command string based equation entry is fast and powerful for a user who can visualise the formula and is familiar with a system's commands. Command string input is quick as is depends upon typing skills; advanced users will able able to type faster than they can write (Brown, 1988). If a user is able to determine the correct command string for their formula, it does not take them long to type it in.

For a novice user, command string based entry can be frustrating to use while they are overcoming the initial learning curve. A significant amount needs to be learnt before they can enter complex formulae. There is a lot of information that has to be remembered, such as syntax and command names.

In spite of these disadvantages, the use of command string based equation entry is still popular.

LaTeX

LaTeX (Lamport, 1994) is a software system for typesetting documents. It is in common use in many disciplines, including Computer Science, Mathematics and Physics, for publication quality typesetting of documents and mathematical formulae.

Formulae are expressed in LaTeX as textual command strings. As with all command string based systems, the learning curve is steep and many different commands must be learnt to describe the expression and its layout. Even long time users of LaTeX can

find it frustrating as they still have to occasionally refer to books to find out how to achieve the result they desire.

The example formula given at the beginning of this section is entered into LaTeX as:

```
\int^{20}_{10}{\frac{4x^3}{\ln{x}}}dx
```

For longer and more complex formulae, the nesting and balancing of braces can become difficult. It also gets becomes difficult to visualise the formula from its command string. For example, the formula already presented in Section 1.1:

$$f(a) = \frac{1}{2\pi i} \int^{2\pi}_{0} \frac{f(e^{i\theta})}{T^{-1}(e^{i\theta})} \frac{d}{d\theta}(T^{-1}(e^{i\theta}))d\theta$$

is entered into LaTeX as:

```
f(a) = \frac{1}{2\pi i} \int^{2\pi}_{0}{
\frac{f(e^{i\theta})}{T^{-1}(e^{i\theta})} \frac{d}{d\theta}
(T^{-1}(e^{i\theta})) d\theta}
```

Mathematica

Mathematica (Wolfram, 1996), produced by Wolfram Research, is a powerful system for carrying out a wide variety of mathematics. Formulae can be entered in two ways, either as a command-string, or through a template based equation editor. The template based editor is discussed later in Section 2.1.2.

Mathematica's command string based entry shares the same problem as LaTeX: the user has to know the correct "command" for each mathematical operation and symbol. As Mathematica offers an interactive session, online help is available which helps to alleviate this problem.

The example formula is entered for evaluation into Mathematica with the command string:

```
Integrate[(4x^3/Log[x]),{x,10,20}]
```

The language used by Mathematica is simpler and more consistent than that used by LaTeX. Functions have their names fully spelt out, their arguments are contained within a single pair of square brackets and each argument is separated by a comma. LaTeX often uses shortened command names, has pairs of curly braces around each argument and the number of these pairs varies depending on the operation. The variety of brackets used in Mathematica's expression tends to make them more readable and easier to understand. This difference is probably due to the fact that LaTeX's primary purpose is typesetting, while Mathematica is a mathematics package.

LISP-like Prefix Notation

In a LISP-like prefix notation, all components are entered as *operator*, *argument* 1, *argument* 2, ... , *argument* n. If an argument of an operation is another operation, it is enclosed in braces. For example, in a LISP-like prefix notation, the example formula from above is:

```
(Integrate (/ (^ (* 4 x) 3) (Log x)) x 10 20)
```

LISP-like prefix notation is more difficult for a user to enter, but better for computer processing. It also provides a good intermediate representation of formulae, due to its consistency and regular structure.

2.1.2 Template-style Editors

Template-style equation editors require the user to select, either from a toolbar or menu, templates for the mathematical constructs that they wish to use.

All template editors have a similar style of use. Basic operations such as addition and subtraction can be entered from the keyboard by pressing the appropriate key. For operations which are not on the keyboard or are two-dimensional in nature, such as exponentiation, integration, square-root, summation and fractions, the user typically uses the mouse to select from a toolbar the template for the operator they want. The operator then appears on the screen, and the user positions the cursor in boxes, using the TAB key, arrows on the keyboard, or by clicking in the boxes with the mouse, and then fills in the boxes. The boxes are initially placeholders for, and finally contain the operands for the operators.

Figure 2.1 shows a screenshot of a user part way through entering a formula using Microsoft's template based equation editor. Along the top of the window is a toolbar that offers templates for the operations that the system supports. In the main entry area, you can see a partially entered formula. The box in the lower part of the fraction template is yet to be filled.

A number of template based editors exist, typically as part of some larger system. Some examples include:

- Microsoft's Equation Editor. This usually comes as part of Microsoft Word (Microsoft, 1993).

- Mathematica's equation editor for its graphical front-end, produced by Wolfram Research (Wolfram, 1996).

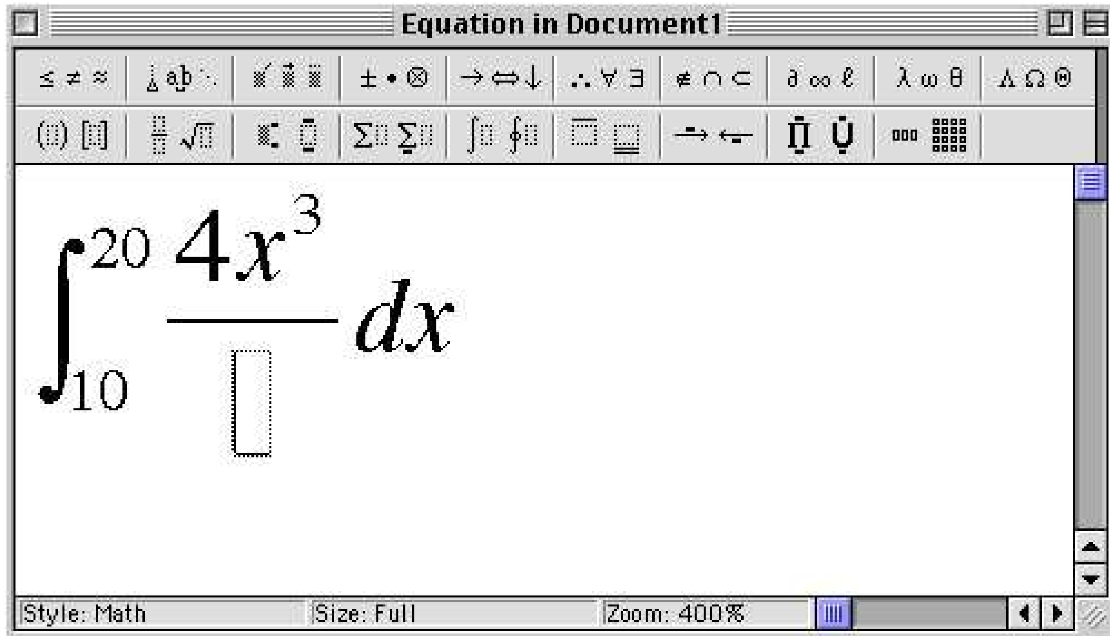


Figure 2.1: Screenshot of Microsoft's Equation Editor.

- LyX (Ettrich, 1999), a front-end for the LaTeX typesetting system. It provides a WYSIWYG interface for the creation and editing of LaTeX documents, and includes a template based equation editor.

- INFORM (van Egmond, Heeman and van Vliet, 1989). An interactive syntax-directed formulae editor.

- Newton (Hayden and Lamagna, 1998) is a tool designed for teaching introductory mathematics. As part of it, there is a formula entry system which is based on a template style of entry.

2.1.3 Graphical Online Pen Entry Systems

An example of a pen entry system is that developed by Littin (1995). Littin's system allows the entry of formulae with a mouse or pen and tablet. Characters are recognised as they are drawn, using a feature-based character recogniser. Because of a low processing requirement for his parsing technique, the formula is parsed as the user writes. The output of the system is a linear format, such as a LISP-like prefix notation.

As explained later in Section 2.3.1, Littin's use of a modified SLR(1) parser puts a number of restrictions on the system. Users are limited in the order in which symbols for a particular formula can be entered, though the order is fairly reasonable. Editing

is also limited to the modification or deletion of the most recently entered symbol. As a result, arbitrary editing of formulae with his system is impossible.

After a character written by a user has been recognised, it is morphed to a predefined ideal stroke shape. Because the formula is parsed as the user enters it, the morphing also rearranges the positions of the characters to the correct arrangement according to the parser's current understanding of them. The rearrangement is done so that enough room is left around the most recently entered symbols so that the user can still write around them.

Littin claims that moving characters to their correct positions as the person writes encourages them to write in straight lines. He also notes that the morphing of characters to what they have been recognised as and rearranging the formula in real time provides valuable feedback to the user on how the computer's interpretation of their formula is going.

When compared to other methods of formula entry, pen based systems have the advantage that, assuming they are well designed, they are more natural and intuitive to use. A system that allows the user to enter their formulae as they would by writing on a piece of paper has the advantage that it offers a style of interaction that is as similar as possible to writing with a pencil and paper, yet it also has the power of a computer available to perform computations on and manipulations of the formulae entered.

The disadvantage of pen based systems is that allowing freehand input of formulae means that the system must be able to deal with the sloppiness inherent in handwritten input, and ideally the arbitrary order in which users enter formulae. Littin's system avoids this by limiting the order in which users can enter symbols for their formulae, and restricts the editing of formulae to the most recently entered symbol. Ideally a system would allow a completely arbitrary symbol entry order.

Pen and tablets are also not common hardware for home users and writing neatly and quickly with a mouse is very difficult.

2.2 Issues In Formula Recognition

The goal of processing a mathematical formula is to take a list of symbols and their locations, then return a description of the formulae that they represent. Recognising the symbols themselves is an issue as well.

This section discusses a number of general issues that arise during the processing

of mathematical formulae.

2.2.1 Input

When designing a parser, a supply of input is required. A lot of systems that process typeset input use input generated by LaTeX, described in Section 1.1, for several reasons. It is a convenient way to generate input. The process can be easily automated so that an input string can be passed into a system and, after the formula processing, the result can easily be compared to check that it was the same as the input string. The existence of an input string guarantees that there is a "solution" to the parsing process. If something was known to be generated by LaTeX, then it should be possible to regenerate the LaTeX for it. Generating input from screenshots gives clean input data, free from noise and other artifacts that would be introduced as part of a printing and scanning process.

For handwritten input, data is gathered as the user writes with a pen and tablet. In contrast to typeset input, it is quite possible that there will be input that, although is quite reasonable for the person who wrote it, can not have LaTeX generated for it, no matter how good the underlying formula processor is. This may be as a result of LaTeX not being powerful enough to represent the user's input or, more likely, the formula processor not being programmed to anticipate a particular user's style for laying out formulae.

Each individual author will have a fairly consistent style that they use, allowing for variations in the positions and sizes of symbols. Mathematicians also invent their own notations to improve the brevity and readability of their formulae. To accommodate this, an online handwriting based entry tool would ideally be easily extensible by the user, possibly through some sort of GUI tool.

To simplify the problem of recognising mathematical formulae, the consistency of input can be improved by restricting oneself to a certain style of mathematical notation. For example, this can be achieved for typeset input by taking all input from a single source, such as LaTeX, or an individual publication. Within this single source, the style (i.e.: fonts, sizes, spacings, etc.) will be relatively consistent.

Both typeset and handwriting based systems have to recognise the characters that are input to the system. There are issues of segmentation and recognition, then dealing with errors arising from these steps.

If we are using handwritten input, the fact that the user may give sloppy input, erroneous input, or an incomplete formula must be faced. While books can have

mistakes as well, the likelihood of a user giving erroneous input is much higher.

2.2.2 Noise Versus Small Symbols

The system that is processing the input data has to be able to distinguish between noise, dots, commas and symbol annotations, for example: A' and $\dot{x}$. If the information is noise, then it must be removed from consideration. However, removing something which is an annotation must be avoided. This is a more significant problem in the processing of scanned input, as noise is likely to arise from the scanning. Online input with a pen and tablet is not as susceptible, unless you want to account for a user accidentally tapping on the tablet with the pen and drawing odd dots and lines. This does not occur very often, and it's easy enough for the user to notice that they have done so and let them either undo or delete the mistake themselves.

2.2.3 Symbol Segmentation and Recognition

Whether both typeset or handwritten, the input is processed to form a set of symbols, their positions, and sizes. For typeset input, working from scanned images of pages, there is a large variety of fonts, sizes and styles. Within a single publication this will be restricted to a smaller subset. Raw input pixels have to be segmented into individual symbols and then recognised. The analogous problem for online handwritten input is determining which strokes belong to which characters, then recognising them.

Some symbols do not have a constant aspect ratio, for example: brackets, Σ, and $\int$. Their size depends on the symbols that they are associated with. The segmentation process must also be able to find symbols that are inside others. This allows for the recognition of the square-root operator, "$\sqrt{}$" and any other symbols inside it.

Some systems (Miller and Viola, 1998; Chou, 1989) enable feedback to the character recogniser from later stages of formula processing, so that the identity of symbols can be determined based on surrounding symbols. This helps with cases like that illustrated in Figure 2.2. The symbol shown in Figure 2.2(a) could be either a o (the letter o) or a 0 (the digit zero). It is not until it is viewed in the context of the surrounding symbols that its identity can be determined. In Figure 2.2(b) it is a 0, in Figure 2.2(c) it is an o.

For handwritten input there is also a large variety of writing styles. It would be ideal to have a recogniser sufficiently flexible so that it would be possible to train it to work well with a particular user, but also have sufficient generality so that multiple

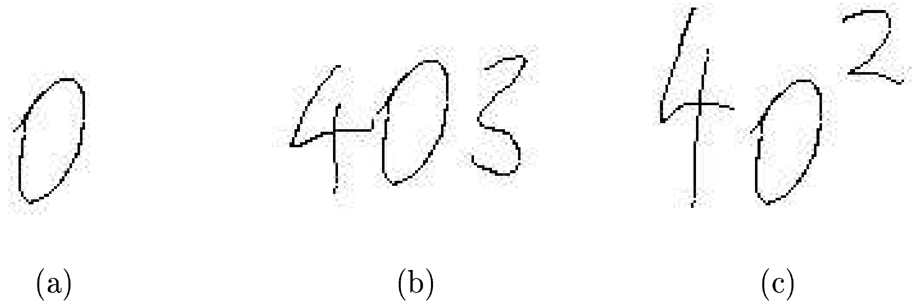


Figure 2.2: Without the context of the surrounding symbols, the identity of a symbol sometimes can not be determined. In (a), it is not possible to determine whether it is an o (the letter o) or 0 (the digit zero). In (b) it is a 0, yet in (c) it is an o.

users can use it without additional training.

2.2.4 Ambiguous Symbols

Some symbols have many possible meanings and a distinction can only be made by examining it in the context of surrounding symbols. Examples of unambiguous symbols, and several ambiguous ones are listed here.

- "!" is always a postfix factorial operator. Its argument can always be found to its immediate left.

- "$\int$" is always an integration operator, but we do not know how many arguments it has. There will be an integrand and a differential (the "dx" part), but either zero, one or two limits.

- "$-$" (a horizontal line) can be an infix subtraction operator, a prefix negation operator, or a fraction bar. It is also possible that it is part of some other symbol, such as $=$, $\leq$, or $\subseteq$.

- "." (a dot) can be multiplication, a decimal point, part of a symbol, or an annotation, e.g.: $3x.y$, 2.71828, !, or $\dot{x}$.

- a_{ij} can mean either the array element $a(i,j)$: the ith row in the jth column in a 2D array, or $a(i \times j)$: the element which is the product of i and j in a 1D array.

- The a in "X^a" can either be a power or, as some authors use it, an index into an array.

Some of the ambiguities listed above are concerned with understanding the underlying meaning of things: their semantics. Others are to do with syntax. For example, the last case is to do with semantics. Determining the meaning of X^a is impossible without the knowledge of what the author intended it to mean. The second example above, determining the number of limits on an integral, is a syntactical problem. If a limit is found, its function is unambiguous. The problem is that the number of limits to look for is indeterminable in advance.

A large amount of reliance is placed on the knowledge and experience of the person reading mathematical formulae. They are expected to understand the context in which something is written, and thus interpret things correctly. To build such experience into an automated system can be difficult.

If the purpose of parsing the formula is to produce LaTeX that generates output that looks like the user's input, determining the underlying meaning is not as important; we are only interested in appearance, not meaning. If we are generating input for mathematical computation packages, such as Mathematica or Matlab, to do calculations with or operations on the formulae, then it is important to know the underlying meaning of conventions that the formula's author uses, so that a correct command string can be produced.

Anderson (1971) and Bernstein (1971) believe that syntax and semantics of a formula are different, and say that the parsing stage should only return something which describes the *layout* of the formula. Bernstein's view is that if we are intending to pass the formula onto some later stage that has its own input format, then the problem of going from the layout description to this input format should be done as a subsequent stage of processing. This could be done, for example, with a 1D string parser.

An example of a formula represented by a layout description follows. This is taken from the paper by Fateman, Tokuyasu, Berman and Mitchell (1996).

The formula

$$\int \frac{x^q - 1}{x^p - x^{-p}} \frac{dx}{x} = \frac{\pi}{2p} \tan \frac{q\pi}{2p}$$

is represented in a positional notation as:

```
(hbox
  (vbox integral nil nil)
  (vbox quotient
        (hbox (expbox x q) - 1)
        (hbox (expbox x p) -
```

```
        (expbox x (box - p))))
(vbox quotient
        (hbox d x)
        x)
=
(vbox quotient
        pi
        (hbox 2 p))
Tang
(vbox quotient
        (hbox q pi)
        (hbox 2 p)))
```

The `hbox` and `vbox` are operators that perform horizontal and vertical concatenation of symbols and subexpressions, in a similar manner to the concatenation operators that Martin uses (Martin, 1967), described in Section 2.3.2. For example, a fraction is a vertical concatenation of the numerator, a horizontal line and the denominator.

Splitting the formula processor into two parts with the first stage being a layout processor returning a description of the layout of the formulae, and the second stage being a formula processor that takes the layout description and returns the command-string for the formula, has the advantages that:

- it breaks the system into two distinct, independent, simpler stages.
- the layout processor does not have to take into account the meaning of the formula, as it is not the final stage in the process. As a result it decouples the layout processor from the formula processor, simplifying its code. All author-dependent customisation can be done at the level of the formula processor, independent of the layout processor.
- either the layout processor or formula processing unit can then be easily taken out and replaced with minimal effort. Each unit in itself is relatively simple with respect to a combined function unit, and has very well defined inputs and outputs.

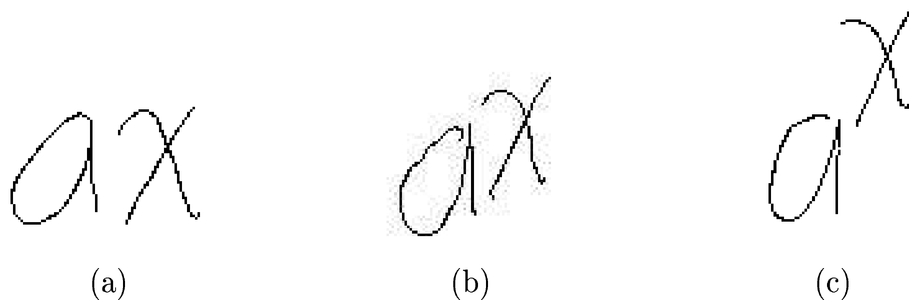
It can also be argued that a single combined function unit can provide the same thing, if it uses a carefully chosen final language. For example, LISP-like notation essentially describes the layout of a formula. Splitting the process into two parts

means you have to write a parser that will take the positional description and output a more human-readable version, such as LaTeX or a Mathematica command string. It also makes it harder for the layout processor to use contextual information in making choices in ambiguous situations, as the layout processor is now a separate part.

2.2.5 Identifying Significant Spatial Relationships

A lot of information is conveyed by the relative location of symbols in a formula. Operations such as exponentiation and implicit multiplication use the relative positions of symbols to indicate the operation intended. This is in contrast to operations that use an explicit symbol to indicate the operation, such as addition where a "+" will always appear between its operands.

Integration uses a symbol to indicate the operation, but uses the relative positioning of symbols to indicate the parameters for the integration. The limits appear near the top and bottom of the integral symbol, and the integrand between the symbol and the differential.



(a) (b) (c)

Figure 2.3: The relative location of symbols can be ambiguous. (a) represents a times x, and (c) represents a to the power of x, but what about (b)?

Because so much information is conveyed by the relative positions of symbols it is important, when interpreting a formula, to correctly identify the intended relative positioning between symbols. In many cases there is a grey area between alternatives. Figure 2.3 illustrates this. In the context of a mathematical formula, it is reasonably "obvious" that Figure 2.3(a) is "a times x" and that Figure 2.3(c) is "a to the power of x", but what about Figure 2.3(b)?

This is primarily a problem for the processing of handwritten formulae, as there must be a degree of leniency in the allowable positions that a user can write symbols, but it does also apply to the processing of typeset formulae.

Geometric relationships between symbols, or groups of symbols, can be determined by using:

- global thresholds.
- local thresholds, based on the symbol involved.
- statistical labelling, which determines the probabilities of various possible arrangements of the symbols.
- constraints based on symbol identity, for example: a^2 is legal, while $a^@$ is not.

Template based equation editors, where the user selects a template for the operator they want and then fills in the boxes in the template, avoid this problem of having to determine the correct geometric relationship between symbols. By having the user select operators from menus and then fill in the appropriately positioned boxes for the operator's arguments, the user is explicitly specifying the structure of the formula, even to the extent of what the arguments for each operator are.

2.2.6 Ambiguity of Symbol Placement

Taken in small local contexts, it is not possible to determine the correct relationship between symbols. For example, seeing "x_i",the i could either be a subscript of x, as in $x_i y_j$, or a coincidental alignment as in $a^x i$. More examples of ambiguities, even ones that would confuse a human, are in a paper by Martin (1971).

2.2.7 Little Redundancy

There is very little redundancy in mathematical notation. Because of this, there are very few cross-checks that can be made to confirm that an interpretation of a formula is correct. Some operators come in pairs, e.g. left and right brackets, or an integral sign and differential, but the majority do not.

2.2.8 Connected and Overlapping Symbols

When symbols are connected or overlap, there is the problem of separating them. This is more of a problem with scanned input, as the input is simply an image. For handwritten input, there is information available on the timing and order of the strokes drawn.

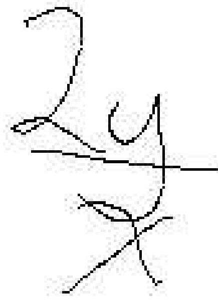


Figure 2.4: Overlapping symbols.

Figure 2.4 shows another problem, where the tail of the y overlaps the fraction bar and the x. This makes it hard to determine the geometric relationship between these symbols reliably. Anderson (1968) initially represents the position of each symbol with a rectangular bounding box, the smallest rectangle that contains all of the symbol's original pixels. He then shrinks these bounding boxes to a single centre point to avoid problems when considering the relationships between symbols. The position of the centre point is dependent on the identity of the character. The way Anderson defines the centre points of bounding boxes is covered in more detail in the discussion of his equation parser in Section 2.3.2.

2.2.9 Ambiguity in the Formula

Zhao, Sakurai, Sugiura and Torii (1996) discuss "tacit agreements", which fall into two classes: determinable and indeterminable. The indeterminable agreements include the examples of ambiguity that Martin talks about in his paper (Martin, 1971). One of Martin's examples is:

$$\text{Does} \quad \sum_{i=5}^{10} i + Y \quad \text{mean} \quad \sum_{i=5}^{10} (i + Y) \quad \text{or} \quad \left(\sum_{i=5}^{10} i\right) + Y?$$

Indeterminable agreements require the knowledge and experience of the reader to resolve. Determinable agreements correspond to rules in the interpretation of formulae, such as the implicit precedence of operations. For example, readers of the formula $a + b \times c$, knowing the standard precedence of matematical operators, understand that the multiplication operation has precedence over the addition. This means that they interpret is as $a + (b \times c)$ and not $(a + b) \times c$.

Zhao et al. discuss different types of grammars, the complexity of each depending on the level of formality of the formula entry system. More formality means that the user spends more of their time specifying "boxes", not dissimilar to the boxes in template-based equation editors, that encode the geometric and logical relationships between various elements in their formulae. The higher the formality, the fewer tacit agreements that have to be encoded into the grammar.

Zhao et al. describe three levels of formalisation: strong, weak, and free. As you move from one to the next the complexity of the grammar increases, as it has to be able to determine more of the tacit agreements in the formulae. Unfortunately, although the strong formalisation is the easiest to write a grammar for, and is the one that offers the most confidence that after processing that you have got the right thing, it also involves the most additional work for the user who is entering formulae. Free formalisation is the exact opposite.

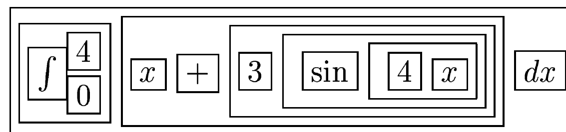
Strong Formalisation

Every structure in a system using strong formalisation has to be put in a box by the user, all determinable and indeterminable tacit agreements are indicated by the user during input. No information is needed in the grammar on the priority of operators, as everything is in a hierarchy of boxes that the user has supplied.

For example, the formula:

$$\int_0^4 x + 3\sin 4x \; dx$$

is encoded as:



It can be seen that the user has had to explicitly encode the precedence of all the operators, along with specifying the two dimensional layout of the formula, and the arguments for each of the operators.

Weak Formalisation

The user has to supply fewer boxes in a system using weak formalisation; the grammar now encodes the precedence of operators. The boxes that no longer have to be drawn are those that originally indicated the precedence of operations.

The resulting grammar is called a "weak grammar" which uses grammatical categories such as sentence, relation, calculation, term, factor and atom. The priority of operators is encoded using these categories.

For example, to encode the implicit operator precedence of multiplication over function application, for expressions such as "$\sin xy$", a grammar can be designed with rules like:

<sin op> ← sin <term> and

<term> ← <term> <factor>

The example formula above is now entered as:

$$\fbox{$\fbox{$\fbox{$\int$}\begin{array}{c}\fbox{4}\\\fbox{0}\end{array}$}\ \fbox{$\fbox{x}\ \fbox{$+$}\ \fbox{3}\ \fbox{sin}\ \fbox{$\fbox{4}\ \fbox{$x$}$}$}\ \fbox{$dx$}$}$$

This level of boxing of entries is similar to that used by template based equation editors.

Free Formalisation

The only boxes required are those that specify the layout of formulae.

$$\fbox{$\fbox{$\fbox{$\int$}\begin{array}{c}\fbox{4}\\\fbox{0}\end{array}$}\ \fbox{$x$}\ \fbox{$+$}\ \fbox{3}\ \fbox{sin}\ \fbox{4}\ \fbox{x}\ \fbox{dx}$}$$

When using a free formalism, the grammar for parsing the formula has to be extended to determine the start and end of groups of symbols, such as operands for operators. In the example above, the grammar has to determine where the integrand starts and finishes.

No Formalisation

From the user's point of view, an ideal system would be a step beyond free formalisation, where the system would determine where the boxes are. This would let the user concentrate on the meaning of the formula and not worry about having to explicitly define its layout. The only problem with this is that the user then has to trust the system to interpret their formula correctly. The system has to be sufficiently powerful to do so, without overly restricting the positions that symbols can be placed with respect to one another.

2.2.10 Post-processing Error Correction Rules

These rely on the programmer anticipating what sort of errors may occur and providing means for dealing with them in advance. For example:

- “5in” $\rightarrow$ “sin”,
- “c0s” $\rightarrow$ “cos”,
- “$/ < i < n$” $\rightarrow$ “$1 < i < n$”,
- “5_2” $\rightarrow$ “S_2”

The limitation of this approach is that it is not possible to automatically generate these rules. However, a stochastic grammar is able to automatically choose likely alternatives when given an unparsable formula. Miller and Viola (1998) give an example, where their system attempted to parse “0^{0_0}”. The system realised that “0_0” was not a legal construct and concluded that it was most likely to be “00”. As the output “0^{00}” would be illegal, due to a two digit number not being allowed to begin with a “0”, it considered all the digits to be approximately the same size, and use an alternative recognition of the first digit, giving “600” as the best interpretation. While this was not the “correct” interpretation of the formula given that the input was “0^{0_0}”, it was the most likely interpretation within the constraints of the grammar the system was using.

2.3 Formula Parsers

This section discusses types of existing formula parsing systems, past and present. These systems are possibly components of some larger existing system. Their ability to process input ranging from neat typeset print to sloppy handwritten entry is discussed.

The goal of a formula parser is to start with a set of recognised symbols, and return a description of the formula that they represent. Blostein and Grbavec (1996) give a good overview of the categories of existing techniques for parsing mathematical formulae.

2.3.1 Modified Grammars

One technique is to take an existing one dimensional grammar that parses 1D strings, and modify it so that it incorporates checks of the geometric relationships between

symbols. This technique is only applicable to online input as time is used to order the symbols before parsing. This is the approach used by Littin (1993).

Littin uses a SLR(1) parsing technique, with additional tests for checking the geometric relationship between tokens. He does this by first building a SLR(1) parser then extending it where necessary to include these geometric tests.

To enable the use of this modified SLR(1) parser, the input has to be ordered correctly: the user has to enter the symbols for a formulae in a predefined order. For example,

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

has to be entered in the order a, $+$, b, — , c, $-$, d. For some formulae there are a number of possible orders, but this only occurs when a symbol has both a subscript and superscript. The user is then able to decide whether to write the subscript or superscript first. Editing of the formula during and after entry is limited to the deletion or alteration of the most recently written symbol.

Permitting the user to enter symbols in an arbitrary order would mean that this type of grammar could not be used. He justifies this restriction through the fact that people tend to enter formulae in a fairly standard order, which he informally verified by observing several people writing a number of formulae. Although this assumption is reasonable, if users want to be able to go back later and edit their formulae, they are unable to. Littin has created a formula *entry* system, rather than a formula *editing* system.

The modification to the SLR(1) grammar adds geometric tests which check that the symbols are in the correct locations. Tolerance for the sloppiness of handwritten input is implemented by putting a threshold on the distance symbols can appear from their expected locations.

One major advantage of SLR(1) parsing is its time and space efficiency. Littin shows the computational complexity of his system to be $O(g)$, where g is the number of symbols in the input formula.

2.3.2 Box Languages

A box language divides the input plane into areas based on the symbols found. For example, the rule for a summation includes a term that subdivides the input area, as in Figure 2.5.

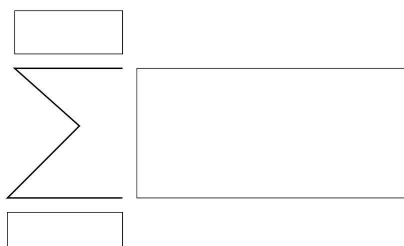


Figure 2.5: A box language summation template.

When the parser finds a "Σ", it looks in the appropriate areas above and below the "Σ" for the limits, then to the right for the expression being summed.

The way that box languages divide up the input area can be implemented in two ways. It can use explicit definitions in the grammar rules that specify where the boxes are, with respect to the symbol being considered. This approach is used by Anderson (1968). An alternative approach is to use "concatenation operators" which offer geometric operations such as "vertical concatenation". Using concatenation operators, a fraction can recognised as a vertical concatenation of the numerator, a horizontal line and the denominator. This approach is used by Martin (1967).

When the concatenation operators are applied, they define the subdivision of the input plane, restricting the positions in which to look for symbols. The way the concatenation operators actually subdivide the input area is defined externally, not as part of the grammar.

The use of box grammars is a common approach, from the early formula parsers (Anderson, 1968; Martin, 1967) through to systems currently being developed (Fateman, Tokuyasu, Berman and Mitchell, 1996; Zhao, Sakurai, Sugiura and Torii, 1996).

Martin's system from 1967 (Martin, 1967) is one of the earliest systems for the parsing of mathematical formulae, processing handwritten formulae entered using a pen and tablet.

Martin gives details of the method he uses for analysing the formula on the 2D input plane, and how to decide where to look next for each part of the equation, based on these positional or "concatenation" operators. The rule for addition in Martin's grammar is `(C= T* + E*)`. This defines the horizontal concatenation (`C=`) of a Term (`T*`) a plus symbol (`+`) and an Expression (`E*`).

Martin's system uses several specific hard-coded tests and rules in addition to the grammar, to ensure that his system works correctly. He constantly looks out two characters ahead for exponents on derivatives, e.g.: the i's in $\frac{d^i x}{dy^i}$. Each symbol input

to the system has a bounding box that defines the area it covers. As the system works in a single left-to-right pass on the input, Martin needs to ensure that the correct symbols are encountered first. He extends the bounding boxes of Σ's (for summation) and fraction bars (——) one character to the left. Doing so means that the symbol indicating the operation is found first, so when the symbols around it are found, the parser is able to associate them correctly with the operator.

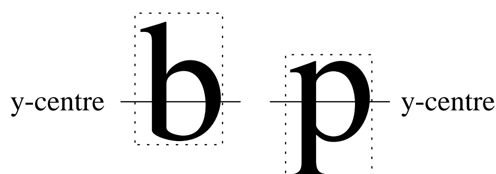


Figure 2.6: These y-centres of these symbols line up, although their bounding boxes do not.

Another well known early equation parser is Anderson's system (Anderson, 1968). He assumes that the OCR problem has already been solved, so each character or "syntactic unit" has known physical bounds and an x- and y- centre, not necessarily the average of the left and right or upper and lower bounds. Working with these x- and y- centres avoids the problem of the bounding boxes of ascenders and descenders not lining up. Figure 2.6 demonstrates this. The bounding boxes are not lined up, however the y-centres of the symbols do.

To save on processing time, Anderson uses a preprocessing step to do an initial lexical analysis of the symbols input. Individual symbols are preassigned with their syntactic category, instead of using rules in the main grammar such as:

$$a|b|c|..|z \implies < letter >$$

Instead of using concatenation operators, Anderson puts tests in as part of rules in the grammar, limiting the positions where symbols can be legally positioned. As a result, the 2D rules in his grammar have several parts to them, that:

- check to see if the right syntactic units are present.
- check the layout of these syntactic units, using geometric constraints.
- have a rule to determine the physical characteristics of the result of applying the rule. This calculates things such as the size, position, and x- and y- centres of the new bounding box.

- build up a parse tree or a LISP-like expression that represents the formula.

The paper by Martin (1971) discusses various aspects of input, parsing, and display of mathematical formulae. It provides useful results of a study they did on the syntax and layout of mathematical formulae. He determined that there is no one “official” layout, but he does come up with a number of general observations about the layout of formulae that hold true in most cases.

He discusses a method for parsing formulae, which is the same as that described by Martin in his earlier paper (Martin, 1967), and how to avoid problems with it. He also discusses the splitting of formulae across several lines when the formula is too long, display of formulae, ambiguity in input and how to judge if it is a valid expression.

More recent papers by Fateman et al. (1996) and Fateman and Tokuyasu (1996) discuss their recent work on the automatic interpretation of typeset formulae that have been scanned in from books of tables of integrals. Once interpreted, they intend to store the formulae so that they can be retrieved from an online integral lookup table, for use in computer algebra systems.

In order to simplify the problem, they assume that the layout of the typeset notations being input to the system is constant. This is a valid assumption, as all their input is being taken from a single book. They also assume that the output of the OCR system that feeds the recogniser is 100% correct, so the misrecognition errors are of no concern. Test input is currently cleaned up by hand if there are characters that have been mistakenly joined, or if there is excessive noise that is confusing the OCR stage.

Their formula parsing technique is similar to the box language used by Anderson (1968). The input to their system is taken from the output of a sub-system that automatically processes raw bitmaps of scanned pages or screen-captures of previewed LaTeX formulae. The sub-system automatically locates and recognises text in these images. This is then passed on to the formula parsing stages.

Initially, before the main parsing stage, they do a lexical analysis to collect up multi-character components. For example, “cos” would originally be passed as three separate characters, i.e.: “c”, “o” and “s”. These are put together to make a single “cos” element. Other multi-element symbols are gathered up as well, such as “=” and “i”, should this not have happened already in the preceding OCR stage. The main parsing stage then takes this and turns it into a description of the symbols’ layout using positional operators. This is then parsed to generate a representation of the mathematical formula.

2.3.3 Projection Profile Cutting

Projection profile cutting, also known as "structural analysis", determines the structure of a formula from a number of repeated horizontal and vertical projections of a formula's image. Based on these projections the formula is subdivided, each subdivision being recursively projected and further subdivided. A tree structure is created, representing the formula's geometric structure. This structure is then further processed, taking into account the symbols in the formula, and the formula is determined.

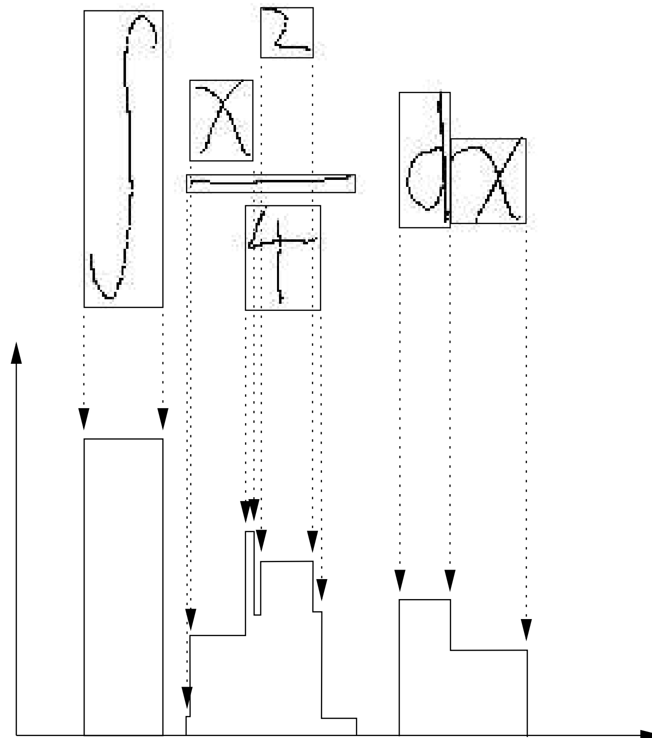


Figure 2.7: Building a projection profile.

Figure 2.7 shows how a projection profile is created. The height of the histogram is determined by the area of the bounding boxes (Ha, Haralick and Phillips, 1995). The histogram can also be created using the density of pixels at each x- or y-position. For example, if this formula was subdivided based on the minima in the histogram, it would be split into three parts: the integral symbol, the fraction, and the differential. The fraction could then be horizontally projected which would identify the numerator and denominator.

Blostein and Grbavec (1996) say the disadvantage of projection profile cutting is that special processing is required for square-roots, sub- and super-scripts. They report

that projection profile cutting has been used on both typeset and handwritten input, although with handwritten input it has trouble with square-roots, closely-written symbols and skew.

Projection profile cutting is also used to process scanned text documents, subdividing the text into columns, paragraphs, and lines (Srihari, 1986; Ha et al., 1995). It is also used for analysing scanned images of sheet music, to separate the staves and musical symbols.

2.3.4 Procedurally Coded Math Syntax

Procedurally coded math syntax uses a collection of rule-of-thumb observations about formulae. These observations are coded into a formula processing program. An example of a rule, quoted in Blostein's paper (Blostein and Grbavec, 1996), "A length threshold of 20 pixels is used to classify a horizontal line as a long bar or a short bar. If a long bar has symbols both above and below it, it is treated as a division. If there are no symbols above it, it is treated as boolean negation. If a short bar has no symbols above or below it, it is treated as a minus sign. If it has characters above or below it, then combination characters (e.g.: $=$, $\geq$, $\leq$) are formed."

A collection of rules such as these are used to parse formulae. The use of thresholds is sufficient for the processing of typeset input taken from a uniform source, but the high variability of handwritten input could make it fail.

The rules that are hard coded into the system perform essentially the same function as the rules in a box language, described in Section 2.3.2. The only difference is that in procedurally coded syntax, they are built into the system as part of the code for the formula processor. In a box language, the rules are provided through a modifiable external data file.

Using procedurally coded math syntax means that it may be easier to write more complex or "intelligent" rules that use extra processing that could not be encoded as part of box language rules. However, the major disadvantages arise from the fact that rules are coded into the system itself, so that changing them involves rewriting parts of the program. It may be impossible for the end user to make modifications or extensions. The ability to modify a handwriting based formula parser is important due to the variability of notations, and the need to allow advanced users to create new notations.

Rules are typically added to the system as necessary throughout its development, correcting errors as they occur. As a result, the set of rules for a system progressively

grow, with each new rule addressing the current problem. Systems end up with a large number of rules, with specialised sections of code for dealing with particular situations and problems.

2.3.5 Stochastic Grammars

Stochastic grammars are reported to yield good results with both typeset and handwritten input. A stochastic approach can be added to any type of grammar, and two examples of systems that use this type of approach are Chou's (1989), and Miller and Viola's (1998). Chou's work processes typeset input, while Miller and Viola are now moving from typeset to handwritten input.

A stochastic grammar has associated with every production a probability that the production is used. Thus, for any given sequence of productions in a given parse, the overall probability of this sequence can be calculated. The correct parsing of a set of symbols is the parsing that has the highest probability.

To use this approach, each production in the grammar needs to be assigned a probability. There are a number of algorithms for assigning probabilities, typically working from a set of example strings which are known to be in the language that the grammar describes. Chou's paper (Chou, 1989) describes how to adapt the "inside/outside algorithm", originally designed for linear one dimensional input, to a two dimensional grammar that uses vertical and horizontal concatenation operators. The inside/outside algorithm makes a number of passes over the grammar and examples from that grammar's language, determining the probabilities for each rule in the grammar.

Stochastic grammars cope well, and in a pleasing way, with geometric tests as symbols can be given a probability that it has a particular geometric relation to other symbols or subexpressions. This is in contrast to other approaches which judge arrangements to be either valid or invalid. Miller and Viola (1998) model the positions of symbols as Gaussian variables, the probability that two elements are in a particular relationship relative to each other is defined by a two-dimensional Gaussian distribution around the expected position of the second expression. This helps cope with ambiguities, such as those described in Section 2.2.5.

Another advantage of a stochastic approach is that it can take as its input the output of the character recogniser in the form of symbols and possible alternatives, along with confidence values. As a result, the stochastic parser itself can choose from the alternatives in ambiguous cases, or when an error occurs, to get the most likely parse.

To simplify matters, the Miller and Viola sub-class symbols into sets of equivalent symbols:

- ascender letters (b,d,h,i,k,l,t,A-Z (except Q), δ).
- descender letters (g,p,q,y,γ).
- small letters (a,c,e,m,n,o,r,s,u,v,w,x,z,α).
- ascender/descenders (f,j,Q,β).
- binary operators ($+$,$-$,$=$).
- zero (0).
- non-zero digits (1-9).
- other symbols, each in a class of their own (round, curly and square brackets, fraction symbol).

Chou maintains all possible recognitions of symbols with a probability over a given threshold. Miller and Viola note that all symbols in each subclass are syntactically equivalent, so only keep the best from each subclass.

Miller and Viola use an A^* heuristic to guide their search with the probabilities of each production being calculated during the parsing process. The heuristic provides an estimate of the number of steps between the current state and the solution. For a heuristic to be A^*, it is required to never overestimate the number of steps. Thus, if such a heuristic is used to guide a search, then the shortest path to the solution will be found.

Their paper reports the success their system has, and how much faster their system is with the techniques that they use. They report that the heuristics reduce the parsing times for formulae from minutes to seconds. The paper by Miller and Viola has some very good results: it currently works very well on typeset text and their preliminary results with handwritten input are also very encouraging.

The use of stochastic grammars seems to be one of the two best approaches available for the processing of handwritten input, the other being graph rewriting.

2.3.6 Graph Rewriting

Graph rewriting uses a graph, consisting of nodes and arcs, to represent a formula. "Rewrite rules" are used to progressively reduce the graph, repeatedly replacing sub-graphs with new graphs. This technique is described by Blostein and Grbavec (1996). Lavirotte and Pottier (1995; 1997; 1998) present a system they have implemented that uses this technique, processing scanned LaTeX formulae, and an optimisation they have developed for speed.

To parse a formula using a graph grammar approach, a graph is first defined that represents the recognised symbols and the geometric relationship between them. Nodes in the graph, one for each symbol, have attributes holding information about the identity, lexical class (letter, digit, etc.) and position of symbols in the formulae. Arcs in the graph represent relative positions of the symbols, above, below, up-right, down-right, etc.

Rules in the grammar are also graphs, typically one to five nodes in size, defining sub-graphs that are searched for in the graph representing the formula. These sub-graphs are typically templates for expressions or parts of expressions found in formulae. As the rules in graph grammars are productions, each rule also contains a second graph that is the result of applying the rule: what the first graph is replaced with. Finally, the rule also contains information on how to transfer the attributes of the nodes of the first graph to the replacement graph. There may also be components in the rule that define how to treat the arcs that were connected to the subgraph that has been removed.

These rules are extended by Lavirotte and Pottier to contain graphs that define the contexts in which it is allowable to apply a particular rule. This increases the efficiency of the parsing process, as the contexts define extra conditions that must be met before a rule is applied. The extra conditions remove some ambiguities from the grammar, and thus reduce the possibility of exploring erroneous derivations due to choosing the wrong rule.

As the parsing process works by successively finding sub-graphs and replacing them with smaller graphs, at the end of the parsing process a single node is left representing the original formula. The original formula can then be determined by examining the attributes of this remaining node.

Graph rewriting is a very general and powerful tool and has been used for a wide range of tasks. Graph rewriting has been used in the recognition and interpretation of schematic diagrams (Bunke, 1982), and the description and processing of "structured"

pictures, including flow charts, organic chemistry molecules, and images of particle trajectories produced in physics experiments.

Blostein reports that graph grammars are tolerant to irregular symbol positioning, as found in handwritten input. Work by Lavirotte and Pottier has only been on single typeset formulae and works well. Input to their system is scanned images of printed LaTeX formulae.

Lavirotte and Pottier optimise the graph reduction process by adding context information to the rules in the graph grammar that avoid ambiguities where two or more rules can be applied in a given situation. These rules are created semi-automatically, also using information such as operator precedence information supplied by the person who builds the grammar.

As a graph rewriting system is used in this thesis, it is described in more detail in Chapter 3.

2.3.7 Data Driven and Knowledge Driven Modules

Blostein (1996) outlines systems known as a "blackboard architecture", after the similarity to a number of human experts communicating by writing on a blackboard. This approach takes a number of independent modules that communicate via shared memory. Central to this approach is the shared memory, appropriate data structures to hold and exchange the information, and controlling logic that arbitrates and directs the access to the data.

Each module takes information from the blackboard, processes it and then puts results, possibly with confidence information, back onto the blackboard. This information is then available for other modules to work with.

One of the benefits of a blackboard architecture is that it supports multiple, possibly conflicting, hypotheses and the ability to explore them all. It also allows the easy integration of new "knowledge sources" to the system. It also makes it easy to set up communication between various modules, so that a high level module, for example a formula parser, can provide contextual information to a low-level module, such as a character recogniser.

This approach is also useful to bring together multiple approaches, and to be able to automatically choose between them in a given situation.

Although this approach has a number of advantages, it was not used as part of the system described by this thesis. Although it would have provided a method for exploring many different solution paths and given the ability for different parts of the

system to communicate with each other, the blackboard architecture approach was not used in order to keep the overall complexity of this system down.

2.4 Summary

Existing commercial formula entry systems are either command-string or template based. While these are reasonably powerful and not too hard to use, they do not provide easy entry and editing of formulae. The ideal input method is via a pen-based interface, which means that the formula has to be parsed either as it is entered, or once it is complete.

Each of the existing methods for parsing handwritten or typeset formulae have their own strengths and weaknesses with respect to the various issues of formula processing. Of the approaches presented here, a formula processor using either a stochastic grammar or graph rewriting appears to be the best choice for parsing handwritten formulae.

Chapter 3

The Formula Processor

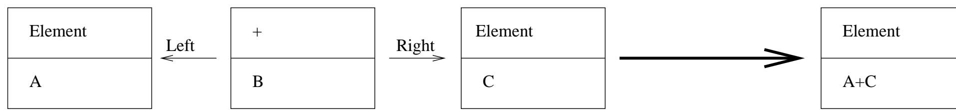
The formula processor implemented is a graph rewriting system, similar to that described by Lavirotte and Pottier (1997). Using an input of symbols and their bounding boxes, a graph is constructed with nodes representing the symbols in the formula. Each node has data associated with it, holding the identity of the symbol, the location of its bounding box and a unique ID. As parsing proceeds, nodes will come to represent subexpressions within the formula. Directed arcs are built in the graph, within predetermined restrictions, signifying the geometric relationships between the symbols in the formula. Each arc has a label that stores what geometric relationship it represents between the two nodes it connects.

By making it possible to build more than one arc between any two given nodes, uncertain relationships can be represented. For example, to represent the possibility a symbol may be either to the "top-right" or "right" of another, two arcs are built between their respective nodes.

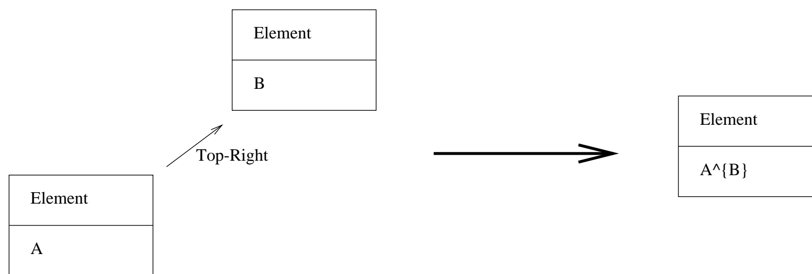
The restrictions on the building of arcs limit the arcs to "sensible" ones, based on their types and positions relative to each other and to other symbols. This helps the later parsing process by simplifying the graph built.

After being constructed from the symbols in the input formula, the graph is parsed by a graph grammar parser. The grammar used by the parser describes the syntax and layout of mathematical formulae through a number of small, typically one to five node, graphs that are templates for subexpressions that are found in mathematical expressions. Figure 3.1 shows some rules from a graph grammar.

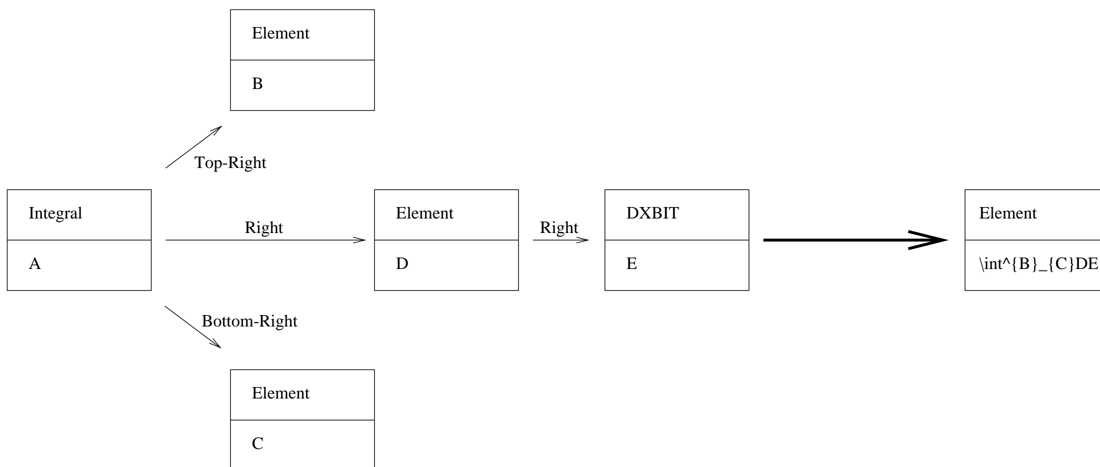
In Figure 3.1, we can see boxes representing nodes in the graph. Each box has two parts. The lower part is the "data" component, and the upper part is the "type" component. When rules are being matched to a graph representing a formula, the



(a) Addition.



(b) Superscript.



(c) Integral with limits.

Figure 3.1: Sample rules from a graph grammar.

"type" values are considered. When the rule is being applied, the "data" components dictate the data component of the resulting node after application. As nodes are combined, a new bounding box is calculated based on the nodes being combined.

The superscript rule in Figure 3.1(b) shows how a superscript operation is defined. Two nodes of type "element", with a "top-right" arc between them are found. If the rule is applied, these nodes are then collapsed to a single node. The type of the new node is "element" and the data value for the new node is constructed from the data value of the original two nodes as shown, the "A" and "B" being the original data values of the original two nodes.

It can be seen in these rules that not all the data values of the original nodes are always used. The discarded data values are typically those of the operator symbol. For example, in the integral rule in Figure 3.1(c), the integral is node "A", however the integration is represented by the LaTeX command `\int` in the final string.

It is possible to build either an abstract representation of the formula, such as a parse tree, or a LaTeX string as is being done here. The parse tree is a much more versatile output format and can be linearised after parsing to generate LaTeX or some other notation. Here, the direct generation of LaTeX is sufficient for this system's purposes. A LISP-like prefix notation is also commonly used (Anderson, 1968; Martin, 1971; Fateman et al., 1996), and is more suited as an intermediate representation if you do not want to use LaTeX or an internal data structure for the parse tree.

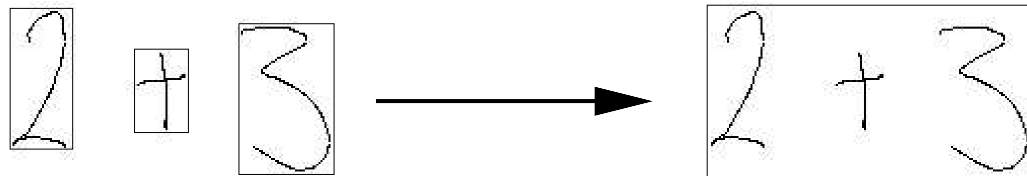
As described in Section 2.2.4, Anderson (1971) and Bernstein (1971) recommend having the formula parser initially produce a layout description of the formula, then having a later stage of processing to determine the actual formula based on this. As the system aims to generate LaTeX that represents the formula that the user has entered, it is valid to bypass the intermediate layout representation and generate LaTeX directly. However, should it prove necessary to generate the positional notation, changing the grammar to produce a LISP-like layout description would be simple, as it is a matter of changing the grammar from producing LaTeX to producing the LISP-like layout description code.

3.1 Formula Processor Details

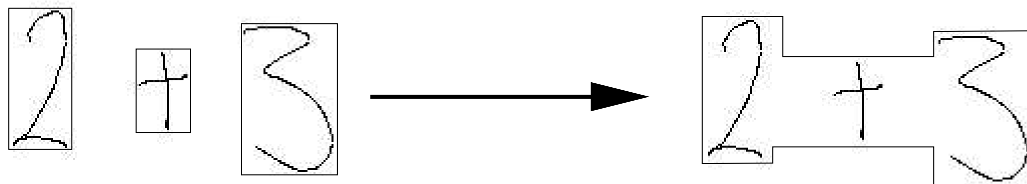
This section gives details of the internal workings of the graph rewriting formula processor. It covers the definition of bounding regions, the input to the formula parser, then how the formula parser builds a graph that represents the formula and parses it.

3.1.1 Bounding Regions

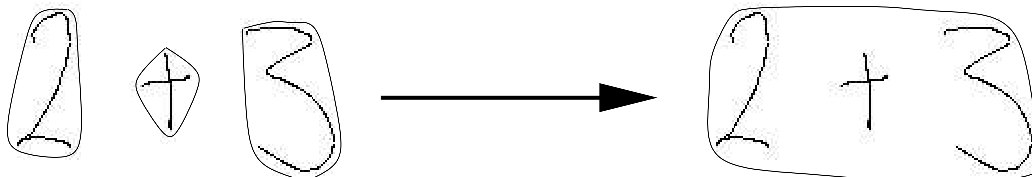
Bounding region information for a symbol, or groups of symbols, is important as it is used in the building of arcs between nodes, as described in Section 3.1.4, and deciding whether or not to apply rules, as described later in Section 3.1.6.



(a) Rectangular Regions.



(b) Individual Rectangular Regions.



(c) Smallest Convex Hull.

Figure 3.2: Construction of bounding regions.

Figure 3.2 shows three different methods for managing bounding regions. Figure 3.2(a) is the approach used by this system. The original bounding boxes are the smallest rectangles that enclose the symbols' strokes. When combined, the new bounding box is made from the outermost extents of the bounding boxes of the original symbols. Other approaches are to keep track of the individual bounding boxes that went into it, as shown in Figure 3.2(b), though this raises the issue of how to deal with the gaps *between* the boxes. Figure 3.2(c) illustrates the use of the smallest convex hull around the pixels or strokes of the original characters, as Miller and Viola do (Miller and Viola, 1998). When the symbols are combined, the new bounding region is the smallest convex hull enclosing all the symbols.

The only disadvantage of using rectangular bounding boxes is that the bounding

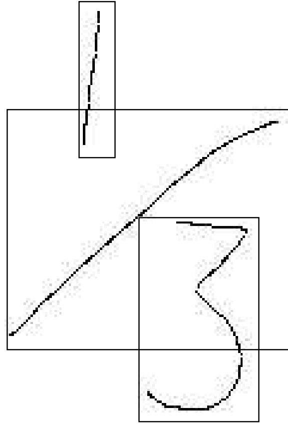


Figure 3.3: These bounding boxes overlap, although the symbols do not.

box of sloped characters is excessively large and takes in a large amount of empty space. As a result, users can accidentally put things overlapping or inside other symbols without intending to. Figure 3.3 demonstrates this. From the user's point of view, the characters do not overlap, but a system which works with the bounding boxes will say they do. This causes problems when, for example, parsing the formula because, the system will interpret the geometric relationship between the symbols incorrectly. As a result, the formula will either be misparsed or be completely unparsable.

Treating the symbol you are testing against another as a central point, and seeing if this point is inside the other's bounding box helps, but the problem can still occur. Figure 3.4 shows that the centre point of the 1 is not inside the fraction bar, but the centre point of the 3 still is. The use of this approach has, in spite of this problem, worked well unless users write excessively sloped fraction bars or integral symbols.

It is in situations like this that Miller and Viola's (1998) convex hull approach for bounding regions is much better. The computational complexity for creating complex hulls from a set of n pixels is $O(n \log n)$. The union of two convex hulls can be computed in $O(l + m)$, where l and m are the number of vertices in the convex hulls. The intersection of two convex hulls can be determined in $O(l + m)$ also. Convex hulls will also at times take in large amounts of empty space for large subexpressions in a formula, as the rectangular regions do, but for single symbols the area they cover is more intuitively what "belongs" to a single symbol. In the case of a sloped fraction

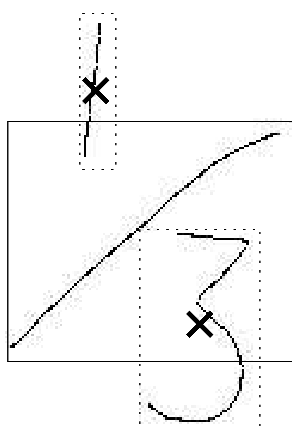


Figure 3.4: Using centre points instead of bounding boxes for geometric tests. While the 1 is outside the fraction bar's bounding box, the 3 is not.

bar, a problem for rectangular bounding regions, the convex hull approach is a great improvement.

This system does not use the convex hull approach due to the fact that it was discovered after the system was already using rectangular regions. The rectangular regions work well enough that changing the system to use convex hulls is not currently necessary.

3.1.2 Input to the Formula Processor

The input to the formula processor is a set of tuples. There is one tuple per symbol that the system has recognised, each tuple holding the symbol's identity and its bounding box information.

For example, the formula shown in Figure 3.5 is encoded as:

```
integral 14 15 37 127
x 63 42 25 39
d 150 54 20 56
x 171 73 31 35
2 92 16 24 21
- 61 85 70 8
4 86 100 28 43
```

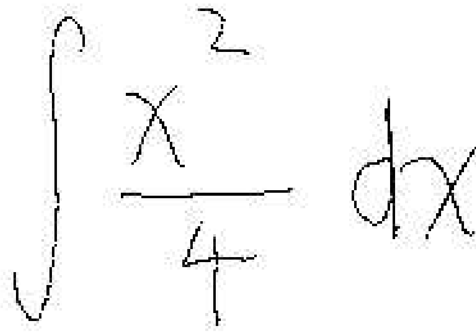


Figure 3.5: A simple formula.

Note that these do not have to be sorted in a "logical" input order, as order information is not used by a graph rewriting parser.

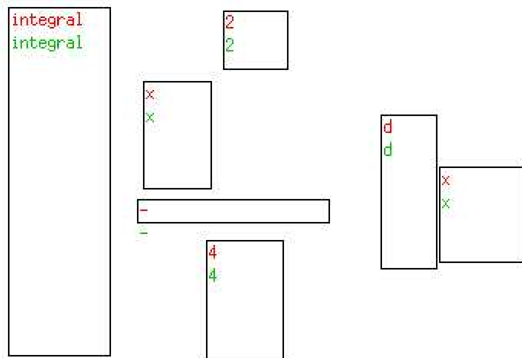
3.1.3 Building the Initial Graph

The tuples are used to build a graph encoding their information. A node is created for each symbol in the formula, and the data and type attributes for each node are both initially set to the symbol's identity. These may be changed in the later preprocessing step which will perform an initial categorisation of the symbols and will group basic collections of symbols such as multi-digit numbers, into a single node.

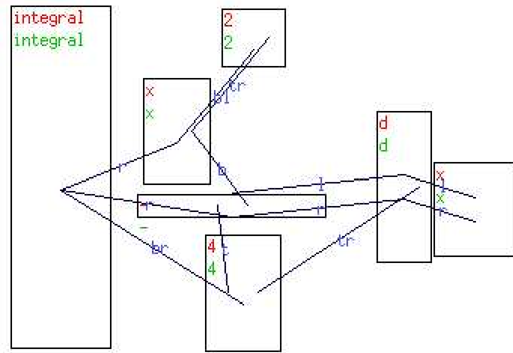
Figure 3.6(a) shows the initial graph built, prior to the preprocessing and initial building of arcs. The nodes are represented by rectangles. The position and size of each rectangle conveys the position and size information encoded within the node. The top label in each node shows the "data" value for the node, what the node actually is, and the lower label is the "type" value, which holds the node's lexical class.

3.1.4 Building the Arcs

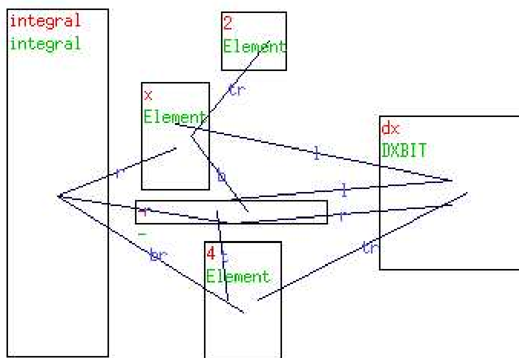
Directed labelled arcs are added to the graph, representing the geometric relationship between the nodes they link. Instead of making a complete graph, with arcs between every pair of nodes, it is restricted so that only "reasonable" arcs are put in. The constraint on having only "reasonable" arcs simplifies the resulting graph. The fewer arcs in a graph, the less likely it is to represent multiple formulae. For example, if a 2 is linked so that a single x node is both to it's "right" and "top-right", it simultaneously represents the formula $2x$ and 2^x. On the other hand, too few arcs and the graph will



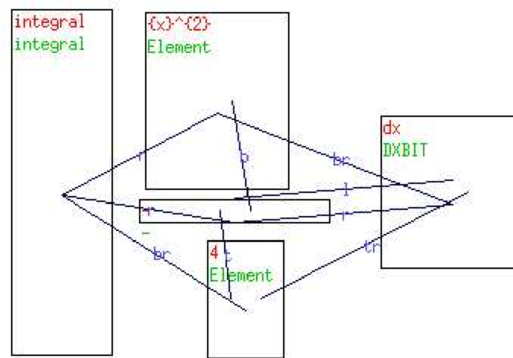
(a) Initial graph.



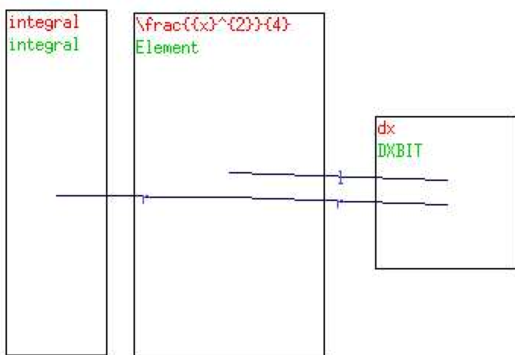
(b) Initial graph with arcs built.



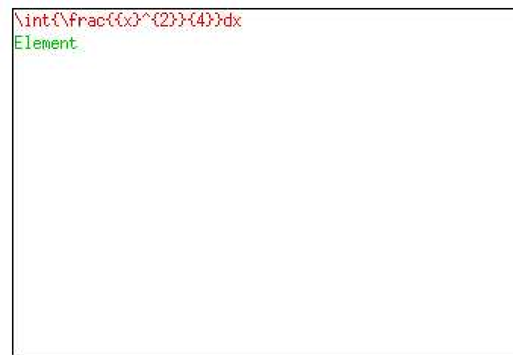
(c) After the preprocessing step.



(d) After applying the "superscript" rule.



(e) After applying the "fraction" rule.



(f) The final graph, after applying the "integral" rule.

Figure 3.6: Preprocessing and parsing of a simple formula.

not represent a valid formula at all, either because it is not connected or an essential arc is missing. Depending on the graph matching algorithm used by the graph-rewriting parser, having fewer edges in the graph may also result in a speed increase.

Every time the graph is changed, in later preprocessing and parsing steps, the arcs are rebuilt. While graph grammars can have rules that specify how arcs are handled as nodes are replaced and added, for a formula processing application the arcs have to be rebuilt every time the graph changes. This is because you need to check if arcs that may not have originally been in the graph have to be added. For example, in the formula:

$$x^{-y}$$

there would be no link between the original x and $-$ nodes, as the arc construction type check, described below, does not allow the expression x^{-}. After collapsing the $-$ and y to make a single $-y$ node, this node has to be linked to the x for it to continue parsing. Without globally recomputing all the arcs in the graph, it is not possible to automatically determine whether or not new links such as this should be added.

As each arc represents a geometric relationship between the two nodes it connects, various checks are made to confirm that the link makes sense. There are three tests, related to the types, the geometric relationship between them, and whether or not there is anything else between them.

Type Check

The grammar designer can specify particular types of nodes that are not permitted to have particular types of arcs going into, or coming out of them.

For example, an integral sign never has anything to its top-left, so the graph builder is told never to build arcs so that a "$\int$" has something to the "top-left" of it. A "+" can never be found to the bottom left of something, so "bottom-left" arcs ending at a "+" are not built.

Currently, these restrictions are read in from an external data file. Ideally these rules should be automatically determined, at least in part, by analysing the grammar.

Geometric Test

A geometric check is made to ensure that the symbols or subexpressions that nodes represent are actually arranged the way that an arc between these nodes implies they are.

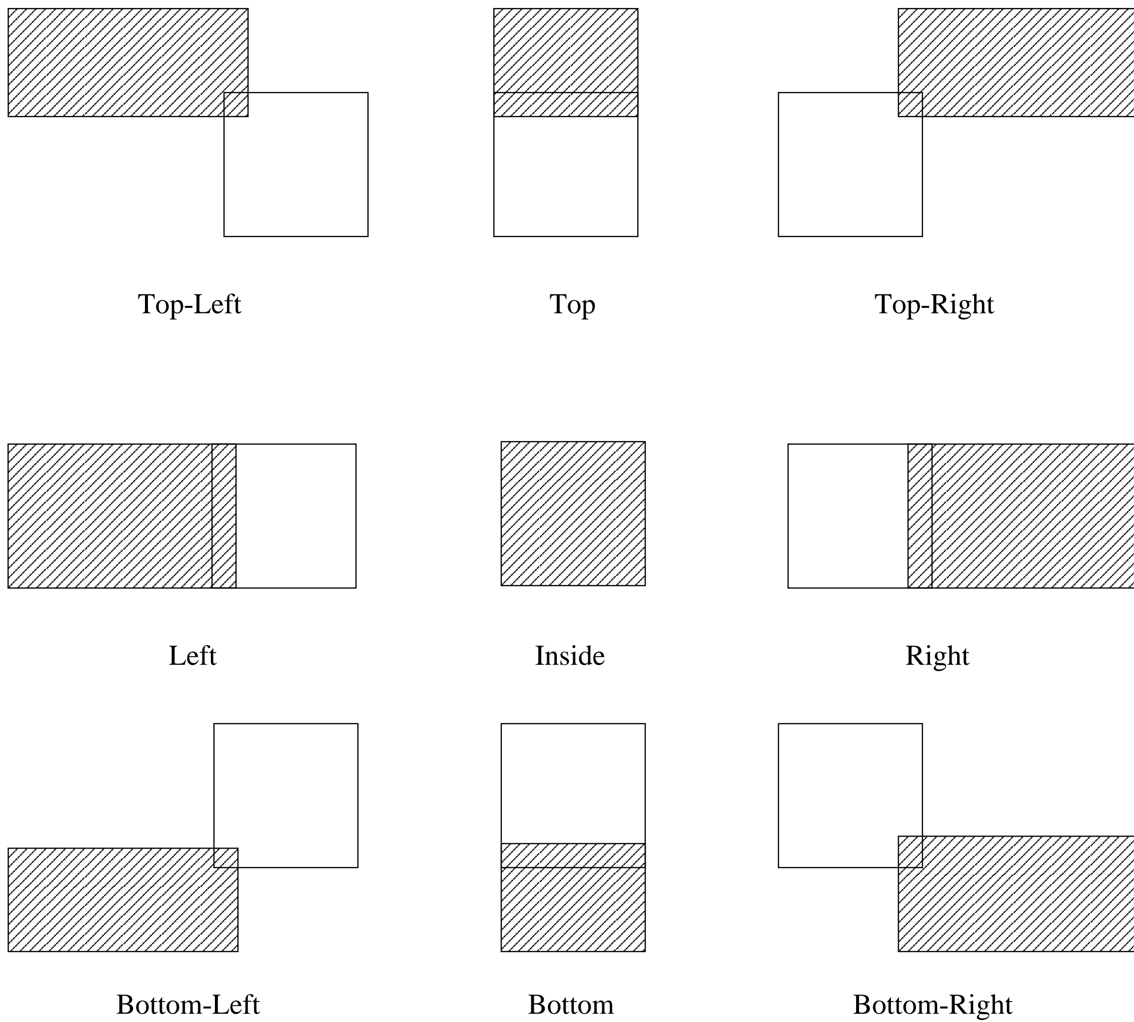


Figure 3.7: Regions used by the geometric check.

Figure 3.7 shows the regions used by the geometric check. There are nine regions, including an "inside" region, all defined relative to the first symbol's bounding box. Note that neighbouring regions, for example the "top-right" and "right" regions, overlap. This helps deal with problems occurring when people accidentally write characters too close to one another and end up with overlapping bounding boxes. It also allows some leniency in the placement of characters.

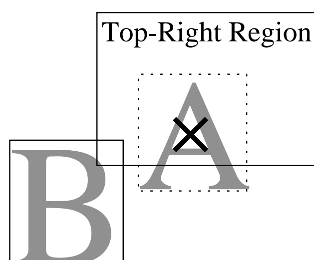


Figure 3.8: Is "A" to the top-right of "B"? Yes.

Before building an arc between two nodes A and B that would encode relationship x, it is tested to see whether or not symbol A is actually geometrically x of B. The centre point of A's bounding box is tested against the regions defined in terms of the dimensions of B's bounding box. Figure 3.8 illustrates the check to see if A is to the top-right of B. This approach of using rectangular regions is also used by Littin (1995).

Symbols have to be sufficiently close to one another to be linked. The impact of this on the user is that symbols in their formulae can not be spread out too much. Increasing the maximum range means that the user can space things out more, but the graph builder ends up putting more false connections between nodes, which then results in longer and more erroneous parsing.

There are a variety of ways of dividing up the input area around each character to determine which areas are "to the right" and "above", etc. An alternative approach to the rectangular subdivision of the input area is to test based on the angle that the line between the centre points of the items concerned makes with horizontal. For example, an item is to the "right" of another if the angle is between $-22.5°$ and $22.5°$.

This works well for individual symbols, but fails for groups of symbols such as subexpressions in a formula. Figure 3.9 illustrates this. In Figure 3.9(a) B is in the "top-right" region relative to A. However, if the length of the superscript B grows, as in Figure 3.9(b), its centre point moves into the "right" region. Using a rectangular subdivision, as this system does, or other shaped regions, such as that used by Lavirotte

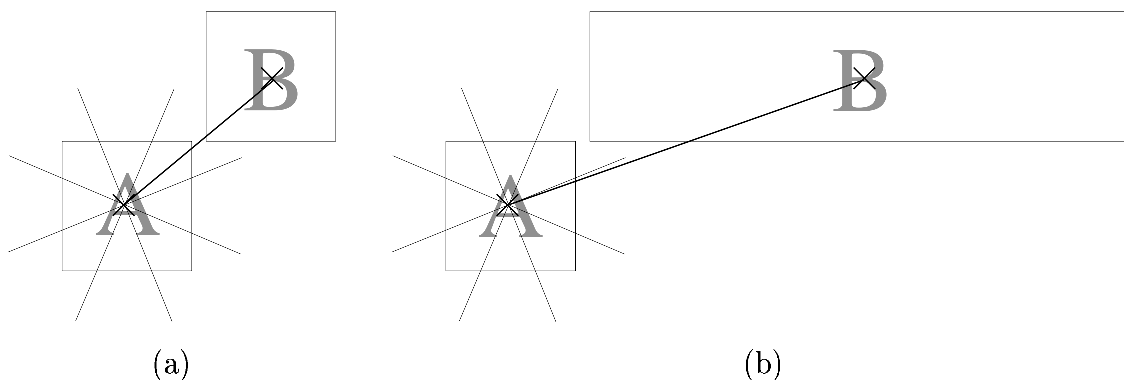


Figure 3.9: As the length of the exponent grows, it moves from the "top-right" to the "right" region.

and Pottier (1998) avoids this problem.

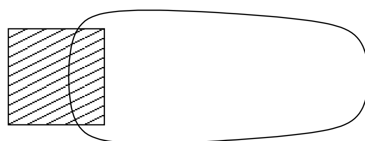


Figure 3.10: The shape of the "right" region that Lavirotte and Pottier use.

Figure 3.10 shows the region Lavirotte and Pottier use for "right". The shape of this region is based on the fact that most mathematical notations, while 2D, are based on normal reading, from left to right. As a result, symbols which have a "top-" or "bottom-" aspect to them are typically close to the parent symbol, while horizontal links (such as "left" and "right") can be far off. This is a good idea as it means that fewer erroneous links will be built. The idea behind this technique has been used by this system by making the height of the geometric regions involving a "top" or "bottom" component less than that for "left" and "right".

Overlap Check

Arcs are not built between two nodes if any other bounding regions, belonging to other symbols or subexpressions, are crossed by a line running between the centre points of the symbols or subexpressions being linked. Within the subset of mathematical formulae this system can currently process and to the limit of my mathematical knowledge, this will not restrict the system in any way.

Figure 3.11 shows a situation where an arc is not built. There is no link built

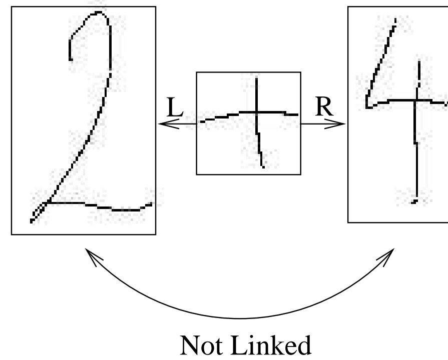


Figure 3.11: No arc is built between symbols that have other symbols between them.

between the 2 and the 4 because the +'s bounding box is on the line between the centres of the 2 and 4.

This process for building the graph works well, though possibly building too many arcs in the graph due to too much leniency in the geometric test. Unfortunately, this is a trade off between having too many arcs but being able to parse sloppy formula, and between having a more modest number of links, but requiring the input to be neat and closely spaced.

The complexity of building all the arcs in the graph varies between $O(n^2)$ and $O(n^3)$, where n is the number of nodes in the graph. The variability is due to the fact that, if the initial $O(1)$ type and geometric tests fail, then the arc building algorithm can rule out having to carry out the $O(n)$ overlap test.

Continuing the earlier example, Figure 3.6(b) shows the initial graph for the formula with the arcs built. Note, for example, that there is no link between the "x" and "4". Although the 4 is a subscript position relative to the x, no link is built because it would pass over the fraction bar.

3.1.5 Initial Graph Preprocessing

The next step is the preprocessing step. After observing that there are a subset of rules in the grammar that can be applied first, then ignored for the rest of the processing, a preprocessing grammar was made. This approach is not new, also being used by Anderson (1968). The steps in the preprocessing grammar could be included in the main grammar, if desired, though possibly at the expense of parsing time.

The result of applying all the rules in the preprocessing grammar, which do basic

categorisation of symbols and collect up multi-symbol items such as numbers, is shown in Figure 3.6(c).

3.1.6 Main Processing

The main parsing occurs once the initial graph has been created and preprocessed. The graph grammar is a set of rules which are templates describing the way various mathematical constructs are made.

The system checks all the rules in the grammar against the current formula graph, trying to find one that matches a subgraph in the current version of the formula's graph. If it is unable to find a rule that matches, it backtracks the parsing process to an earlier point that had more than one match and proceeds from there with an alternative choice. The backtracking is controlled by a priority queue using a simple heuristic. The heuristics will choose the graph with the smallest number of nodes. In the case of a tie, it chooses the search that has applied the largest number of rules so far.

The first rule that matches in this example is the "superscript rule", shown previously in Figure 3.1(b). This says that an "element" with another "element" to the "top-right" is a superscript. The A and B nodes are collapsed into a single node, their position and bounding box created from the two original nodes. Figure 3.6(d) shows the new graph. The data value for the new node is created by taking the data values of the original nodes, represented by A and B, and inserting them into the string "A^{B}".

The method used for searching for such a matching subgraph within the main graph is a brute force approach and is the main bottleneck of the parsing process. For every rule in the graph grammar, a search is done in current graph to see if it exists. To find a n node rule graph in an g node formula graph, all n node sub-graphs of the graph are generated, testing to see if any match the rule graph. The number of n node sub-graphs of a g node graph is $C_g^n = \frac{n!}{(n-g)!g!}$. This search is repeated for every rule in the grammar. So, as the size of the formula graph grows, or the size and number of the rules in the grammar grow, the searching takes longer.

Bunke and Messmer (1997) describe a method for speeding up attributed graph matching. This works by doing an initial pre-process of the rule graphs in the grammar and finding common substructures in them. They end up with a tree structure that describes how to build all the rule graphs progressively, starting from all the initial types of nodes.

The main graph is then searched for the rule graphs by, for each node in the graph, taking a node then testing all the rule graphs that could be built up starting with that type of node. This testing involves seeing, as the rule graphs are progressively built, whether they match the formula graph. Should they not match, the saving is made due to the fact that all the rule graphs that shared that non-matching sub-part are ruled out.

This technique was not used since, at the time the graph searching was implemented, it was not clear whether or not optimisations would be necessary. As this was only a prototype system, keeping the complexity of the system to a minimum was also desirable.

The next step applies the “fraction” rule. Figure 3.6(e) shows the new graph. Finally, the integral rule, shown in Figure 3.1(c), is applied. This gives the final graph, shown in Figure 3.6(e). From this point a number of productions are applied that change the node’s data type from “Element” to “Formula”, the parser’s goal. It can be seen that as the parsing proceeds, the “data” value inside the nodes builds up to the final, in this case LaTeX, representation of the formula.

If the parser gets to a point where it is unable to match any of the rules in the grammar to the current graph, it backtracks to an earlier point and tries alternative productions. Because of the current implementation of the graph processor, not having contextual information as Lavirotte and Pottier do, there are numerous cases where more than one rule can be applied to a given graph. These alternatives are tracked using a priority queue that prioritises based on the number of nodes that are in the graph and the current parse depth.

In determining the order in which to apply rules, a priority system can be used where each rule has a priority, either implicitly defined by its position in the grammar, or with an integer associated with each rule (Pottier, 1995). A grammar can be constructed so that only a maximum of one rule in the grammar can be applied at a time. Lavirotte and Pottier (1997) take this latter approach, describing a semi-automatic method for turning an ambiguous grammar to an unambiguous one by adding “context rules” that describe the conditions which must be true before a certain rule is applied. The system currently uses an implicit priority, based on the rule’s position in the grammar, which means at times erroneous productions are made and the parser has to backtrack.

The only optimisation in the otherwise brute force graph matching is that before searching the graph to see if a rule matches, an initial test is done if all the node types required by the rule do exist somewhere within the graph.

Due to the implicit ordering of rules, a “perfect” parse almost always chooses the first rule that matches from the grammar every time. After finding that the slowest part of the parsing process was checking the rules in the grammar against the current formula graph, the system always initially follows the first match found in the grammar. When this approach finally runs out of choices, the system then backtracks and looks for further matches.

The “Nothing Inside” Test

Before the system applies a rule from the grammar, an additional test is made to see if the application of the rule would mean that other nodes end up inside the nodes created as a result of the production.

As long as people do not write formulae with overlapping symbols, this does not restrict the ability of the system to parse mathematical formulae. This approach was independently developed by Miller and Viola (1998) who use the same condition, but testing with convex hulls, to limit the application of rules in their stochastic grammar.

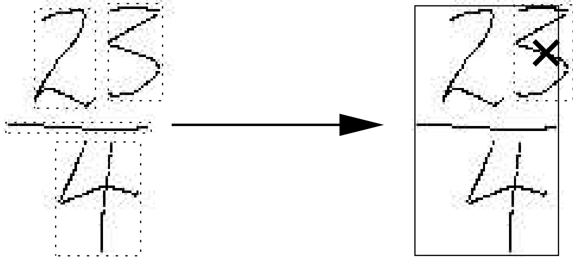


Figure 3.12: The “nothing inside” test. If a grammar rule collapsed the $\frac{2}{4}$ to a single node, the centre point of the 3 would end up inside its new bounding region. Because of this, the application of the rule is not permitted.

This test is useful because it reduces the number of rules that can be applied in a given situation, but does not limit the ability of the system to parse mathematical formulae. In situations like that shown in Figure 3.12, it is possible to apply a fraction rule that collapses the $\frac{2}{4}$ part, leaving the 3 behind. If the $\frac{2}{4}$ were to be collapsed, the 3 would end up inside the new bounding box. Noting this, we can avoid applying the rule.

A problem arising from the “nothing inside” test, is shown in Figure 3.13. A fraction has been written so that the fraction bar overlaps the integral sign. When deciding whether or not to collapse the integral $\int x^2 \; dx$, the denominator of the fraction in

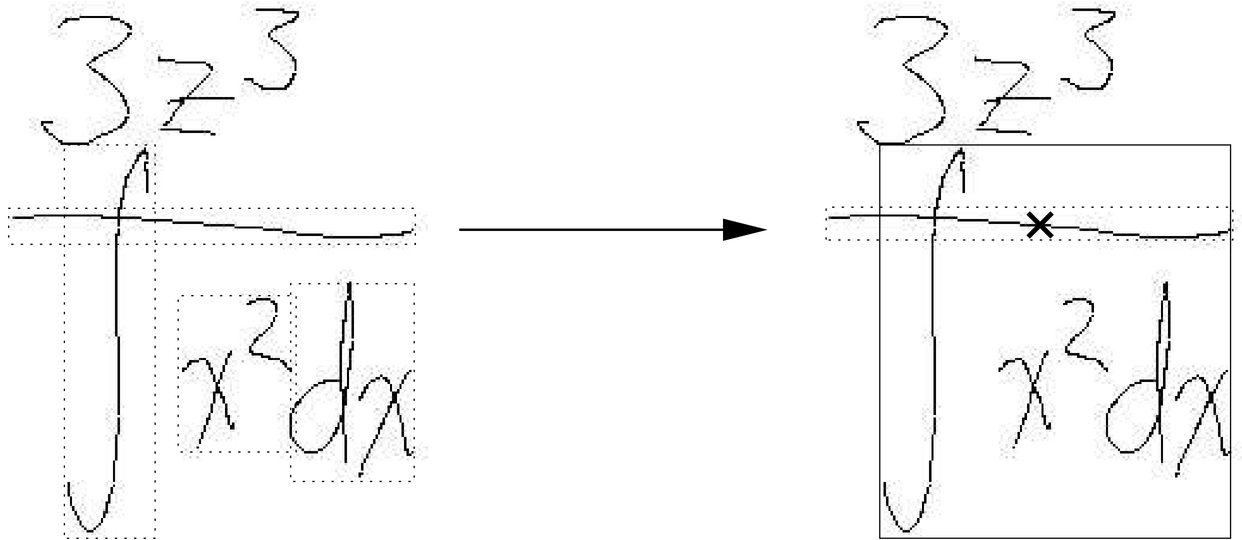


Figure 3.13: A problem with the no-inside restriction. Collapsing the integral is forbidden due to the fraction bar ending up inside it.

Figure 3.13, the bounding box that would be created is found to include the centre point of the fraction bar, so the application of the rule is cancelled. This means that the integral will *never* be collapsed, making the formula unparsable.

Fortunately, as people tend to avoid overlapping symbols as they write, the problem of symbols overlapping each other like this is not common.

3.1.7 Parser Implementation

The graph rewriting parser uses two main data structures: a labelled directed graph and a priority queue. A graph is represented within the system using:

- an adjacency matrix, that allows for quick adjacency tests.
- a collection of nodes, each having a number of attributes that store the identity of the node, its lexical class, and the position of its bounding box on the input area.
- a collection of arcs which store, in addition to which nodes they link, information about geometric relationship between these nodes.

The priority queue is implemented as a doubly linked list, the items in the list ordered by their score. Each item in the queue holds a copy of the formula graph,

along with additional information about the state of the formula processing at that point.

3.2 Formula Straightening

Excessively sloped formulae can be difficult to process reliably, no matter how good the underlying formula processor is. This is because the geometric relationships become ambiguous.

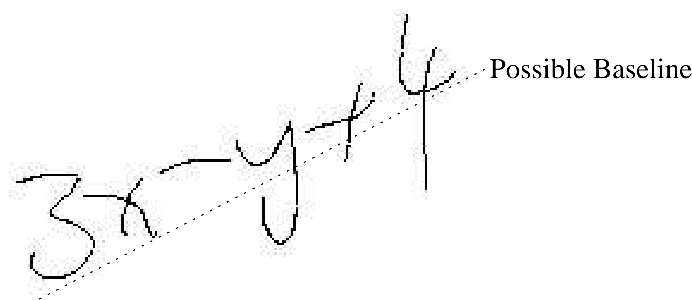


Figure 3.14: A sloped or skewed formula can be difficult to process reliably.

Figure 3.14 shows a skewed formula. While the author perhaps intended it to be $3x - y + 4$, it is most likely to be parsed as $3x^{-y+4}$.

In sloped formulae, the positions of the symbols are most commonly skewed, but the symbols themselves are not rotated. A brief investigation into the automatic straightening of formulae was carried out. Least squares linear regression and Hough transforms were used to attempt to determine the baseline of symbols entered. From the baseline determined, formulae were then skewed to bring the baseline back to horizontal. While it was occasionally helpful and showed initial promise, it had trouble with short formulae, and formulae with exponentials or subscripts. For example, if the formula shown in Figure 3.14 was meant to be $3x^{-y+4}$, the slope correction would "correct" it to $3x - y + 4$.

These ambiguities can be difficult to resolve in general, but attributes such as the relative size of symbols can be used. For example: knowing that exponents are typically smaller than the base symbol, top-right arcs will be only be built if the target node is smaller than the parent node. Care has to be taken, however, if the target node is a subexpression in a formula.

3.3 Summary

This chapter described a basic graph rewriting formula parser. It works well for small formulae, but slows down significantly as the size of the formula or grammar grow. Any future development of the system should include more advanced techniques for graph matching and take advantage of contextual information.

The graph rewriting parser is able to process formulae well, and is easy to experiment with: adding and removing rules to the grammar for different mathematical constructs is simple.

Chapter 4

The Interface

A prototype interface was built in order to test the hypothesis that a pen-based formula entry system was a viable alternative to existing systems, and that a graph-grammar based parser would be able to parse handwritten formula well. In the development of this interface a number of new interface concepts for the correction of character recognition errors were created.

4.1 Aspects of User Interface Design

There are a large number of publications (Microsoft, 1995; Apple Computer, Inc., 1987; Schumacher Jr., 1992; Smith and Mosier, 1986; Shneiderman, 1992) that discuss how to design a good user interface, some offering "standard" interface design guidelines.

Two conclusions may be obtained from these publications. Firstly, the interface should be consistent with other applications on that platform. Secondly, when a user is required to perform a task, it should be consistent with their computing experience.

For evaluating the effectiveness and quality of user interface designs, Nielsen (1994) has created a list of ten usability guidelines, which summarise most design guidelines.

- Visibility of system status. The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

- Match between system and the real world. The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

- User control and freedom. Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

- Consistency and standards. Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

- Recognition rather than recall. Make objects, actions and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

- Flexibility and efficiency of use. Accelerators, unseen by the novice user, may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

- Aesthetic and minimalist design. Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

- Help users recognise, diagnose, and recover from errors. Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

- Error prevention. Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

- Help and documentation. Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

The use of colour in a user interface is also an important consideration. While the use of colour can provide a lot of useful feedback to the user, the designer must consider the fact that monochromatic displays are still used, as well as the existence of problems such as colour-blindness. 1 in 11 males and 1 in 300 females are colour blind. 1 in 3 million people have complete colour blindness.

4.2 Pen Based Computing

The interface is designed primarily for pen input, however a mouse can still be used as all operations use at most one button. This single button operation is achieved with the pen by pressing it against the tablet.

While some pens do offer additional buttons on their barrel or at the top end of the pen, it can be inconvenient or difficult to use these with precision. Thus, all functions in addition to basic drawing of strokes, such as editing operations, should either be conveyed to the system through the use of menus, toolbars, keystrokes, or ideally specialised gestures.

There are a number of "standard" editing gestures used in pen based computing. Books such as Microsoft's "The Windows Interface Guidelines for Software Design" (Microsoft, 1995) describe a set of suggested gestures to use. Some of these gestures are based on "traditional" proofreading marks, or reasonable abbreviations. For example, as shown in Figure 4.1, a circled "u" means undo.

Figure 4.1: A suggested undo gesture.

With a system based on a character recogniser, it is inevitable that recognition errors will occur as the user writes their formulae, no matter how advanced the recogniser is. Humans often have trouble reading each other's writing, and, at times, people are even unable to read their own. Thus, a simple method for correcting errors needs to be provided.

4.3 A Pen Based Formula Entry System

A complete user interface for equation editing has been developed, using the recognition and parsing modules described earlier. The interface allows handwritten entry of mathematical formulae, correction of errors made in the automatic interpretation of what the user enters, and basic equation editing.

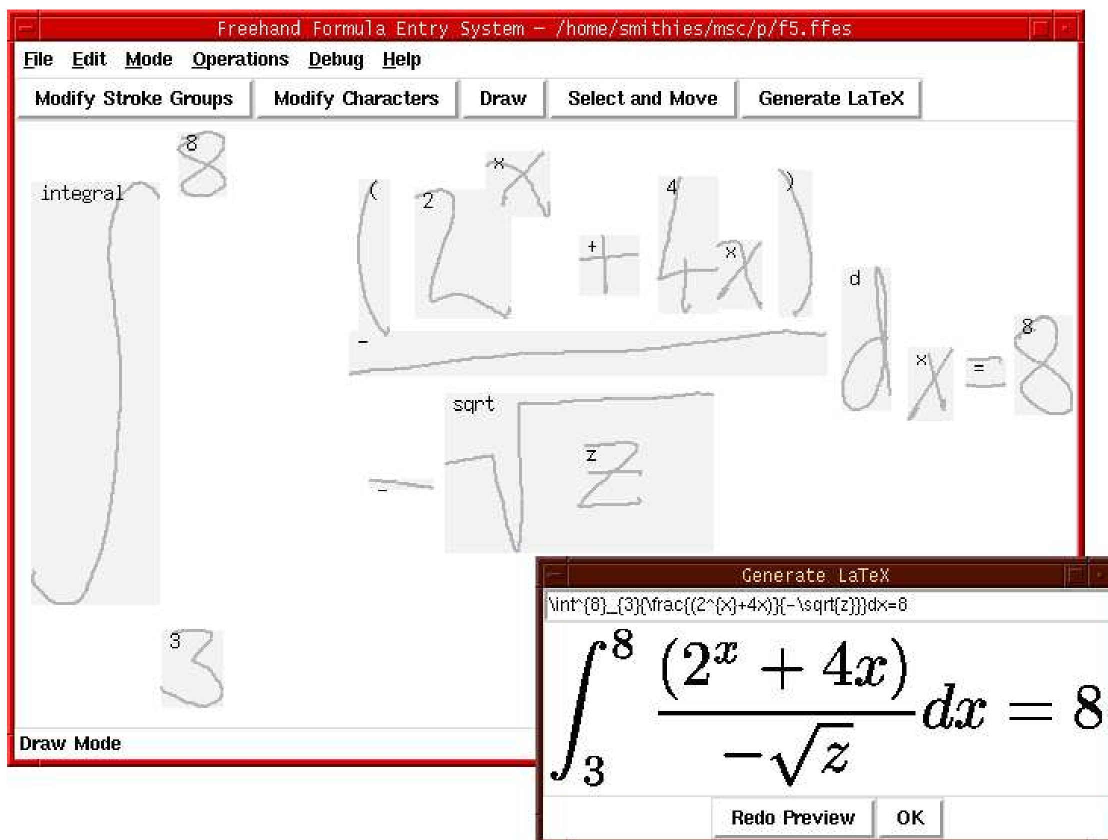


Figure 4.2: A formula that has been entered into, and parsed by, the system.

Formulae can then be parsed to determine the LaTeX command to generate them, and the result is automatically passed through external tools to generate a preview. Figure 4.2 shows a screenshot of the system. The formula has been written in by the user, and parsed by the system which then presents a typeset result. The user is able to copy the LaTeX code from the entry area at the top of the preview window and insert it into their LaTeX document.

The remainder of this chapter describes the new system created. The user interface controlling routines are written in Tcl/Tk. The parser, character recogniser, and stroke grouping routines are written in C and C++. All implementation was done on a 180MHz Intel Pentium Pro based system, running X-Windows under Linux.

Each of the elements used to make the final system would be greatly improved if they were able to take advantage of contextual information. Using contextual information has been a focus of research in formula recognition (Anderson, 1968; Lavirotte and Pottier, 1997; Miller and Viola, 1998). However, even with higher level information and improvements in character recognition systems, errors are still possible. Knowing this, designing an interface that simplifies the correction of such errors is important.

4.3.1 The Character Recogniser

The underlying character recogniser used by this system is the one used by Smithies, Novins, and Arvo (1999). Symbols are encoded as collections of polylines representing individual user-drawn strokes. The recogniser uses an extremely fast on-line recognition algorithm based on nearest-neighbour classification in a feature space of approximately 50 dimensions. Rubine (1991) and Avitzur (1992) both use a similar feature-based strategy.

To train the character recogniser, the user supplies ten to twenty handwritten samples of each character. These samples are stored and used by the recogniser to recognise the input characters.

Although the recogniser is theoretically user-dependent, the system is relatively user-independent in practice. For example, even though the recogniser was trained using samples supplied by just two people, others had little difficulty in using the system.

For each group of strokes passed to the character recogniser, the top n interpretations of those strokes are returned, along with a confidence for each interpretation. This confidence information is important as it is used by the stroke grouping method, described in Section 4.3.3.

To get higher recognition rates from the character recogniser, and thus improve the performance of the stroke grouping, more versatile classifiers, such as neural nets (Yaeger, Webb and Lyon, 1996), and perhaps the use of contextual information from later processing stages, as described by Miller and Viola (1998) could be used.

Virtually any character recognition module can be incorporated into this system. The only requirement imposed by the system on the recognition module is that it must be capable of ranking the n most likely candidates for a single pattern by a numerical measure of confidence, and that the confidence measures of different patterns must be directly comparable.

4.3.2 Basic Input

Upon startup, the program is in "draw mode" which permits the user to enter strokes into the system by drawing with the pen on the drawing tablet. As the user writes, the system automatically interprets their strokes, after waiting for the user to get a number of strokes ahead before it begins processing. Since the processing, described in Section 4.3.4, annotates the user's input, this delay helps avoid any potential distraction for the user, and also ensures that the processing will not interfere with the symbol that the user is currently drawing.

Processing also automatically begins after a user definable period of inactivity, defaulting to one second. Thus, the system will "notice" that the user has finished entering their formula, and automatically catch up recognition of all outstanding strokes. Alternatively, the user can tap the pen on the tablet or choose a menu option for the same effect.

Other basic editing operations such as selection, moving and cutting are provided as well, through a *select and move* mode. This lets the user drag a box around a region containing the characters that they wish to be selected. The contents of the selection can then be dragged around the screen at will, or deleted.

The user interface supports multi-level undo and redo, and allows for the loading, saving and printing of formulae that the user has entered.

4.3.3 Stroke Segmentation

As the character recogniser works on a symbol by symbol basis, the stream of strokes provided by the user drawing with the pen on the tablet must be divided up into separate symbols.

There are a number of approaches that can be used for stroke segmentation. These are discussed below.

Pauses Between Symbols

The user entering symbols is required to wait for a brief period, for example 500 milliseconds, between each symbol that they write. This delay allows the system to tell when the user has finished each symbol.

This is appealing because it is easy to implement, and offers a high accuracy of determining where each symbol ends. Unfortunately, it is frustrating for a user to write like this as it is not natural. Being forced to concentrate on writing slowly also takes away from the user's ability to concentrate on the formula itself.

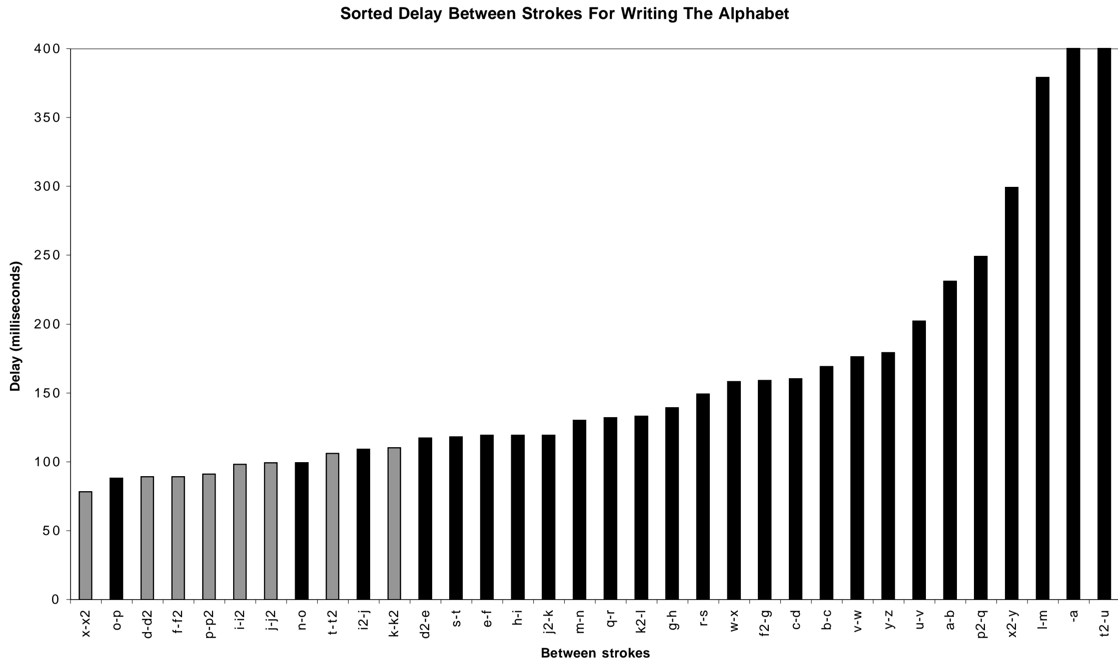
Finer Timing Information

A small investigation was made to see if the delay between strokes of separate symbols was longer than that of the delay between strokes in the same symbol. This investigation was not in depth, and only provides an indication of the possible success of such an approach. The idea was that strokes within symbols would be drawn with less delay than strokes of separate symbols.

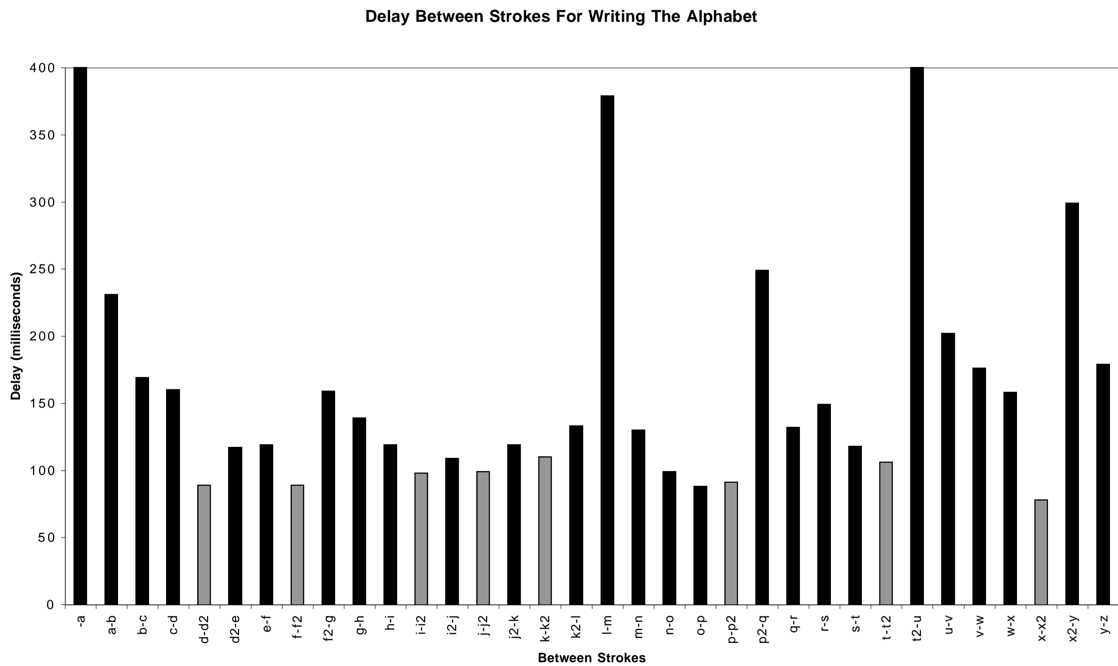
The two graphs in Figure 4.3 show the delays between strokes as the letters of the alphabet were swiftly written out from a to z. The labels on the x-axis show which strokes the delay was between. The label "$a - b$" means that this is the delay between the letters a and b. Labels such as "$x2$" indicate the second stroke for the symbol "x". The lighter shaded bars indicate the eight strokes which are second strokes for single symbols.

If the assumption that shorter delays imply that two strokes belong to the same symbol is correct, then this should be evident in Figure 4.3(a) that shows the sorted delay data. The assumption holds true apart from the $o-p$, $n-o$, and $i2-j$ delays. If a threshold is to be determined, there is a small difference in the duration of the delay between the last of the gray bars, and the beginning of the solid bars: the difference between the $k - k2$ and $d2 - e$ delays is only seven milliseconds.

Looking at Figure 4.3(b), where the data is in its original entry order, suggests that it may also be possible to determine which strokes belong to the same character by noticing that they fall in troughs. Still, this is not reliable as the $o-p$ and $q-r$ delays also come into this category, and the $p - p2$ delay which should, does not.



(a) Sorted by delay.



(b) In order of entry.

Figure 4.3: Delays between strokes for writing the alphabet.

Timing information on its own does not appear to be sufficient for segmenting strokes into symbols. It is not possible to consistantly determine where strokes should be grouped together or separated. For such a scheme to work, variations due to the user pausing or slowing down as they write would also have to be compensated for, and thresholds would have to be automatically determined as the strokes are progressively entered by the user.

Overlapping Strokes

Looking at the geometric relationship between strokes can yield some useful information. If two or more strokes overlap, it can almost be said with certainty that they belong to the same symbol. The doubt is only caused by the fact that a sloppy writer may accidentally overlap adjacent symbols. From my experience, most people writing with a pen and tablet are not inclined to overlap symbols, so this is not a problem.

We can make the assumption that any strokes that touch other strokes all belong to the same symbol. Typically, if a person is writing with reasonable care then this is not a problem.

Mathematical expressions typically consist of symbols taken from the set of Arabic numerals, Roman and Greek alphabets, along with other miscellaneous symbols and notations. Nearly all of these are drawn with overlapping strokes apart from a small minority, for example: i, j, %, !, =, $\therefore$ and Θ. In these cases, the overlapping strokes approach is going to fail.

Procedural Code to Group Strokes

If the distance between two strokes is less than some threshold, or if they satisfy some other criteria, we can consider them to be the same character. For example, if there are two "short" "horizontal" lines within y pixels of each other, then combine them to make an "=". Similar rules could be written for other unconnected characters.

Unfortunately this means that for every unconnected character such a rule has to be written, which limits the system's character set and makes it harder for an end user to extend.

It is also possible to have a generic rule that says that strokes that are approximately horizontally aligned, but vertically near to each other, are part of the same symbol. This handles cases such as i, j, and =, but symbols such as Θ and $\therefore$ need a different approach.

Combined Stroke Grouping and Formula Parsing

Let the character recogniser recognise an "i" as a "." and an "ı" (the dot and the "i" *without* the dot). These characters can then be pieced back together by a preprocessing stage (Fateman et al., 1996) or the parser (Pottier, 1995) at some later point. This is similar to using procedural code for grouping the strokes, but it is the preprocessor or parser that is making the decisions. If it is done by the parser, it allows the system to backtrack and correct any mistakes it might make.

The Approach Used by This System

Most of the methods considered for segmenting strokes into symbols did not appeal as they either relied on heuristics, or forced the user to have a certain behaviour.

If the character recogniser is able to return confidence information for interpretations of strokes, it is possible to automatically test different combinations of strokes and pick the best. This is a new approach, although is similar to that used by Yaeger, Webb and Lyon (1996) which wasn't seen until afterwards.

The approach that the system uses is to combine overlapping strokes into indivisible units, then use the character recogniser to test groupings of strokes and these indivisible units. This solution works well, only limited by the strength of the underlying character recogniser. While it is less reliable than other stroke segmentation methods, such as the user pausing between characters, it does allow a much more natural and fluid entry of symbols into the system.

The process makes two assumptions: strokes that cross belong to the same character and all the strokes that belong to the one character will be drawn before the user moves onto the next. In other words, all i's must be dotted and all t's crossed before the next symbol is drawn. From casual observation of people writing with pen and paper, and from observation of people using this system, neither of these assumptions interfere with people's writing: most people tend to write like this anyway.

The process used to determine how to segment the strokes supplied by the user works as follows:

- Determine the maximum number of strokes that a symbol can have. This can be determined by analysing the character data set used by the character recogniser. Call this maximum number m.

- Wait until the user has entered $2m$ strokes, this way there will be at least one fully completed symbol to recognise.

- Collect together all strokes that cross or touch, and call each of these strokes a single "unit". For all the remaining strokes, put them all into a unit also, each of these units having one stroke in each. There will be k, $k \leq 2m$ units.

- Generate all possible groupings up the first $min(k, m)$ of these units, and assign a confidence level to each. Each grouping corresponds to a possible grouping of strokes into symbols. The character recogniser is used to recognise the symbols in each grouping and return a confidence level for each symbol. The confidence level for a given group corresponds to the lowest confidence level across the symbols recognised in each group. This technique, of the confidence level of a group as a whole being equal to the lowest confidence within it, is often applied in expert systems (Turban, 1992).

  Two other methods investigated for determining the confidence of the overall group were:

  - to use a product of the confidences of each symbol within a group. This unfairly penalises groups with more symbols.
  - to use the average confidence of the symbols in a group. Although this worked, it can boost what should intuitively be a low scoring group. For example, if a symbol has a confidence of near zero, other symbols contributing to the average can pull up the overall score.

  The relative confidence scores for two sequences of confidence levels using these three difference methods is illustrated in the following table:

Symbol Confidences	Minimum	Average	Product
0.8 0.3 0.1	0.1	0.4	0.024
0.8 0.3 0.1 0.8	0.1	0.5	0.0192
0.8 0.0 0.1	0.0	0.3	0.0

- Of all the groups generated, select the group with the highest confidence and return information on the group selected to the user interface. This information includes the stroke groupings and what each group of strokes was recognised as. The system also includes alternative recognitions for each character for use in *modify characters* mode, described in Section 4.3.6.

With the current character training data, $k = 4$ so all groupings of 1, 2, 3, and 4 units are generated and tested. Due to the assumption that all strokes belonging to

the same symbol are drawn in order, we can avoid a combinatorial explosion. The total number of combinations of up to the first k units is $2^k - 1$.

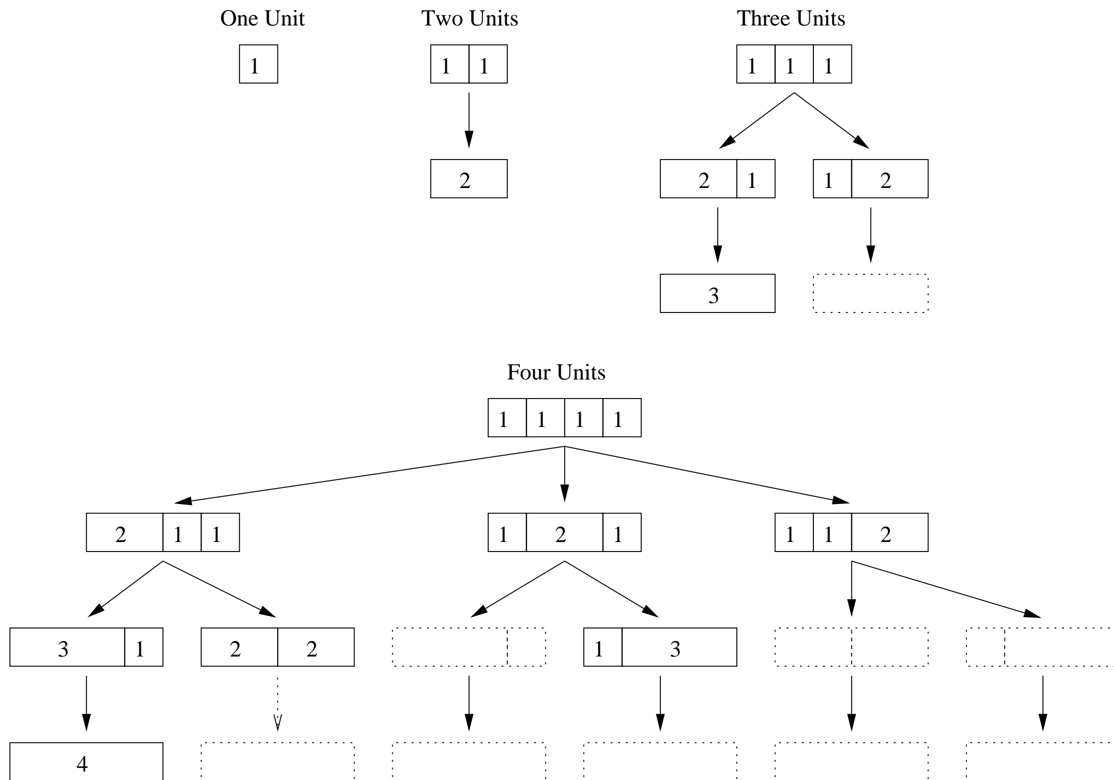


Figure 4.4: All groupings for four units.

Figure 4.4 illustrates the new method developed for generating all the groupings of up to four units. To generate all groupings of n units, the tree is built starting with a parent node of n 1's. The children of a node are then created by the addition of all adjacent pairs of boxes. Children that already exist elsewhere in the graph are not generated. The children that are not generated are represented by dotted boxes.

These groups indicate the groupings of strokes to try. For example, the group "1 2 1" means: "Take the first unit on its own, then the next two units together, then the last unit on its own."

4.3.4 Online Annotation

As the user writes, the system runs a background thread which applies the above stroke grouping technique and recognises the symbols that the user has written.

As mentioned in Section 4.3.2, there is a delay between the user's writing and the system's processing of their strokes. The number of strokes ahead that the system

lets the user proceed is automatically determined by analysing the data used by the character recogniser. The limit is set to *twice* the number of strokes in the character with the largest number of strokes. With the current set of data for the character recogniser, the lag is eight strokes.

As characters are recognised, the system places a shaded bounding box over them and annotates the box with the character that the system has determined it represents.

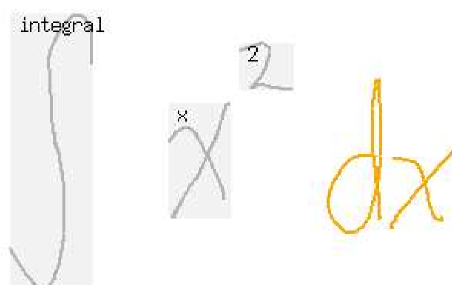


Figure 4.5: A user beginning to enter a formula. The first three characters have been recognised, and the remaining two are still waiting to be recognised.

Figure 4.5 shows a screen capture of the drawing area of the program as a user is beginning to enter a formula. The first three symbols have been recognised by the system and their bounding boxes are marked and annotated with the system's current interpretation. As a character is recognised, the colour of its strokes are changed to indicate that the recognition has taken place.

4.3.5 Stroke Regrouping

The strokes entered by the user are initially grouped by the stroke grouping algorithm, described in Section 4.3.3. The algorithm tries all possible grouping of strokes, ranking each based on confidences returned by the character recogniser. As the success of this algorithm relies on the accuracy of the character recogniser, it is possible that strokes will be incorrectly grouped.

From a study of new users using the system with a character recogniser trained to a style of writing similar to theirs, 13% of the characters they wrote were misgrouped by the automatic stroke grouping process. This does improve if the user has trained the character recogniser.

A simple and effective method is required to fix grouping errors. There are two possible situations that the user has to correct:

- Strokes that should be recognised as a single character are grouped as parts of separate characters, or
- Strokes that should be recognised as part of separate characters are grouped into a common character.

The user can correct both of these types of errors by entering *modify stroke groups* mode. In this mode, drawing with the pen will temporarily mark out a line. Upon finishing the line, any strokes that were touched by that line are forced into a group of their own, possibly causing a regrouping of other strokes. The temporary line then disappears, and the system automatically reruns the character recogniser on all affected groups. This technique has been called "squiggle select", as the user can either draw a line or squiggle over the strokes they want grouped together. SGI Inperson (SGI, 1999) uses the same technique for selecting objects in a multi-user collaborative white-board application.

Drawing a line through the strokes to be grouped is better than circling or drawing a box around the strokes. A single line is able to target specific strokes easily, and is easier to draw. If there is a group of closely spaced strokes, it is easier to pick out individual ones with a single line.

Figure 4.6 shows the *modify stroke groups* mode being used to correct the two types grouping errors. Figure 4.6(a) shows the initial state, in which the strokes in the "=", the "4" and the "2" are not correctly grouped.

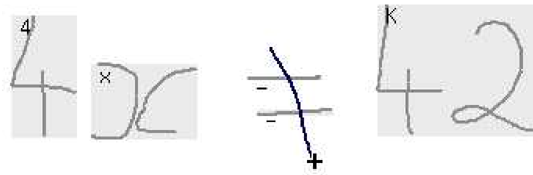
First, the user draws a line through the two strokes of the "=" that should be combined into a single group, as shown in Figure 4.6(b). Figure 4.6(c) shows the result after the pen was lifted. Note that the temporary line has disappeared and the "=" has now been correctly recognised.

To split the "4" and the "2" apart, the user draws a line through one or more strokes that should be split off from the larger group. In Figure 4.6(d), a line is drawn through the two strokes of the "4". A line through the "2" would have had the same effect. Figure 4.6(e) shows the final formula, with the strokes now correctly grouped and recognised.

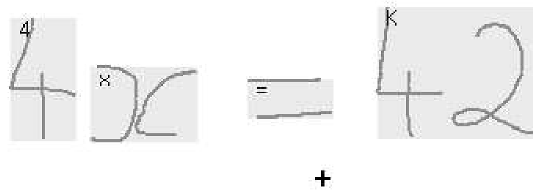
4.3.6 Modify Characters

No matter how good the underlying character recogniser is, errors in the recognition of symbols are still going to occur. The *modify character* mode allows the user to click on a misrecognised symbol's shaded bounding box and select from a pop up

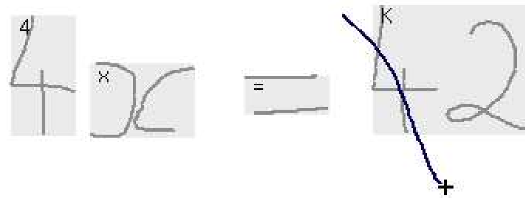
(a) Initial grouping.



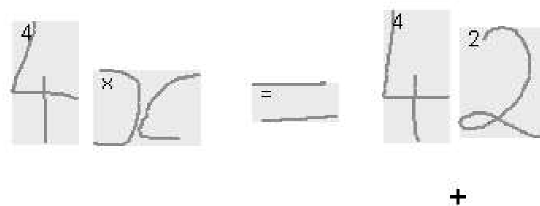
(b) The user indicates that two strokes should be grouped together.



(c) The system displays the regrouped and re-recognised characters.



(d) The user indicates that two strokes should form their own group.



(e) The final result.

Figure 4.6: Modifying stroke groupings.

menu the correct interpretation for that symbol. The pop up menu contains the best choices, currently the top five, supplied by the character recogniser for that grouping of strokes. If there are repeated symbols in the top five, due to the fact that the character recogniser is able to recognise multiple styles for individual symbols, all but one of the occurrences are removed. Should the character that the user desires not appear on this pop up menu, the user may chose an *enter* option, and type the correct character from the keyboard. Symbols that do not appear on the keyboard are entered using a long-hand name. For example, "Σ" is entered "`Sigma`". The user is also able to choose non-keyboard systems from a toolbar.

Figure 4.7 shows a user correcting a misrecognised character in modify character mode. The "z" in Figure 4.7(a) that the user drew was misrecognised as an "2". By clicking on the character a pop up menu appears, as shown in Figure 4.7(b). Selecting the correct choice from this menu then overrides the recogniser. Figure 4.7(c) shows the corrected character.

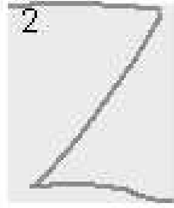
Even though the pop up menu correction method is easy and intuitive, users found the process of correcting character interpretation errors somewhat tedious if the correct alternative was not on the pop up menu. Having to resort to manually entering the character is distracting as it requires that the user switch from using the pen to using the keyboard. High character recognition rates are therefore very important, and any serious user of the system must take the time to train the recogniser with their own handwriting.

4.3.7 Parsing and Preview

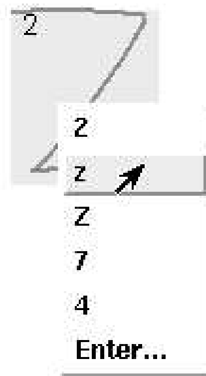
When the user selects the menu option to parse the formula, the system pops up a window that shows the workings of the parser as it attempts to process the formula. The parsing can be cancelled by the user at any point.

Figure 4.8 is a screenshot of the system part-way through processing a formula. This is the same formula as shown in Figure 4.2. In the screenshot you can see the graph rewriting formula processor displaying its current graph. As the parsing proceeds this graph is updated. This graph was primarily intended as a debugging aid and is of little use to most users of the system. However, someone who understands how the system works can use it to diagnose why their formula may be taking a long time to parse, causing erroneous parsing, or causing the parser to be unable to parse the formula at all.

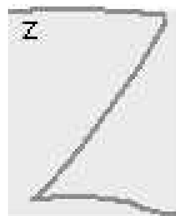
Once the formula has been parsed and a LaTeX string generated for it, external



(a) Initial interpretation.



(b) Selecting the correct interpretation from the pop up menu.



(c) The corrected character.

Figure 4.7: Correcting a misrecognised character.

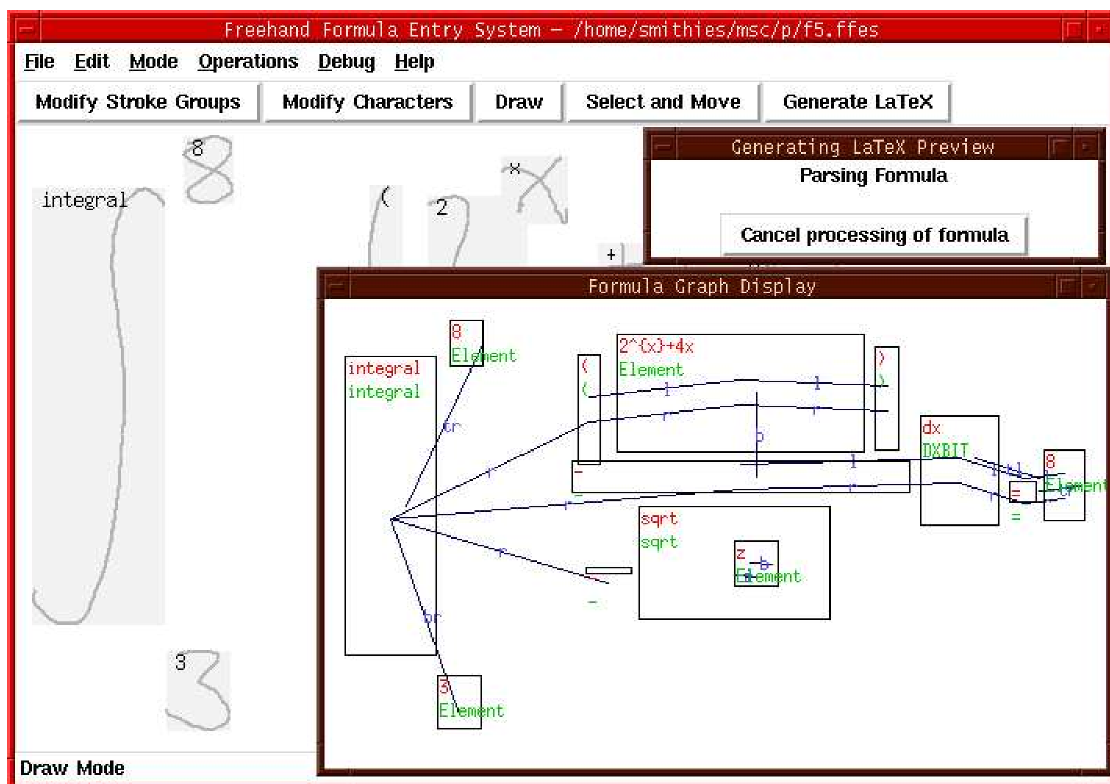


Figure 4.8: A formula being processed.

tools are used to generate a preview image which is then displayed. While the parsing at times can be quick, less than a second for small formulae, the preview generation is currently slow: taking eight to ten seconds per formula.

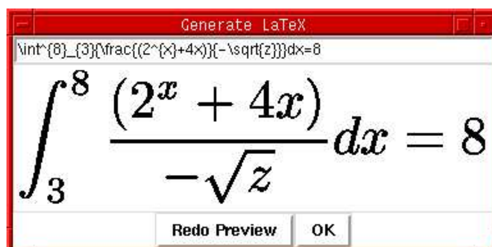


Figure 4.9: The LaTeX preview window.

Figure 4.9 shows the window that appears after the preview is generated. The LaTeX command string at the top of the window is in a text entry area and can be copied and pasted into the user's LaTeX document. The user is also able to edit this command string and press the *redo preview* button. This regenerates the preview for the new, edited, command string. However, editing in this field does not change the formula entered in the main pen-based formula editor. This facility is provided for the user if they wish to make small changes and quickly check their effects.

Should the parser be unable to process the formula successfully, it indicates to the user what the best parse found was. This is described in Section 4.3.8.

4.3.8 Correcting Equation Parsing Errors

The most difficult problem that the user faces is that the formula parser sometimes fails to recognise the user's formula. If this happens, the user has to rearrange their formula so that it is parsable.

If the system is unable to parse the formula, the system shows the "best" parse found so far by boxing sections for their formula. Figure 4.10 shows the display that the system gives the user when it is unable to parse the user's formula. In this example, there is a single limit on the integral, but the parser's grammar does not accept this construct. As a result, the best that the parser was able to do was reduce the underlying graph to two nodes, these nodes being one for the integral itself, and one for the extra limit. This is displayed to the user by outlining the bounding boxes of these two nodes, these outlines overlaid on the user's input.

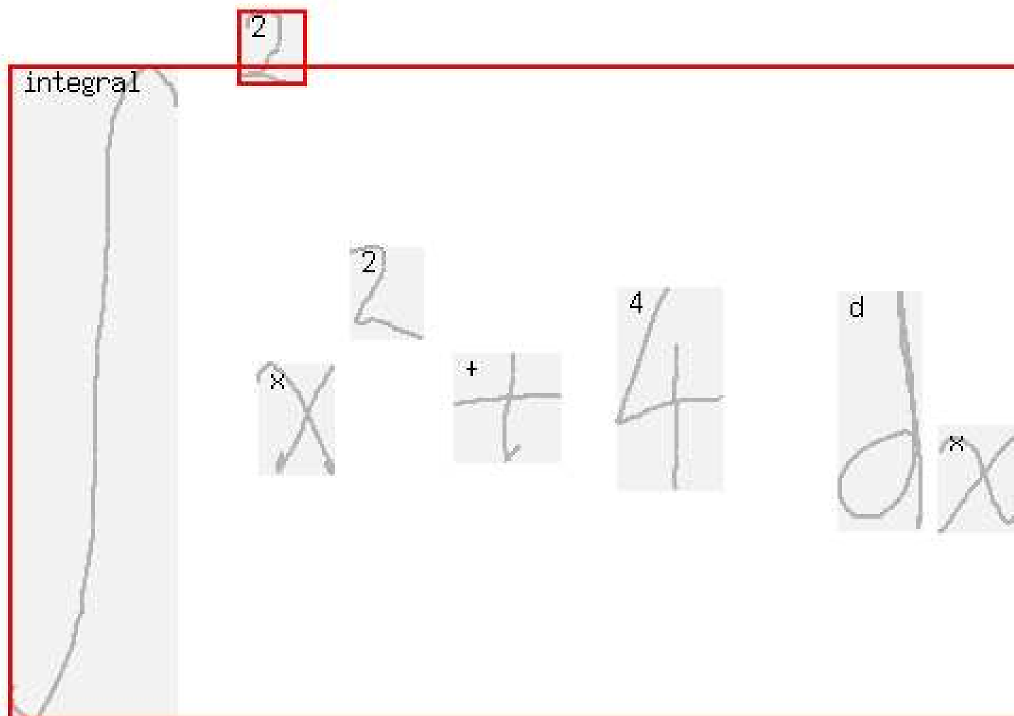


Figure 4.10: The display for a formula that the system was unable to parse.

The graph grammar based parser allows for some leniency in the placement of characters so it usually parses formulae that are part of the grammar on its first attempt. Nonetheless, significant deviations in placement from what the grammar expects can cause parsing failures. In such a case, the user must manually realign the input characters by using the *select and move* mode.

Other problems that lead to the misparsing of a formula include misgrouped or misrecognised characters that are not noticed by the user. When the formula does not parse, the user has to find these mistakes and correct them using the *modify stroke groups* or *modify character* modes.

Chapter 5

User Testing

Until an application has undergone user testing, it is not possible to confidently say whether or not it is truly effective and useful. User testing can provide insights into how people may use a system, in contrast to how you may envisage them using it. Those who develop a system may find it easy to use and understand because they have been using it over the period it was developed, and thus are unable to give an unbiased evaluation of a system. They are too familiar with it.

In a user test a number of people, representative of the final intended users of an application, use it and give feedback. Usually, they will be observed and asked questions to determine if the design of an application is logical, understandable and usable, and if there were any flaws and inconsistencies.

The main motivation for user testing this system was to evaluate the desirability of a pen-based formula entry system: to see if it is something that people would want and use over existing systems, such as command-string or template-based editors. It also provided the opportunity to see how people found the user interface ideas: the modify stroke groups with a squiggle select and the pop up menu for overriding the character recogniser. It also provided the opportunity to gather data on the accuracy of the new stroke grouping algorithm and the performance of a graph rewriting formula parser on handwritten input.

This chapter discusses aspects of conducting user testing: designing the test, choosing participants, ethical considerations, running the test itself and the post-test analysis. This chapter also discusses usability inspection, another method for evaluating user interfaces.

5.1 Designing the Test

There are a number of ways to conduct user testing. The first, and possibly the most common, is the “thinking aloud” method (Dumas and Redish, 1993). When doing this sort of testing, an observer sits with the participant as they use the system and the participant is asked to voice every thought that they have related to using the system. If the participant forgets to talk, the observer prompts them. Also, if more information on the thoughts that user is having is required, the observer can ask them what they are thinking. Ideally, the observer is not anyone who wrote the system or has a vested interest: impartiality is desirable.

While this method has the advantage that you can “get inside” the user’s head and find out what exactly they are thinking, the user has to remember to always voice their thoughts. Remembering to and constantly vocalising one’s thoughts is a skill that very few people have, as it is something that people do not usually do. Vocalising thoughts also has the disadvantage that it slows the person down and breaks the natural flow of operation and thought that they might normally have while using an application (Wildman, 1995).

Paired user testing (Wildman, 1995) is an approach that tries to overcome the awkwardness of getting a person to think aloud as they work. In this approach, the user testing of a system is done by pairs of users, who are both working with the program on a single machine at the same time. The users are instructed to discuss what they are thinking and doing. When they strike a problem, they are encouraged to discuss the problem and how they are going about solving it.

Paired user testing gives a much more natural and relaxed interaction with the system. It can be argued that results gleaned from such a study are more representative of a real world situation as it reflects the fact that in the real world people tend to rely on their peers for advice and help with applications that they are not familiar with. With this approach, it is easier to gain insight into how they solve the problems that they face.

For the user testing of this system, the participant used the system while an observer (myself) watched them and helped them when they were not able to solve problems themselves. After they had finished using the system, The observer then discussed any issues that may have arisen throughout the testing. This was supplemented by an anonymous questionnaire and a verbal question session.

The thinking aloud method was not used as it would have meant that a large amount

of time would have had to be spent transcribing and analysing the users' comments. This is something that was outside the scope of this thesis. The system is also relatively small so, as the aim of the testing was to get feedback on the pen-based entry and new interface ideas, the questionnaire and verbal questions were sufficient.

5.2 Choosing Participants

The ideal users for testing a system are those in the application's target population, having a full knowledge of the terminology and techniques in their field, and a knowledge of the tasks that an application will be required to be capable of. Testing with people from the target population also means that they have a comparable computing skill level to that the end users will have.

Landay (1997) recommends that if such ideal users are unavailable, a "closest approximation" can be used instead. For example, if a system is being targeted for doctors and it is not possible to arrange to have a real doctor test your system, then a medical student can do the testing with almost equivalent skill, and provide similar feedback to what the doctor would have given.

The goal of testing of this system was to get feedback on pen-based formula entry systems, and the new interface concepts created. A number of people were chosen that effectively represent the people who could be expected to use a formula entry system: mathematicians, physicists, computer scientists, and high-school students.

One important aspect to consider is the number of participants to involve in the user testing. If too few are used, only a small number of problems in an application will be found. Too many, and a lot of time will be spent with minimal returns for the additional time and effort of organising and analysing the additional participants' results and responses.

A large amount of the literature on doing user testing and usability studies discusses this important issue of how many people to involve. The curve shown in Figure 5.1 is typical for the number of evaluators and the proportion of problems found with a user interface, taken from Nielsen (1994).

This suggests that the "ideal" number of people to test or evaluate a system lies between four and ten. Nielsen (1994), for example, believes that evaluation tends to work best with three to five evaluators. Dumas and Redish (1993) suggest six to twelve participants for user testing. These figures apply to large systems, such as word processors or email programs. This system is relatively small in comparison.

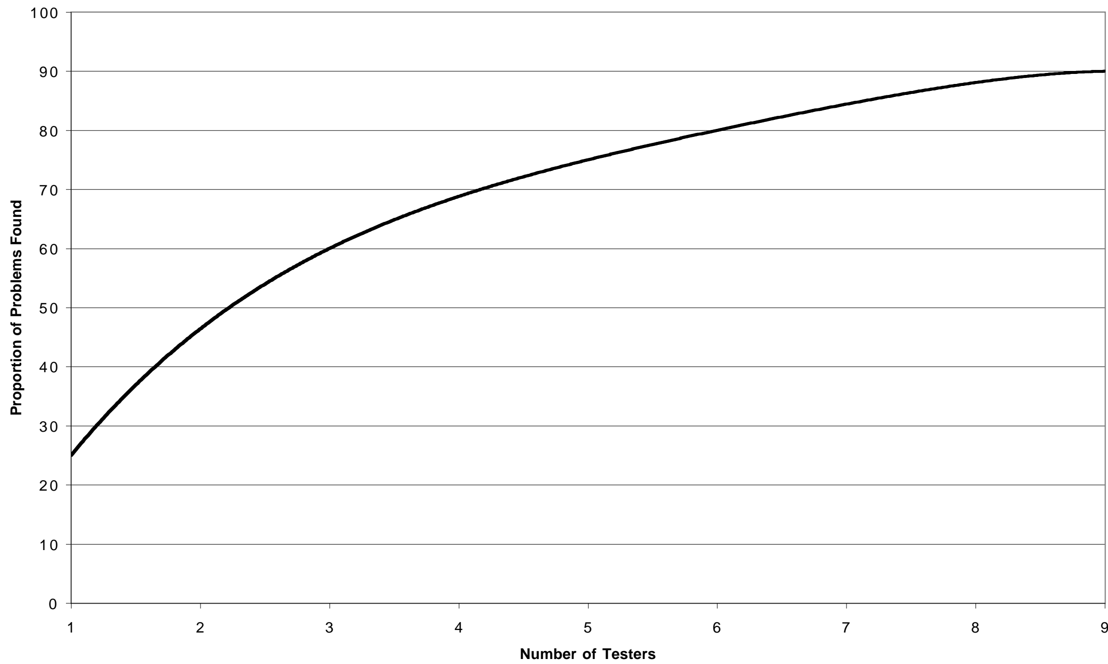


Figure 5.1: The proportion of problems with a user interface found as the number of evaluators is increased.

A total of nine participants took part in the testing, though one of these solely observed another participant using the system and made comments based on what he saw, without using the system himself. The nine participants consisted of two high school students, two physics postgraduates, two mathematics postgraduates, and one postgraduate and two undergraduate computer science students.

There are a number of different ways that participants can be obtained for user testing. You can advertise for participants, then screen them as they apply to see if they are suitable. You can go through a recruitment agency, giving them a list of the criteria that the participants must have, and they will do the screening for you. It is also considered acceptable to use personal networks to get participants (Dumas and Redish, 1993), though this is not ideal. Care has to be taken because, as such participants are likely to know you personally, their impartiality may be limited. They may be inclined to be kinder in their responses and opinions, hoping to tell you what you would prefer to hear, rather than be honestly critical. Balancing this is the possibility that friends may possibly be less shy during the test, due to them not being apprehensive of saying critical things about something to someone that they do not know.

The people who tested this system comprised of three people I, the observer, had never met before, and six people I already knew: either as friends of friends or personal

friends.

Of the people who used the system, six had not seen it before, two had seen it, and only one had used it before. This meant that there was little bias from participants' previous experience or knowledge of the system. Only two of the users had used a pen and tablet before, unfortunately resulting in the remainder of the participants having to become familiar with the pen and tablet as they used the system. As a result, this tended to decrease the ease with which they were able to use of the system. From my personal experience, the time it takes for a person to become comfortable and accurate with a pen and tablet varies between hours and days.

5.3 Ethical Considerations

One important thing that must be addressed when doing user testing are the ethical considerations. These are the aspects that relate to how the user doing the testing is treated and how they might possibly feel as a result of the testing.

It is likely that the person testing the system can get the impression that it is them who is being tested, and not the system itself. This is quite easy to believe, especially if the testing involves the person being video recorded, along with them being either directly observed by a person sitting beside them, or through one-way glass. Landay (1997) tells of how users have left user testing sessions in tears as a result of the pressure they felt and their frustration with and embarrassment from not being able to use the application they were testing.

The user testing participants must be aware that participation is voluntary and that they are able to leave at any time they wish to, as well as being allowed to take breaks as they need them. Ideally, a user test should last no more than one or two hours (Dumas and Redish, 1993). Aspects of the environment that the user is in need to be considered as well; you make the user physically comfortable. The aim is to get the participant sufficiently relaxed so that they behave as normally as possible, thus giving a real indication of how the system would be used in the "real world." If the environment is set up properly, the participants' comments will relate to the application and not external factors.

For the testing of this system, an entire session with a single user would take about an hour. This time includes an introduction to the system, the testing of it itself, and then the oral and written questionnaire. Over this hour long period, I have casually noticed that after about forty minutes the user is likely to start getting

bored or frustrated with problems that they might be having. This time did vary somewhat, though, depending on the amount of trouble they were having and the degree of enthusiasm that they had towards taking part in the testing.

Some organisations require formal ethical approval before allowing any user testing to take place. In the ethical proposal, the researcher sets out what they propose to do and how they have addressed the possible issues that may arise as a result of their testing. At the University of Otago, where this research was carried out, researchers must go through an approval process before being allowed to do user testing. These steps involve the researcher submitting a document outlining the process of the testing that they intend to do, detailing the intended treatment of the participants, confidentiality, informed consent, the participant's right to withdraw and the intended use of the data gathered. The ethical statement prepared is included in Appendix B.

In accordance with these rules a consent form was created, reproduced in Appendix C, for the participants to read and sign. It outlined what they were going to be doing and what was going to happen to them. It described the conditions under which the testing was going to be done, and what was going to be done with the information gathered. As each participant's activities were video taped, it was explained that the purpose of the video camera was to record the screen and that the tapes were to be later reviewed to gather information about the behaviour of the system. A monitor connected to the video camera allowed them to see that only the screen was being recorded. The participants were also informed of the points covered in the consent form verbally.

5.4 The Test Itself

As the use of a pen and tablet was new to most users and most of the users had not used, or even seen, this system before, each testing session started with an initial training phase. The users were able to become familiar with the use of a pen and tablet and become familiar with using the system. The training was conducted by first giving them a short demonstration of the system, then the observer guiding them through a set of practice formulae. These formulae were similar in complexity and structure to those that were used in the main testing phase. When the users felt that they were ready, or had worked through all the formulae, they moved onto the main testing phase.

During the training the possible additional pressure of being observed was minimised by not having the video camera running.

To guide the testing session, users were given a set of five formulae to work through, starting with a simple one and progressively increasing in complexity. Having a set of predetermined formulae meant that all the participants were doing the same things. As a result, it was possible to make comparisons and statistics across the users who are used. It also means that if, in the future, further testing is done, comparisons can be made across users using the system under different conditions.

The five formulae that were entered by the users for the unaided part of the user testing are shown below. These formula are representative of the complexity of formulae that the formula processor's current grammar can handle.

$$(1) \qquad x^2 + 4$$

$$(2) \qquad \int x^2 + 4dx$$

$$(3) \qquad \int_0^2 \frac{x^2 + 4}{4} dx$$

$$(4) \qquad \sum_{z=0}^{9} z^3 + 4z + 2$$

$$(5) \qquad \int_3^8 \frac{(2^x + 4x)}{-\sqrt{z}} dx = 8$$

As the participants used the system, their activities were recorded by a video camera pointed at the computer's screen for later analysis. Dumas and Redish (1993) give floor plans of a number of professional user testing suites, all having a number of video cameras trained on the user and computer, and often additional observers watching through one-way glass.

An additional way to monitor participants' activities is to modify the application, or run an additional program, so that an electronic record of the user's actions is kept. Individual keystrokes and mouse activity can be recorded. Automatically generated reports on various aspects of their use of the user interface can then be produced, as well as providing logs for later analysis. The user testing of this system relied solely on the video recordings of the user's screen and any observations made by the observer watching the users use the system.

On the conclusion of each testing session, the participants were asked to respond to both an oral and written questions about the system. The advantage of a written or online questionnaire is that nobody has to transcribe the participant's answers, and it

also offers the opportunity for anonymity: the participant can freely express themselves without fear of retribution. Having a verbal question session means that the questioner is able to ask additional questions to get more information in relation to answers that the participant has given, or as a result of something the participant did during the testing. It is important to design the questions so that they are non-leading. It is also suggested to have some redundancy in the questions that you are asking: designing questions with overlap.

For the user testing of this system, the participants filled out an anonymous written questionnaire, then answered a number of questions that were asked verbally by the observer. The anonymous questionnaire, included in Appendix D, gathered basic information to gauge the user's experience with computers and formula-entry type systems. It then asked questions to get the user's overall opinion of the system, and how they found it in comparison to other systems that they may have already used. The verbal questions, included in Appendix E, sought the participant's opinions of the new interface concepts developed. Some overlap did exist as a number of comments received in the verbal questioning were duplicated from their answers in the anonymous questionnaire.

5.5 Post-test Analysis

After the testing was completed, it was necessary to analyse the information gathered and to draw conclusions from it. Landay (1997) suggests to:

- summarise the data gathered for statistical measures such as error rates.
- make a list of all the "critical incidents." These are all the major points that occurred, both positive and negative.

Analysis of the above will indicate if the user interface really did work as expected. This analysis may result in the creation of a list of modifications and improvements that should be made to the system.

After the user testing was complete, the video recordings of the computer's screen were reviewed, information on the error rates and the time it took them to enter each formula recorded. All responses to the anonymous and verbal questionnaires were then collated.

The above results were then analysed. The conclusions are presented and discussed in Chapter 6.

5.6 Usability Inspection

An alternative to user testing is *usability inspection* (Nielsen and Mack, 1994). A number of experts in the field of user interface design methodically work through an application and evaluate its usability. There are a number of methods for evaluating usability. For example, it can be judged using a set of heuristics: a set of ideals for user interface designs, or against a cognitive model of the way that people interact with computer programs. The experts are experienced in the design of user interfaces, and can reliably find a large proportion of the major and minor problems in a user interface's design.

Unfortunately, these experts do not come cheap. A cost of US$500 to US$1000 per evaluator is typical (Nielsen, 1994). While this approach works best with interface evaluation experts, either software engineers or users who are knowledgeable in the application's target field, can be substituted for the experts. The non-experts are introduced to and taught about user interface evaluation, then allowed to evaluate the system.

While not using experts reduces the cost, the benefits gained are also fewer. From the studies done by Desurvire (1994), it is seen that the proportion of problems in a system that experts found was five to ten times more than non-experts found. This finding was consistent across all different types of usability inspection methods. The experience that experts have with usability guidelines, user testing and the design of user interfaces leads to more effective usability testing and reporting of problems with user interfaces.

Formal methods for evaluating user interfaces, while they may be successful in finding user interface flaws, are still restricted by the fact that they do not use real users. The interfaces are evaluated by an expert using a set of ideals or an approximate psychological model of humans. The other problem with experienced evaluators is that, while they may by skilled in evaluating user interfaces, they may not know much about the field an application is targeted for and skills that people in that field have, for example: how much experience they have in using computers. Nielsen and Mack (1994) state that usability inspections, having experts analyse the interface, is not yet a substitute for user testing with real users. There is no completely accurate model of the human mind predicting how people will think and react when using a user interface. It is hard to predict what a real user will actually do when faced with a given situation. Even the reactions across a number of real users, all faced with the

same situation, varies. Because of this, a common approach in an application's design cycle is to initially do a usability inspection first, then, after revising the system and addressing the issues found, doing full user testing.

Nielsen and Mack (1994) point out that when choosing to do either no testing at all or some testing – no matter how limited in scope, formality, number of participants, or skill of the testers – the limited testing is still better than none at all. You are still going to get some insights into the usability of an interface and get suggestions and ideas on how it could be improved.

A true user interface analysis was approximated by a self-evaluation against the ten usability guidelines that Nielsen (1994) proposes, and were discussed in Section 4.1. The results of this are presented in Section 6.2.

Chapter 6

Evaluation

This chapter first reports the findings based on the user testing: overall impressions, working styles, timing information and error rates. It then evaluates the user interface with respect to how the test users found it. After evaluating the effectiveness of the user testing, it gives an informal usability evaluation of the user interface with respect to Nielsen's ten usability guidelines (Nielsen, 1994). This chapter ends with a summary of the positive and negative aspects of the system.

While a total of nine participants gave a good indication of the usability of the system, statistical results generated from the user testing information should be treated as preliminary results. For more statistically valid results, a larger number of participants is required.

6.1 User Testing

Participants in the user testing found the interface easy to learn, easy to use and effective for entering formulae. A number of the participants also asked when a fully featured version of the system would be available. When asked to rate the style of interaction that this system used against other systems they had used (typically Microsoft's equation editor, and LaTeX), on a scale of 0 (Worse) to 5 (Better), the answers were all at or above 3, with an average of 4.2.

Testers who were mathematicians, those who would have the highest requirements of such a system, remarked positively on the system and what it was able to do. Their main criticism was that there was only a limited subset of mathematical formulae that the system could understand, thus being of no present value to them. As the system is based on a grammar loaded at run-time, this grammar can be edited so that it can

perform as they desire. Ideally a GUI tool would enable end users to work with the grammar, adding or changing any mathematical constructs as they desire.

Participants in the user testing filled in an anonymous questionnaire and then answered verbal questions. The answers given for the anonymous questionnaire are included in Appendix F. Answers given during the verbal questioning are in Appendix G.

6.1.1 Working Styles

After a short period of using the system, all but one of the participants in the user testing settled into a style of working in which they would write the entire formula, then go back and make any necessary corrections. This style of operation proved to be a more efficient and comfortable method of formula entry than stopping to correct any grouping or recognition errors as they occurred.

This working style may have been partly induced by the fact that the system required the user to interrupt the flow of drawing in order to press a button on the toolbar, to change the system from drawing to one of the two correction modes. Had it been possible to correct mistakes without having to go through a "change-mode" step, this working style may have not been so prominent.

6.1.2 Time Spent Entering and Correcting Formulae

Figures 6.1, 6.2, 6.3, 6.4, 6.5 show the drawing and correction times for each of the five test formulae. Figure 6.6 shows the total times taken by each participant across all the formulae. There is no total time for participant seven, as they did not enter formula two or formula three. Figure 6.7 is a pie graph showing the proportion of time spent by the average user in various stages entering and correcting formulae.

Time spent by users fixing their own mistakes, for example writing the wrong symbol and then having to delete it and rewrite it correctly, is included in this drawing time. The drawing time also includes waiting for the interface to time out and recognise the outstanding strokes that the user has drawn.

Over the period of entry for the five formulae, the total entry times for the participants in the user testing ranged from 117 to 236 seconds, with an average of 194 seconds. An expert user who is familiar with the system is able to enter the same formulae in around 68 seconds.

The difference between the ideal and observed time is due to the fact that most participants did not write quickly with the pen and tablet, due to unfamiliarity and

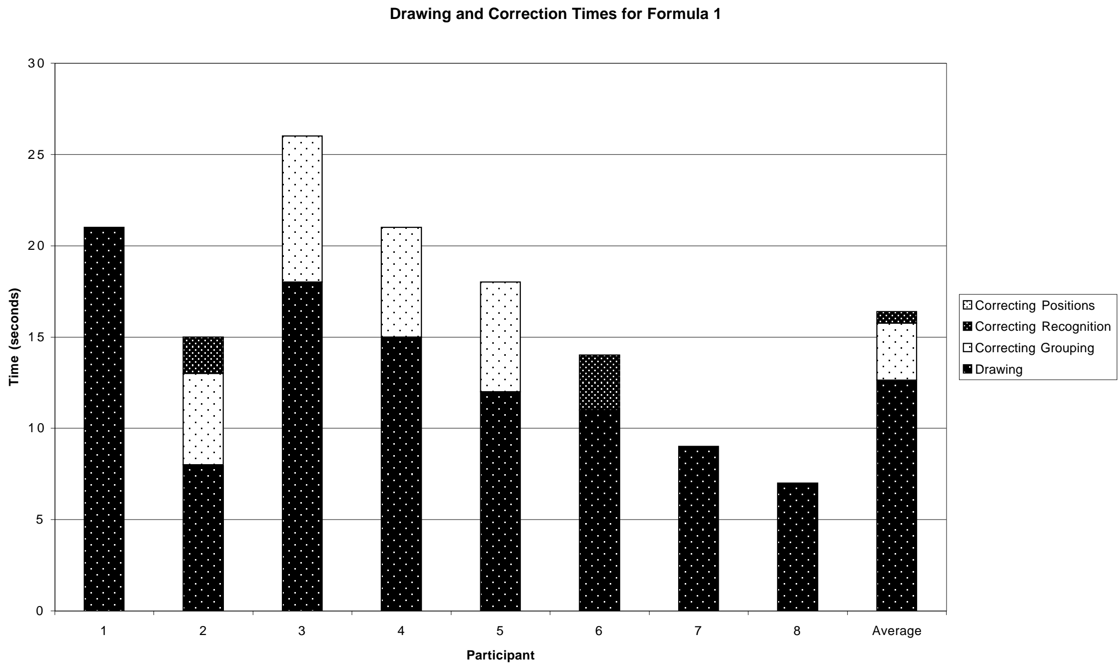


Figure 6.1: Times for people entering Formula 1.

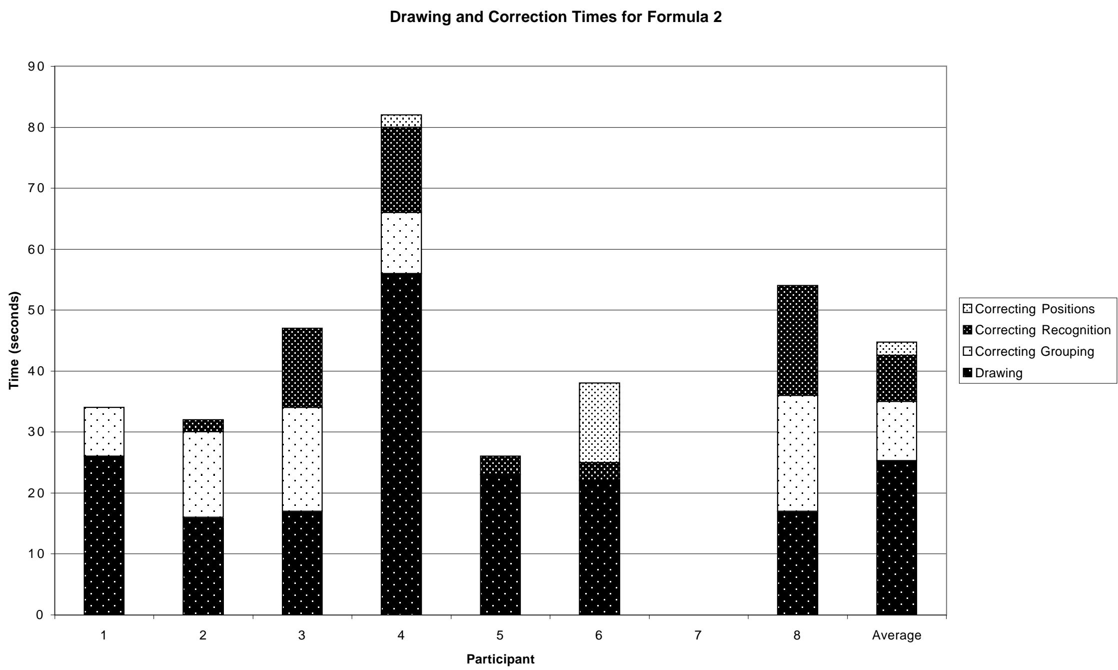


Figure 6.2: Times for people entering Formula 2.

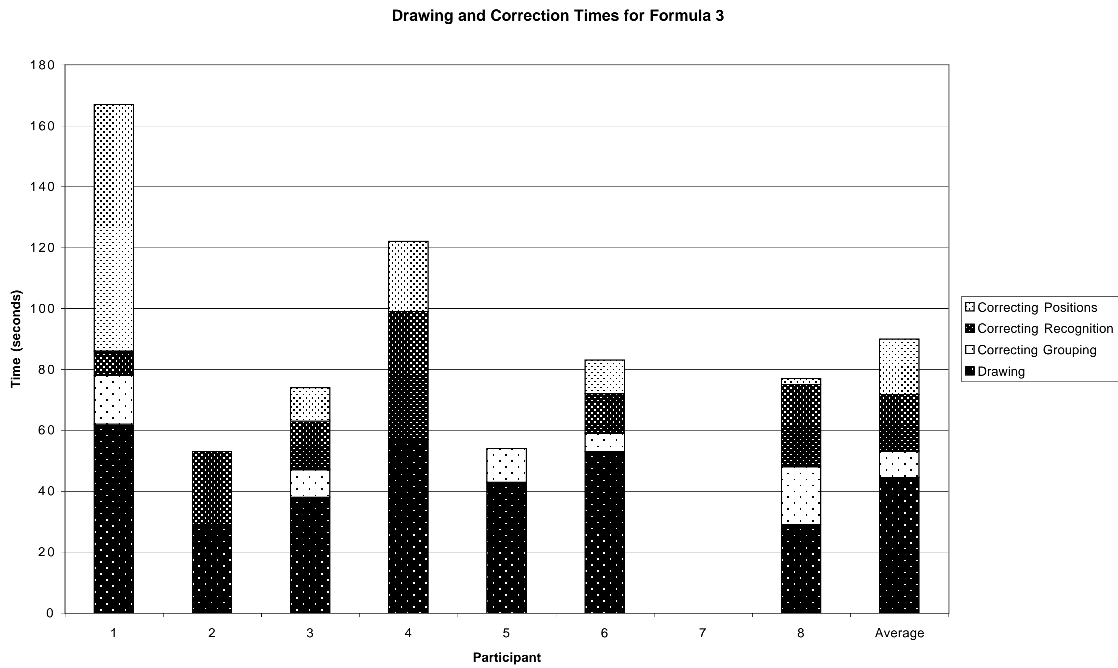


Figure 6.3: Times for people entering Formula 3.

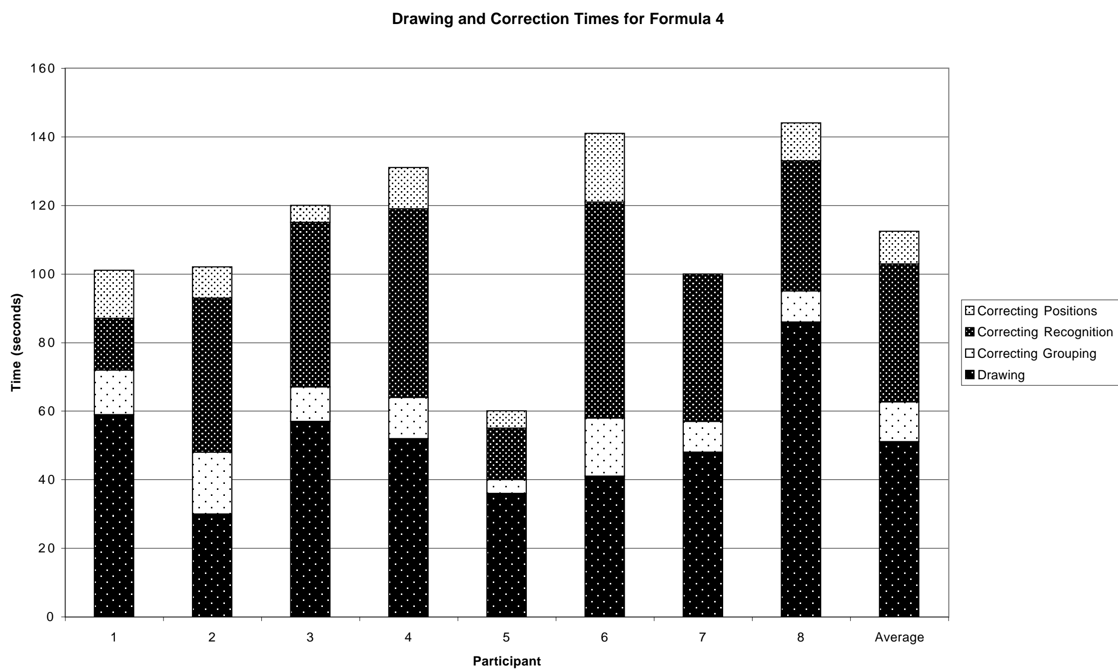


Figure 6.4: Times for people entering Formula 4.

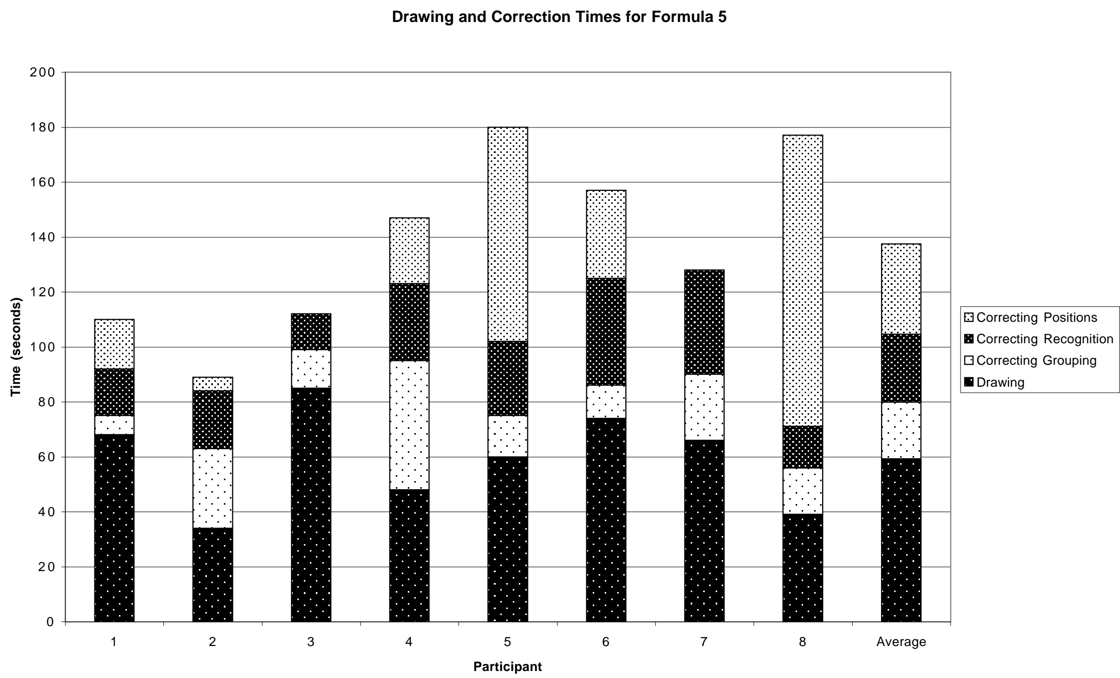


Figure 6.5: Times for people entering Formula 5.

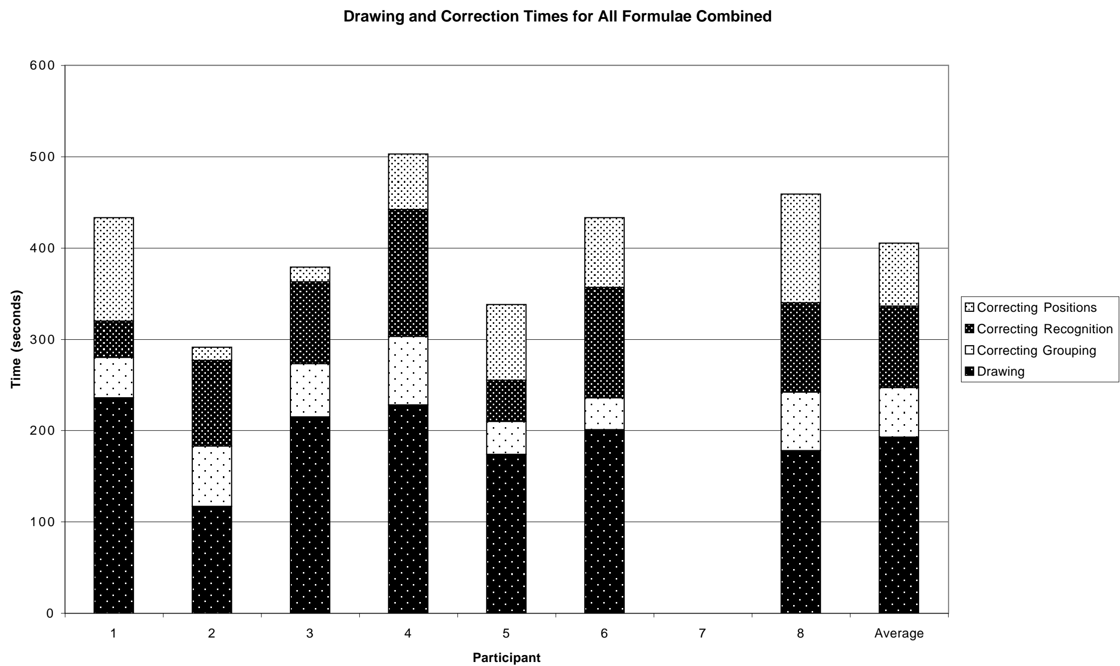


Figure 6.6: Times for people entering all the formulae.

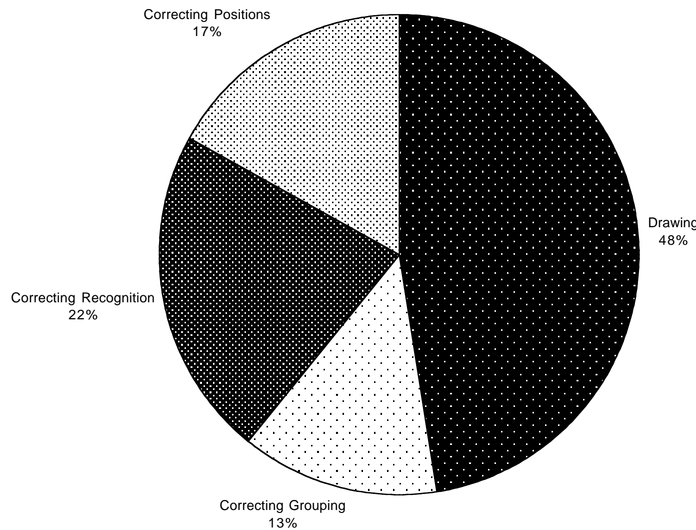


Figure 6.7: Time spent by users entering and correcting formulae.

tentativeness. The users also occasionally chose to delete and rewrite characters instead of using the *modify stroke groups* or *modify character* options for correction when errors occurred.

Due to the fact that the character recogniser had not been trained for each participant and the participants' discomfort with the pen and tablet, the number of misgroupings that occurred was quite high. As a result, a sizable proportion of the user's time was spent making grouping corrections.

Of the total time the users spent entering and correcting each formula, 13% of their time was making grouping corrections. This also includes time that the users spent checking over the formula and looking for grouping errors. Actual grouping corrections took about a second each.

As with the automatic stroke grouping, the low performance of the character recogniser increased the number of symbol misrecognitions. Of the total time spent entering and correcting formulae, 22% of the users' time was spent looking for and correcting character recognition errors. If the correct alternative was on the pop up menu, the correction took less than a second. If the user had to type the correct character in from the keyboard, it took five to ten seconds.

6.1.3 Parsing

When a formula is unparsable or the parser incorrectly interprets the formula, the user has to spend time rearranging the formula so that it is interpreted correctly. The third and fifth formulae, those which were integrals with limits, caused the most difficulty. Of the time users spent entering and correcting their formulae, 17% was spent making changes necessary to get the parser to interpret their formulae as desired.

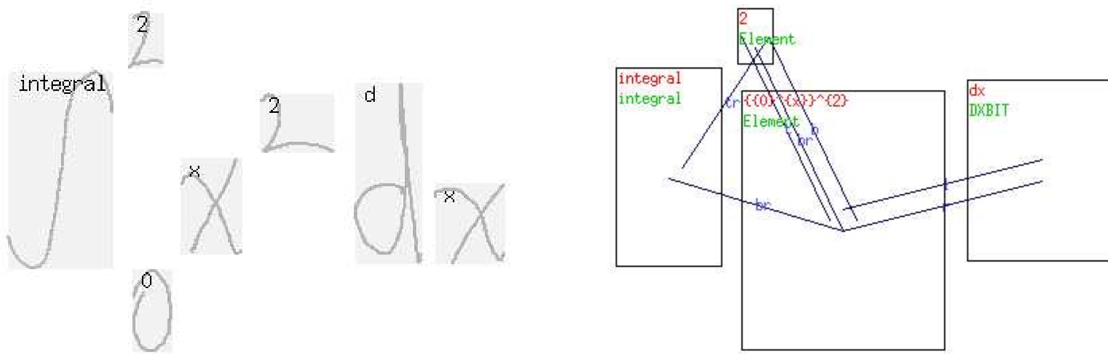
As the parser was slow, a large proportion of the total time using the system, 41%, was spent by the user waiting for the parser to attempt to parse their formula. This time waiting for the parser also includes waiting for the system to generate a preview of their formula from the LaTeX string generated. Preview generation typically takes 8 to 10 seconds per formula.

While the system was parsing the formulae, some participants took the opportunity to talk with the observer. Because they were talking, they did not notice the time that the system was taking to parse their formula and were less inclined to get impatient with the system. In one case a user waited two minutes before cancelling a parsing, but was talking at the same time. If they had not been talking, the user probably would probably have cancelled the formula processing sooner.

In the worst case, the amount of time taken to parse a formula is dependent on the size of the formula, the number of rules in the grammar, and the size of the individual rules within the grammar. The reason for this is that the search used by the system to match rules in the grammar with the current formula graph was a brute force approach that had very little optimisation. For example, when searching for a match of an integral rule, it is searching for a five-node subgraph within the main graph. The time taken is proportional to $C_g^n = \frac{n!}{(n-g)!g!}$ where g is the number of nodes in the graph, and n is the number of nodes in the rule.

The most frustrating problem, from the user's point of view, is that the system spends a lot of time trying to complete a parsing after applying an incorrect rule early on. The system does eventually backtrack and correct itself, but can take several minutes to do so. Figure 6.8 illustrates this. The lower limit on the integral has been interpreted with the integrand as an exponent. Figure 6.8(a) is the formula entered by the user, and Figure 6.8(b) is the state of the graph part-way through the parsing process. The parser has initially taken the integral's limit 0 with the x^2 as a power, giving 0^{x^2}.

The parser then collapses the integration term $\int 0^{x^2} dx$, and finds the upper limit 2 is left over. The leftover symbol then causes the parser to backtrack and try an



(a) Input Formula. (b) Misparsing in progress.

Figure 6.8: Part-way through parsing a formula, with an erroneous decision having been made by the parser.

alternative interpretation of the formula.

While this is a non-catastrophic failure, it can take a very long time for the parser to eventually backtrack and correct this mistake, particularly if it is a large formula or this mistake happens early on in the parsing process.

Another problem was the accuracy of the parser, and its ability to interpret formulae correctly. For the formulae listed in Section 5.4, users took an average of one attempt to get formulae one to three to parse correctly. An average of four attempts were necessary for the more complex formulae four and five. Between parsing attempts, the user would make modifications to the positions of symbols in the formula.

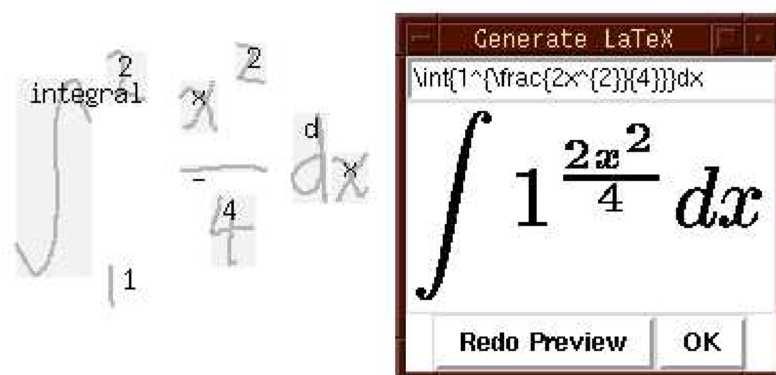


Figure 6.9: A misinterpreted formula.

At times formulae will parse, but the formula returned may not be the most plau-

sible interpretation of a formula. The formula in Figure 6.9 shows an example of this. The integral's upper limit of 2 is multiplied by the x^2, as it is approximately beside the x, and the entire fraction is raised as a power of the integral's lower limit of 1. The error is not detected by the parser as there are no left-over symbols.

The current, non-ideal, solution that the user must learn is to move the limits sufficiently far away from the integrand, so that the parser does not mistakenly incorporate them with the integrand.

Apart from the problem of the erroneous grouping of symbols, due to the lack of contextual tests and overly tolerant geometric layout tests, it handled the positional variations of handwritten input well. These problems only occurred when many symbols appeared close together, so if a user is aware they can avoid these problems with relative ease. In the user testing, it was eventually able to parse all the participants' handwritten formulae.

For the system to be of use to a mathematician, the range of formulae that the system accepts needs to be increased by adding to the underlying grammar. With the current implementation of the graph rewriting formula processor, this will increase processing times even more.

6.1.4 Comparative Timing Results

The times for relatively experienced users entering the five formulae into LaTeX, Microsoft's Equation Editor (MSEE) and the Freehand Formula Entry System (FFES) are presented in the following table. Times are measured in seconds. For comparison, times for the user testing's participants are shown as well, indicative of a novice user's speed.

Formula	LaTeX Expert	MSEE Expert	FFES Expert	FFES Novice	FFES Best Novice
1	3	5	7	16	7
2	6	11	10	45	26
3	14	23	22	86	53
4	14	18	29	112	60
5	23	25	34	139	89
Total time	60	82	102	399	235

The times shown for the system (FFES) is the raw entry plus the time taken for corrections of the occasional grouping and recognition errors. Because of the poor

performance of the formula parser, the time for parsing the formulae is not included. It would unfairly penalise thesystem due to the fact that the formula parser could be replaced with one that was faster. These times are intended to indicate the *entry* times for formulae.

It is important to note that the novice users of the system were not attempting to achieve fast entry times. This data reflects the time of unhurried formula entry, and is included here as a rough upper bound. The "Best Novice" times are the individual best novice times for each formula, selected from all the eight participants in the user testing.

For equations that were near-linear in structure, entering straight LaTeX or using a template-style equation editor, such as Microsoft's Equation Editor (Microsoft, 1993), proved to be faster than using the system. For more complex equations that needed to be "laid out" in 2D, entry time for a user of this system was slower, yet comparable, to that of a relatively experienced user of more conventional systems. In comparison to Microsoft's Equation Editor, users of the system found the entry of formulae to be easier and much less frustrating, particularly if they were doing editing operations.

The novices' average times are much higher than those of the experts, primarily due to low character recognition, averaging 77%. Their unfamiliarity and tentativeness with the pen and tablet interface, as well as not having trained the character recogniser, are the main reasons for this. The two novice users who had prior experience with a pen and tablet performed much better than the averages suggest.

6.1.5 Error Rates

This section presents the success that participants had entering and parsing their formulae. These results were calculated based on information gathered by reviewing the video tapes of their sessions with the system.

Comparison between the error rates for all formulae and the error rates for the final formula was done, to see if there was any improvement as the user became more familiar with the system. There was no noticeable difference at all. This is probably due to the fact that the initial learning curve had been overcome during the training phase before the testing began. Any further improvement would probably be over a longer time scale, such as usage over several hours or days.

Misgrouped Characters

Figure 6.10 shows the misgrouping rates that occurred for each participant in the user testing. As a percentage of the total number of characters written by the users, there was an overall misgrouping rate of 13%.

If the strokes of two symbols are misgrouped as belonging to one symbol, it is counted as one misgrouping error because it takes one regroup stroke in *modify stroke groups* mode to correct it. In other words, the number of regrouping strokes made were counted.

Misrecognised Characters

The number of misrecognised characters was the number of symbols that had not been recognised correctly once the stroke grouping was correct, whether it had been done by the automatic stroke grouper or corrected manually by the user.

Figure 6.10 shows the misrecognition rates that occurred for each participant in the user testing. An overall recognition rate of 77% was achieved for the 583 characters entered by all the test users. Such poor character recognition significantly lowers the usability of the system. For a serious user of the system, taking the time to train the character recogniser is necessary. Once properly trained, the character recogniser offers a recognition rate of over 95%.

Parsing Errors

Depending on the complexity of the formula the average number of rearranging steps before a formula was correctly parsed varied between zero (ideal), and eight.

Figure 6.11 shows the number of parsing attempts for each formula by each participant in the user testing. If the number of parsing attempts is one, then the formula parsed correctly the first time and no rearranging steps were required. As the average number of parsing attempts across all the formulae was 2.3, the overall average number of rearranging steps 1.3.

6.1.6 Evaluation of User Interface Features

This section evaluates specific user features: annotation of symbols as the user writes, the ease of learning and using the *modify stroke groups* and *modify character* correction modes and the ease with which they were able to correct parsing errors.

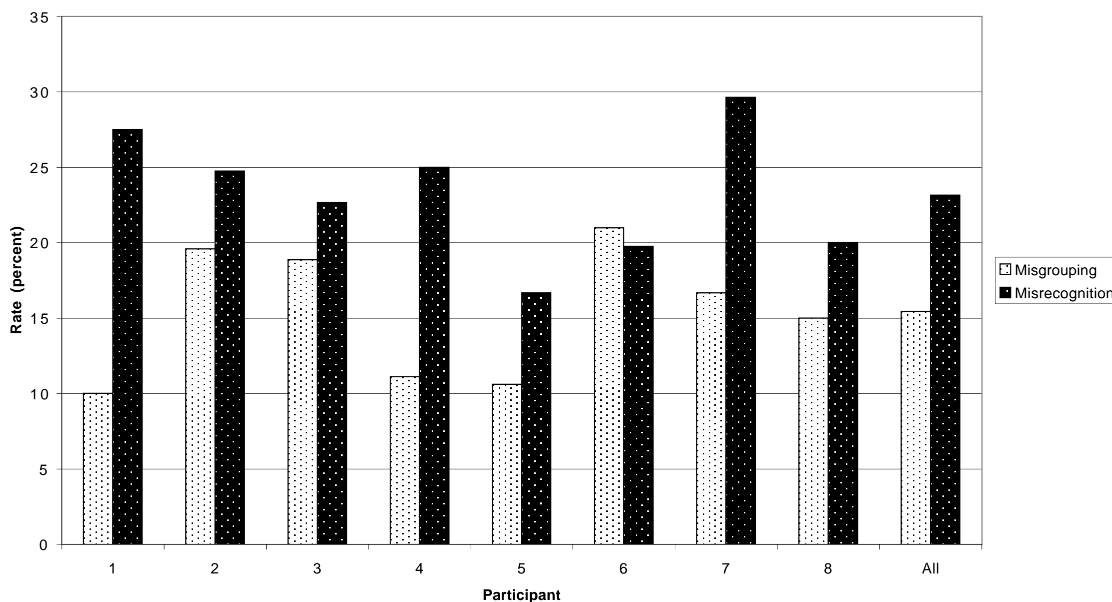


Figure 6.10: Misrecognition and misgrouping rates.

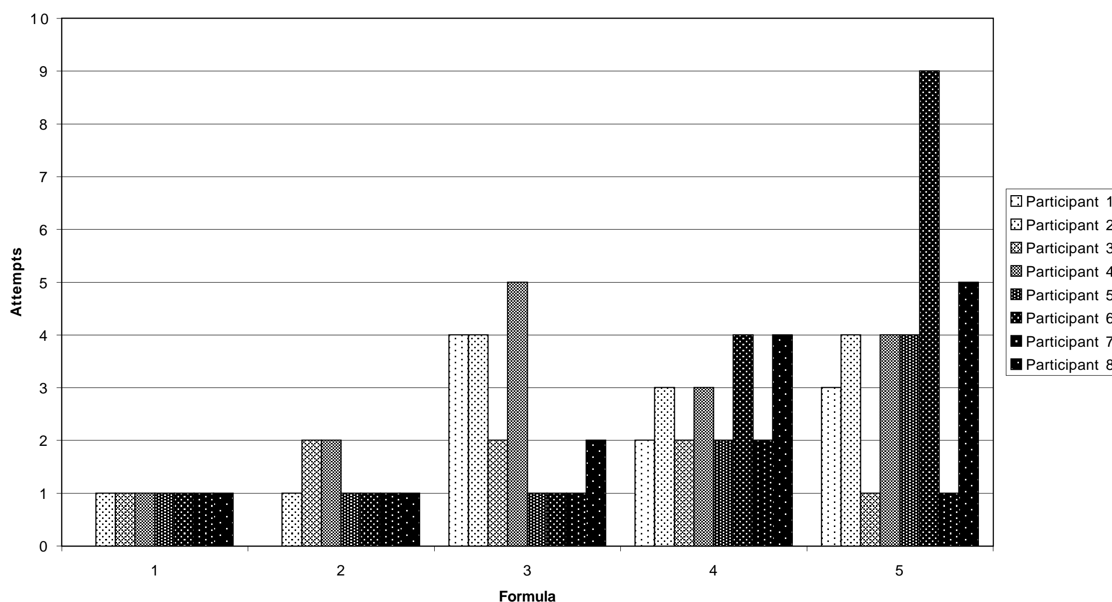
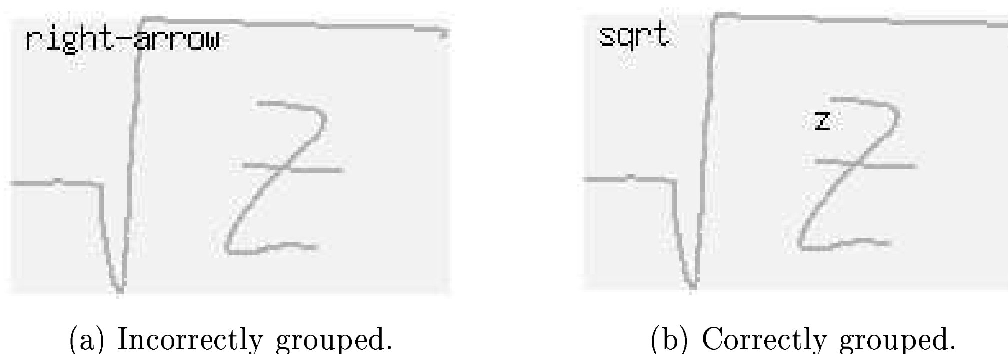


Figure 6.11: Parsing attempts for each formula.

Automatic Stroke Grouping and Annotation as You Write

The shading of the bounding boxes was useful as it provided immediate feedback to the user about the progress of the system's interpretation of their strokes. One alternative approach is to carry out the recognition of the characters once the formula is completely entered, rather than progressively. This could be easily implemented as a user selectable option, however none of the users who used the system felt that the way that the system shaded over and annotated the symbols that they had written was distracting.



(a) Incorrectly grouped. (b) Correctly grouped.

Figure 6.12: The overlapping bounding boxes of the square-root symbol and the z are indistinguishable, making it hard to tell at glance if they are correctly grouped.

One change that needs to be made to the display of the shaded bounding boxes is that overlapping boxes must be distinguishable from each other, perhaps by darkening them where they overlap. This is important for formulae such as $\sqrt{z}$ where the z is inside the square-root symbol. Figure 6.12 illustrates this. The symbols in Figure 6.12(a) have not been correctly grouped, while those in Figure 6.12(b) have. The incorrect grouping is not obvious at a glance.

More obvious bounding boxes are also required for when the user has accidentally drawn small dots and strokes. It has to be easy for the user to see that they have drawn them, and easy for them to be selected and deleted.

Modify Stroke Group Mode

The technique of using the squiggle select to combine a group of strokes into a single character was easily learnt and understood. After being shown the technique once, all the users understood and used it without any problems.

From the user testing it was found that users are not bothered by occasional regrouping steps. This is possibly because grouping errors are easy to detect, in part because of the bounding boxes drawn around strokes that are grouped, and correcting them requires very little effort. Several new users of the system even commented that the method of regrouping strokes was fun to use.

Modify Character Mode

The technique for correcting misrecognition errors, having a pop up menu of the top alternatives provided by the character recogniser and an option to enter the character from the keyboard, worked well. As long as the user did not have to resort to manually entering the symbol's name from the keyboard, the use of the pop up menu enabled quick and easy correction of recognition errors.

Users commented that the pop up menu often did not offer the alternative that the user required, forcing them to manually enter it from the keyboard. It was particularly frustrating for some users to not have an "obvious" alternative offered on the menu, but see it filled with a list of seemingly inexplicable alternatives. For example, the top alternatives to a lower-case b often include 4, h, y, and 7. Of these, only the h seems "reasonable". The features used by the character recogniser to gauge the similarity of characters clearly does not correspond to the user's perceptions.

The use of a dialog box to manually enter a symbol, or choose from a toolbar of non-keyboard symbols, worked well. An improvement would be to have buttons available for all symbols, so that the user did not have to use the keyboard at all if they chose not to.

Parsing

Users often had trouble getting their formulae parsed, and spent a large proportion of their time rearranging terms in their formulae.

When the system was unable to parse a formula, the system tried to indicate to the user the "best" parse it had achieved to that point. The "best" parse was the reduction found by the parser which had the smallest number of nodes. Unfortunately, this did not always correspond to anything very intuitive, and usually was not very useful at all. The number of users who actually saw the parser terminate with this display was very low due to the fact that when given an erroneous formula, the parsing took so long that the user elected to cancel the parsing instead of waiting an indeterminable period of time for the result.

As the system processes a user's formula, it shows a display of the current graph that is being worked with by the parser. From this display an experienced user who has some understanding of the system is able to see what is going wrong and correct the formula appropriately. This display, however, was not helpful for a novice user. The system's ability to parse formulae must be improved if this system is to be useful, and feedback when parsing fails must also be provided. Presently, to correct an unparsable formula, a user is likely to get stuck in a lengthy "formula debugging" loop, tweaking their formula without any real knowledge or direction, until it works.

A useful display for failed formula processing is something that the system currently lacks, and definitely needs to be included. The system should guide the user in some way when it is unable to parse their formula. Perhaps the system could indicate to the user what went wrong, for example indicating missing limits on integrals. However, to automatically provide guidance in correcting formulae may turn out to be impossible. This is discussed further in Section 6.2.8.

6.1.7 Evaluation of the User Testing

This section evaluates the user testing itself, how effective it was, and the shortcomings it had.

The primary goal of the user testing was to determine if a pen-based formula entry system was something that people would want to use, to test the new interface concepts of the squiggle-select for *modify stroke groups* mode and the pop up menu for *modify character* mode and to test how well the implementation of a graph rewriting formula parser worked on handwritten input.

With respect to finding out information on these things, the user testing worked very well.

The second goal of the user testing was to get statistical measures on the performance of the character recogniser and automatic stroke grouping algorithm. This information was easily gained by reviewing the video tapes, though, for any future work, it would be easier if the the system was modified to gather this information automatically instead. For every hour of video tape, it took almost two hours to review it and gather all the information desired.

There are a number of aspects to the user testing that made it less than ideal:

- The observer was myself, the person who wrote the system. Ideally, the observer should have no connection with the system.

- The users were allowed to talk freely with the observer. The observer should really be just an observer, and only a last-resort source of help.
- The full thinking aloud method was not used. As a result, every thought users had as they used the system was not available.
- The entire testing was designed, conducted and analysed by a single person: myself. Having more people involved throughout the entire user testing process would have probably resulted in a better test design, and the ability to easily gather and process more data.
- The participants who took part in the testing represented the potential final users well, though there was no representation of a final "home user".
- The number of participants in the user testing, while giving a good base for opinions, didn't supply enough data for good statistical measures of things such as error rates.

While these are important issues, they did not seem to have a significant impact on this user testing. The goal was to get general opinions of the system, and this has been achieved.

While Nielsen (1994) and Redish (1993) state that nine users is sufficient for user testing, increasing the number of participants in the user testing would have been useful for statistical purposes.

From my experience of user testing this system, after about the sixth participant in the user testing, there was a large proportion of comments that were repeats of existing ones.

The results gained from the user testing were very useful: a number of participants gave valuable feedback as to what the system would have to be able to do before it would be of use to them, or offered some very good ideas as to how the user interface could be improved. The user testing found the flaws that the system had: the weakness of the character recogniser when being used by users that it had not been trained for, and the issues that the formula processor had with processing handwritten input.

6.2 Usability Inspection

This section is an informal usability inspection, conducted by myself. The system is evaluated against the ten usability guidelines proposed by Nielsen (1994) and repro-

duced in Section 4.1.

6.2.1 Visibility of System Status

There is good visibility of system status. At the bottom of the main screen is a status bar which indicates the current mode of operation, such as *draw mode* or *modify stroke groups mode*. As the system is parsing formulae, it is possible to see the parser's current formula graph. This enables the user to see that it is working, and actually doing something.

When the user is entering symbols, the strokes are drawn in one colour. As they are completed and are waiting to be recognised, their colour changes. After recognition, their colour changes again and their bounding boxes are drawn. This display enables the user to know at all times what is happening with their strokes. When drawing in *modify stroke groups*, the line temporarily marked out by the pen is of a different colour to that drawn when entering symbols.

As the user changes between *draw*, *modify stroke groups*, *modify character*, and *select and move* modes, the mouse pointer changes to indicate the mode.

6.2.2 Match Between the System and the Real World

As the system is designed to emulate a pen-and-paper style of interaction, there is a good match with the real world. From comments made by participants in the user testing, the choice of this pen-and-paper metaphor was well received.

The action of scribbling over strokes in *modify stroke groups* is similar to scribbling over a mistake. Similarities like these can cause confusion because the result of the action is not what one intuitively expects for the metaphor of pen-and-paper. Of the people who used the system, none of them found this confusing.

6.2.3 User Control and Freedom

There is a good degree of user control and freedom, though there are small areas which need improvement. The system supports multiple levels of undo and redo, and the parsing process is interruptible with a "cancel processing" button. The separate stage of generating the LaTeX preview is not interruptible though.

6.2.4 Consistency and Standards

In comparison to Microsoft Windows or Apple Macintosh based software, there are few suggested standards for X-Windows based software, unless you are developing for the KDE (1999) or GNOME (1999) desktop environments. So, where appropriate, the guidelines for Microsoft Windows applications were followed. This included items such as hot-keys and menu layouts. In this sense, the interface behaves in a "standard" manner.

6.2.5 Recognition Rather Than Recall

There is very little memory load placed on the user, as all operations are either immediately visible on the screen, or on the first level of menus. Quick and easy access to all commonly used operations, mode changing and generating LaTeX, are available through a toolbar at the top of the window. All other operations are available through the system's menus.

6.2.6 Flexibility and Efficiency of Use

The system offers good efficiency of use through all commonly used operations being available through the use of "standard" accelerator keys, displayed adjacent to their menu items. For example, operations such as cut, undo and mode changing have standard shortcut keys.

6.2.7 Aesthetic and Minimalist Design

The system's design is good, though the colour scheme needs some improvement to increase the degree of contrast between the strokes of a recognised symbol, its bounding boxes and label.

The system's main screen, where the user enters their formula, is relatively uncluttered. Most of the area is the drawing area were the user enters their formulae.

There are relatively few dialogue boxes in the system. The options screen, primarily for making adjustments for debugging purposes, is not suitable for a final product. There is a list of twenty items from colour scheme to formula processor settings.

The other main dialogue box encountered by users is the LaTeX preview window. This is very simple. The window displays the formula generated from the LaTeX generated for their formula, and has a text entry area at the top which the user can copy

and paste the LaTeX code from. The user can also edit this code and regenerate the preview.

The only other dialogues in the system are standard Tcl/Tk load and save dialogue boxes.

The shading and annotation that the system does of the symbols entered by the user adds a degree of extra complexity to the display, though it is not overly obtrusive.

6.2.8 Help Users Recognise, Diagnose, and Recover From Errors

This is a weak point of the system. When the parsing fails the system informs the user that it failed then attempts to show the best parsing of the formula. However, there is no help offered to suggest what the problem could be, though to automatically determine what is wrong with a user's formula could be impossible. The best that could be done would be to determine common errors, such as mismatched brackets, and provide suggestions in these cases.

Errors in recognition and stroke grouping are provided through the use of annotated bounding boxes. These worked well, though occasionally participants in the user testing did not notice that characters were incorrectly recognised.

Simple recovery from recognition and stroke grouping errors is available through the *modify stroke groups* and *modify character* modes. Recovery from parsing errors is possible through the *select and move* mode that allows users to rearrange their formulae, though guidance as to what needed to be done to recover from parsing errors does not exist.

6.2.9 Error Prevention

It is not possible to prevent all errors in stroke grouping, character recognition and formula parsing; even humans occasionally misinterpret handwritten equations. Thus, the provision of easy means of correction is probably the best that can be achieved. In these terms, the system is satisfactory.

6.2.10 Help and Documentation

The system offers no help or documentation, primarily due to its status of being solely a prototype of a pen-based formula entry system and a test of a graph rewriting formula

parser on handwritten input.

6.2.11 Overall Degree of Usability

The overall degree of usability of this system, with respect to these guidelines, is good. As this system is designed as a researcher's test-bed, not a commercial application, some of the issues raised by this evaluation are not so important. If this system was to be released commercially, these issues would have to be addressed, and a proper evaluation would have to be carried out by an impartial user interface expert.

6.3 The Overall System

This section discusses the positive, negative and overall impressions of the system from a user's point of view.

6.3.1 Positive

All users of the system find it easy to learn and use. The method of entry (pen and tablet) is easily picked up by first time users, the method for correcting grouping (squiggle-select), recognition (pop up menu) and the method for rearranging the formula is also easily learnt, understood, and used.

The use of a writing tablet, instead of keyboard or mouse, as the preferred entry method is seen by users as a positive aspect. In spite of the fact that entry using this system took longer than entry with more traditional tools, such as Microsoft's Equation Editor or LaTeX, it was still seen as a preferable method of entry. The pen and tablet is a much more natural way to enter formulae and the interface allows for unrestricted editing of formulae already entered. It is just as easy to make minor or major changes to the contents and structure of the formula. However, there is the issue that the number of home users that own a pen and tablet is small. Writing with a mouse can be difficult and frustrating.

Another strong point of a handwriting based formula entry system is that it does not require any specialised knowledge to use it. The user can just pick up the pen and draw the formula desired on the tablet. Systems such as LaTeX require the user to learn a language for the description of mathematical formulae. This system allows users to write formulae in the most natural way possible.

6.3.2 Negative

There were only two major problems with the system found by participants in the user testing. First was the implementation of the graph rewriting parser, its lack of speed and errors when processing formulae. Second were the incidental problems related to the character recogniser not being trained for each user's writing, which resulted in high misgrouping and misrecognition rates.

Both of these problems can be addressed; the first through improvement of the graph rewriting parser, the second through either improving the character recognition module, or transparently training the character recogniser online using the corrections that the user supplies through the *modify character* mode. Other problems found with the user interface were minor in nature and typically cosmetic.

The system is going to need to be able to handle more complex formulae before it will be of use to a mathematician. For example, a formula such as one that Lavirotte and Pottier's system (Lavirotte and Pottier, 1997) can parse:

$$(\frac{1}{x^2+1})^{(n)} = (-1)^n.n!\frac{\sum_{0\leq 2p\leq n}(-1)^p C_{n+1}^{2p+1} X^{(n-2p)}}{(X^2+1)^{(n+1)}}$$

is unable to be handled by this system, due to the time required by the parser being excessive. Thus, improving the parser to take advantage of contextual information and to use optimised graph searching techniques is important, if further work is to be done on this system.

6.3.3 Overall

Based on the user testing and usability studies of this prototype system, it can be concluded that an application for good, easy to use, pen-based formula entry can be built. The system worked very well and was easy to learn and use. The system had several flaws, but all these can be remedied. Creating a handwriting-based formula editor is possible and also, more importantly, desirable in comparison to existing systems such as template or command string based equation entry systems.

Chapter 7

Future Work

From the user testing, a large number of suggestions were made as to how the system could be improved to make it easier and more pleasant to use. Also, a number of issues were raised with respect to the user interface and style of interaction. Some of the more interesting and important issues are presented and discussed here.

7.1 The Formula Parser

The formula parser sometimes was not able to parse a user's formula quickly, or took a long time to determine that a formula was in error. Ideally, parsing should take no more than ten or twenty seconds, but it can take many minutes. Also, there were interpretation problems that increased the parsing time, or returned erroneous interpretations of formulae.

As the parser makes the initial assumption that the first matching rule it finds in the grammar is the correct one to apply, well formed formulae parse well. It is only when it has to backtrack and generate all the children of a node that it takes a significant amount of time. The backtracking occurs either when there is an error in the formula and it has to exhaust all possible paths of derivation before it determines it is unparsable or, more commonly, the parser makes a mistake in its initial interpretation.

To reduce the chance of the parser incorrectly parsing formulae, the formula parser can be made more selective about the relationships between items, making the regions accepted by the geometric tests, as described in Section 3.1.4, smaller. Thus, the x^2 in Figure 6.9 would be too far from the 2 to be combined. Unfortunately, doing this also limits the parser's ability to cope with the unpredictable nature of handwritten input. It also does not eliminate the problem, it just reduces it. Another alternative is

to change the shape of the areas in which symbols are looked for, from the overlapping rectangular regions that are currently being used to something based more on a more empirical or intelligent study of writing. It might also help to have regions defined on a per-operator basis. In this way, the region for an upper limit on an integral would be differently shaped to the region for an exponent.

The use of context rules, as done by Lavirotte and Pottier (1997) and mentioned in Section 2.3.6, would provide the greatest improvement of the system's performance by reducing the frequency of interpretation errors and increasing the parser's speed. The contextual rules that Lavirotte and Pottier add to the grammar allow such things as reserving the limits for the integral. The additional computation required for the contextual checks would most likely be far outweighed by their gains. It is also possible that using some sort of stochastic grammar would help to rule out the unlikely interpretations returned for some formulae, in favour of more likely ones.

The speed of the parsing process can be improved, from the current $O(n!)$ time complexity, by using advanced graph matching techniques, such as those described by Bunke and Messmer (1997). If $L =$ the number of rules in the grammar, $g =$ the number of nodes in the current formula's graph, and $n =$ total number of nodes in a rule in the grammar, then the speed of their approach varies between $O(L.g.n)$ for best case: when all the rule-graphs have the same structure, and $O(L.g^n.n^2)$ for the worst case: where none of the rule-graphs share any common structure.

The current speed of the parser, even when it works well, is still not fast enough for real time processing of formulae. The formula processing will always have to be an additional step that the user waits for. For real time parsing, the parsing would have to be completed within a second of the user writing a symbol, otherwise it runs the risk of seeming too slow. A sacrifice in speed is the tradeoff for having a system which allows completely arbitrary input order. Systems such as Littin's (1995), that use a modified SLR(1) parser to offer real-time parsing of formulae, restrict the order of entry of symbols in a formula.

7.2 Keyboard Input

Most people can learn to type much faster than they are able to write. As commented by Kajler and Soiffer (1998), and supported by Brown's study (Brown, 1988), keyboards remain the most efficient device for purely textual data input, even for self taught typists. As a result, some formulae will always be easier to enter with a keyboard than

with a pen.

An important facility that should be added to an interface such as this is the provision for the user being able to type formulae in directly, while still allowing them to use pen based input for the things that are difficult to express with keyboard commands, such as integrals and fractions.

With the option to use the keyboard, the interface would then start to regain some of the speed of use that it does not currently have in comparison to LaTeX and Microsoft's Equation Editor, both of which are keyboard oriented.

7.3 Magic Hot-Spots

Currently, if the user wishes to modify the grouping of strokes or correct misrecognition errors, they have to choose an option from a menu or use a toolbar at the top of the screen.

Special hot-spots could be placed on the bounding box for each recognised symbol. These could then be touched and the hot-spot would either react by offering a pop up menu of alternatives, thus giving the functionality of the "modify character" mode, or dropping into "modify stroke group" mode for the next stroke drawn with the pen. However, addition of hot-spots would potentially detract from the minimalist and aesthetic design goal.

7.4 Indication of Areas

On-the-fly indication of where the user could place limits and powers might be advantageous. Perhaps if the user moved the pointer over a symbol, the system would then indicate where the operands for that symbol should be written. Unfortunately, this would interfere with the user's ability to enter formulae naturally, and without having to worrying about the placement of symbols. This also runs the risk of cluttering the display.

7.5 Training of the Character Recogniser

As the user is supplying corrections for misrecognised characters, it would be possible to also use this information to progressively train the character recogniser as the user is using the system. The system would progressively learn to recognise the user's

writing over a period of time. The only danger is that the user might not supply the correct interpretation for a symbol, and end up confusing the recogniser further. However, the system may be able to determine if this is happening when there are significant differences between the symbol supplied and all the other training examples the character recogniser has.

It takes the current character recogniser 4.0 seconds to train on 451 examples of 180 symbols. As each symbol is trained independently of the others, it takes about 0.02 seconds to train it for a single symbol. Due to this high training speed, online training is possible.

Even if the character recogniser took longer to train, the training could be executed in the background and the character recogniser's data could be updated every few seconds.

7.6 Squiggle Select for Other Selecting Operations

"Squiggle Select" was very successful for stroke grouping. It could also be used to select objects to move and delete, instead of rectangular regions. This would mean that the accuracy and speed of the squiggle select would be available for selection.

7.7 Morphing of Symbols

One way to deal with the slightly cluttered appearance that the shaded and annotated bounding boxes can give is to morph the symbols written by the user to an ideal font or stroke pattern (Littin, 1995). Littin parses the formula as it is entered, so he is also able to move the symbols on the input area to their correct positions. While this may initially seem appealing as it gives good feedback on what has been entered and causes a cleanly laid out formula to appear in the input window, this method was not used for the following reasons:

- Having the system altering what the user has written could be distracting. It is possible that, as the user writes their formula, the movement of symbols as they are morphed may distract or annoy the user. It has the potential to annoy the user if the morphing happens near where they are writing.

- Visual stroke information is lost. If the system misgroups the strokes and then those strokes are consumed when they are morphed into the ideally formed char-

acter, it would require an extra step to unmorph the character before the strokes could be regrouped.

- If the system rearranges the formula, it may not leave enough room for the user to go back and modify the formula later. Because Littin's system only allowed entry around the last few symbols drawn, he left some space around them but closed everything else up. In a system which allows arbitrarily ordered input, the morphing may close up a gap that the user intentionally left to fill in later.

- If the final size of the morphed character is dependent on the original size of what the user wrote, the formula may end up looking like a ransom note due to variations in the sizes of symbols that were written.

- There is nothing wrong with leaving the user's original strokes there anyway. If the goal is to end up with a series of typeset formulae on the input area, perhaps as part of larger a computer algebra system, the morphing could be triggered by a menu selection or pen-based gesture on the user's part.

In spite of this, the morphing of characters is something that could be easily added at a later point. Another alternative is to only morph at the user's request or only once the formula has been successfully parsed.

Chapter 8

Conclusion

This thesis has described and evaluated a system that allows the freehand entry of formulae using a pen and tablet. After entering the formula, it is easy to arbitrarily manipulate it: rearranging, adding or deleting symbols. Once the user has entered the formula to their satisfaction, the system can then generate the corresponding LaTeX command string. This command string can then be copied and pasted into the user's LaTeX document.

In addition to combining an existing character recogniser and formula parser with new user interface ideas, this thesis has made a number of contributions:

- An algorithm for automatically grouping the user's strokes into symbols, using the character recogniser to evaluate different possibilities.
- New user interface concepts for identifying misgrouped and misrecognised symbols using annotated bounding boxes, and lagging the recognition so as not to distract the user.
- A "squiggle select" for correcting stroke grouping errors, allowing the user to draw a line through the strokes of symbols that should belong together.
- Showing that a graph rewriting parser has the potential to work well for handwritten input, although this particular implementation did not.
- Showing that a pen-based user interface for the entry and editing of mathematical formulae is desirable and preferable to existing command-string or template based entry systems.

A pen-based approach provides a more natural and familiar interaction method than existing command-string or template-based equation editors. When entering a formula

using this system, a user does not have to learn a special language or notation; more importantly, they do not need to determine the overall structure of a formula before entering it. With the user not having to spend time searching for special symbols in menus, maneuvering the cursor into boxes in templates, or referring to manuals to find the correct commands for the operations they wish to do, entering and editing a formula is much easier.

This system does not offer a fast alternative to existing formula entry systems, even for expert users, but it is much more intuitive, less complex, and easier to learn. Its strengths are in the entry of large, complex formulae, and the editing of these formulae after entry.

The system is an interface overlaid on modules for handwriting recognition, equation parsing, and typesetting. The performance of the system, in terms of both speed and accuracy, can be improved by improving the performance of these elements. However, since there will always be some ambiguity in handwritten input, simple methods for correcting errors, such as the ones presented in this thesis, will always be necessary. With this in mind, this system offers a pop up menu offering character recognition alternatives, and the means to regroup strokes into symbols is provided with a "squiggle select".

The underlying formula processor uses graph rewriting to process the initial formula and determine what the corresponding LaTeX representation is. This type of parser is able to cope well with the position and size variations inherent in handwritten input. The only problems with the formula processor implementation are its lack of speed and occasional tendency to misparse formulae. The use of context rules and advanced graph searching techniques would address both these issues.

The user testing of the system provided very encouraging feedback. While the entry times for the formulae tested were slower than the corresponding entry times for LaTeX and Microsoft's Equation Editor, all the users who tested the system were very positive about the concept of a pen-based formula entry system, saying that it was preferable to, and much more intuitive to use, than existing command-string based or template-based systems. The new concepts in the user interface also received positive comments.

The user testing also highlighted a number of issues that need to be addressed, and provided a large number of suggestions for future directions that this system could take. Some of the most significant are:

- improving the graph rewriting formula processor's speed and accuracy, perhaps

through the use of context rules or advanced techniques for graph matching.

- improving the feedback to the user when their formula is unparsable, possibly offering an indication of what is wrong with it.
- eliminating the need to enter a special mode, by pressing a button on a toolbar or choosing a menu option, in order to modify the character recognition or stroke groupings. Hot-spots on each symbol's bounding box could be used for this.
- expansion of the set of formulae that the system can understand, so that it would be of more use to mathematicians and physicists. The current set of formulae that the system can process is too small for such users.
- creation of a GUI tool to enable end users to easily add to and change the grammar used by the formula processor. This would let users add their own notations and customise the system to understand their mathematical layout style.
- introduction of capabilities for mathematical manipulation, perhaps initiated through gestures. Operations such as the evaluation, expansion, or factorisation of formulae could be accomplished through a link to an external mathematics package, such as Mathematica. However, if this interface was to be expanded to include formula processing operations initiated by a gesture, there are usually no intuitive gestures available for a user. Kajler and Soiffer (Kajler and Soiffer, 1998) warn that a system can be hard to learn and confusing for a user who is faced with a large number of arbitrary gestures that need to be learnt. This is a problem common to all gesture based interfaces.

Ultimately, the creation of a fully featured pen-based formula entry system should be possible, ideally as a front end to symbolic manipulation or mathematics packages. A virtual piece of paper that can not only record writing, but also interpret it on demand and allow gesture-based algebraic manipulation could one day be created. This prototype system is a step in this direction.

References

Anderson, R. H. (1968). Syntax-directed recognition of hand-printed two-dimensional mathematics, *in* M. Klerer and J. Reinfelds (eds), *Interactive Systems for Experimental Applied Mathematics*, Academic Press, New York, pp. 436–459.

Anderson, R. H. (1971). A comment on the recognition of hand-printed two-dimensional mathematical expressions, *Proceedings of Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, California, pp. 100–101.

Apple Computer, Inc. (1987). *Human Interface Guidelines: The Apple Desktop Interface*, Addison-Wesley.

Avitzur, R. (1992). Your own handprinting recognition engine, *Dr. Dobb's Journal* **17**(4): 32–37.

Bernstein, M. I. (1971). Computer input/output of two-dimensional notations, *Proceedings of Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, California, pp. 102–103.

Blostein, D. (1996). General diagram-recognition methodologies, *Graphics Recognition: Methods and Applications*, Vol. 1072 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 106–122.

Blostein, D. and Grbavec, A. (1996). *Recognition of Mathematical Notation*, World Scientific Publishing Company, chapter 22.

Brown, C. M. (1988). Comparison of typing and handwriting in "two-finger typists", *Proceedings 32nd Annual Meeting of the Human Factors Society*, Santa Monica, California, pp. 381–385.

Bunke, H. (1982). Attributed programmed graph grammars and their application to schematic diagram interpretation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **4**(6): 574–582.

Bunke, H. and Messmer, B. T. (1997). Recent advances in graph matching, *International Journal of Pattern Recognition and Artifical Intelligence* **11**(1): 169–203.

Chou, P. A. (1989). Recognition of equations using a two-dimensional stochastic context-free grammar, *Proceedings SPIE Visual Communications and Image Processing IV* **1199**: 852–863.

Desurvire, H. (1994). *Usability Inspection Methods*, John Wiley & Sons, Inc., chapter 7, pp. 173–202.

Dumas, J. and Redish, J. (1993). *A Practical Guide to Usability Testing*, Ablex Publishing Corporation, Norwood, New Jersey.

Ettrich, M. (1999). Lyx webpages.
URL: *http://www.lyx.org/*

Fateman, R. J., Tokuyasu, T., Berman, B. P. and Mitchell, N. (1996). Optical character recognition and parsing of typeset mathematics, *Journal of Visual Communication and Image Representation* **7**(1): 2–15.

Fateman, R. and Tokuyasu, T. (1996). Progress in recognizing typeset mathematics, *SPIE*, Vol. 2660, pp. 37–50.
URL: *http://http.cs.berkeley.edu/~fateman/ocrpapers.html*

GNOME (1999). Gnome project web pages.
URL: *http://www.gnome.org*

Ha, J., Haralick, R. and Phillips, I. (1995). Recursive X–Y cut using bounding boxes of connected components, *Proceedings of the Third International Conference on Document Analysis and Recognition (ICDAR '95)*, Montreal, Canada, pp. 952–955.

Hayden, M. B. and Lamagna, E. A. (1998). Newton: An interactive environment for exploring mathematics, *Journal of Symbolic Computation* **25**(2): 195–212.

Kajler, N. and Soiffer, N. (1998). A survey of user interfaces for computer algebra systems, *Journal Of Symbolic Computation* **25**(2): 127–159.
URL: *http://www.ensmp.fr/~kajler/bibliography.html*

KDE (1999). The K desktop environment web pages.
URL: *http://www.kde.org*

Lamport, L. (1994). *LaTeX: A Document Preparation System*, Addison Wesley.

Landay, J. A. (1997). User testing course note slides.
URL: *http://bmrc.berkeley.edu/courseware/cs160/spring97/lectures/user-testing/*

Lavirotte, S. and Pottier, L. (1997). Optical formula recognition, *Proceedings 4th International Conference on Document Analysis and Recognition (ICDAR)*, Vol. 1, Ulm, Germany, pp. 357–361.

Lavirotte, S. and Pottier, L. (1998). Mathematical formula recognition using graph grammar, *Proceedings of EI '98 (SPIE Symposium on Electronic Imaging: Science and Technology)* .

Littin, R. (1993). The pen input of mathematical expressions, 4th year honours thesis, University of Waikato.
URL: *http://www.cs.waikato.ac.nz/~rhl/papers/publications.html*

Littin, R. (1995). *Mathematical expression recognition: Parsing pen/tablet input in real-time using LR techniques*, Master's thesis, University of Waikato.

Martin, W. A. (1967). A fast parsing scheme for hand-printed mathematical expressions, AI Memo 145, MIT.

Martin, W. A. (1971). Computer input/output of mathematical expressions, *Proceedings of Second Symposium of Symbolic and Algebraic Manipulation*, Los Angeles, California, pp. 78–89.

Microsoft (1993). *Microsoft Word User's Guide, Version 6.0*, Microsoft Press.

Microsoft (1995). *The Windows Interface Guidelines for Software Design*, Microsoft Press.

Miller, E. G. and Viola, P. A. (1998). Ambiguity and constraint in mathematical expression recognition, *Proceedings of the 15th National Conference of Artificial Intelligence*, American Association of Artificial Intelligence, Madison, Wisconsin, pp. 784–791.
URL: *http://www.ai.mit.edu/people/emiller/OQE_slides/index.htm*

Nielsen, J. (1994). *Usability Inspection Methods*, John Wiley & Sons, Inc., chapter 2, pp. 25–62.

Nielsen, J. and Mack, R. (eds) (1994). *Usability Inspection Methods*, John Wiley & Sons, Inc.

Pottier, L. (1995). Recognition of mathematical formulas, using context sensitive graph grammars.
URL: *http://www.inria.fr/safir/whoswho/Loic/issac95/issac95.html*

Rubine, D. (1991). Specifying gestures by example, *SIGGRAPH '91 Conference Proceedings* **25**(4): 329–337.

Schumacher Jr., R. M. (1992). Ameritech graphical user interface standards and design guidelines.
URL: *http://www.ameritech.com:1080/corporate/testtown/library/standard/std-guix.html*

SGI (1999). Sgi inperson online guide.
URL: *http://www.sgi.com/Products/software/InPerson/*

Shneiderman, B. (1992). *Designing the User Interface : Strategies for effective human-computer interaction*, Addison Wesley.

Smith, S. L. and Mosier, J. N. (1986). Guidelines for designing user interface software, *Technical Report ESD-TR-86-278*, MITRE Corporation, Bedford, Massachusetts.

Smithies, S., Novins, K. and Arvo, J. (1999). A handwriting-based equation editor, *Proceedings of Graphics Interface (GI) 1999*, Kingston, Ontario, Canada.

Srihari, S. (1986). Document image understanding, *Proceedings Fall Joint Computer Conference*, Dallas, Texas.

Turban, E. (1992). *Expert Systems and Applied Artificial Intelligence*, Macmillan Publishing company, chapter 7, pp. 254–256.

van Egmond, S., Heeman, F. C. and van Vliet, J. (1989). INFORM: An interactive syntax-directed formulae editor, *The Journal of Systems and Software* **9**: 169–182.

Wildman, D. (1995). Getting the most from paired-user testing, *ACM Interactions* **2**(3): 21–27.

Wolfram, S. (1996). *The Mathematica Book*, 3rd edn, Wolfram Media/Cambridge University Press.

Yaeger, L. S., Webb, B. J. and Lyon, R. F. (1996). Combining neural networks and context-driven search for online, printed handwriting recognition in the Newton, *AI Magazine* **19**(1): 73–89.

Zhao, Y., Sakurai, T., Sugiura, H. and Torii, T. (1996). A methodology of parsing mathematical notation for mathematical computation, *Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, ACM Press, Zurich, Switzerland, pp. 292–300.

Appendix A

Accompanying CD-ROM

The accompanying CD-ROM contains material related to this thesis: the thesis itself, the source code, a commentated movie of the system being used, and a paper published at Graphics Interface '99 (Smithies et al., 1999).

A.1 Readme Text File

Filename: `readme.txt`

This file includes a description of the items on the CD, similar to what is in this appendix.

A.2 The Thesis

Filename: `thesis.ps`

The thesis itself. Requires a Postscript capable viewer or printer.

A.3 Quicktime Movie

Filename: `ffes.mov`

A movie captured in real time of the system being used, with a commentary on what is happening. To play it, a Quicktime capable movie player is necessary.

The movie shows the features of the system, including:

- pen based input.
- automatic stroke grouping and symbol annotation.

- arbitrary symbol entry order.
- stroke regrouping with the "squiggle select".
- character recognition correction with a pop-up menu.
- selecting and moving parts of formulae.
- parsing of formulae, and the generation of a LaTeX preview.

The movie shows a user entering three formulae to demonstrate these features. In the first example, the user initially enters the formula $x^2 + 4$ and allows the system to recognise the symbols entered. The user then changes the formula to an integral, making it $\int x^2 + 4\ dx$. The user then generates the LaTeX for this formula, and the preview is generated.

The second example shows the correction of recognition errors that the system makes. The user enters the formula $x^2 + y^2 = z^2$, but the system groups the strokes incorrectly, taking the y^2 as a single symbol, and the two strokes of the $=$ as separate symbols. The user is able to fix this by going into *modify stroke groups* mode and drawing a line through the strokes that should belong to a single symbol. A stroke through the y splits it off from the 2 and a stroke through the two bars of the $=$ combine them together. At this point, the y is still misrecognised, so the user enters *modify characters* mode. Clicking on the misrecognised y offers a list of alternatives provided by the character recogniser. If the correct symbol is on this list, the user is able to select it. Otherwise, they choose the *Enter...* option which allows them to enter from the keyboard what the symbol actually is. With the formula correctly entered and recognised, the user is able to generate the LaTeX command string for it.

The system allows basic editing of formulae: moving and deletion of single or multiple symbols. This allows major structural rearrangement of formula once entered. In the third example, the user enters $\int x^2 + 4dx$ then, after generating the LaTeX for it, decides that they want the integrand to be a fraction. Entering *select and move* mode allows the user to select the old integrand of $x^2 + 4$ and move it up so that the formula can be edited to make it $\int \frac{x^2+4}{\sqrt{x}}dx$. The user then decides that they want limits on the integral, so they again use *select and move* mode to make room between the integral sign and the integrand. After adding the limits, the final formula $\int_0^4 \frac{x^2+4}{\sqrt{x}}dx$ is complete, and the user again generates the LaTeX for it.

A.4 Tar File

Filename: `ffes31-3.tar`

This file contains the source for the formula entry system, and the data and code for the character recogniser that my system uses.

The character recogniser is part of a larger system called Smartboard. The Smartboard web pages are located at:

`http://www.cs.caltech.edu/~arvo/projects/SmartBoard/`

The file is a Unix tar file, and can be expanded with the command:

`tar xf ffes31-3.tar`

Some files and directories of note in the tar file are:

`msc/ffes_3_March_1999/*.{cc,h}` is the source code for the application, and number of other programs written for testing purposes.

`msc/ffes_3_March_1999/iface/` contains the Tcl/Tk source code for the user interface.

`msc/ffes_3_March_1999/formulae/` contains a number of data files in the format described in Section 3.1.2.

`msc/ffes_3_March_1999/{not_in.data,not_out.data}` define the limitations on building arcs between nodes, as described in Section 3.1.4.

`msc/ffes_3_March_1999/{preprocess.grammar,restof.grammar}` are the preprocessing and main grammars as currently used by the system.

`msc/ffes_3_March_1999/ffes/` contains a number of data files for formulae that users have entered in, readable by this system.

`msc/sb_data/` contains directories of data used by the character recogniser to recognise symbols that users enter.

`msc/SmartBoard/` contains the Smartboard character recogniser code, and utilities for training the recogniser.

A.5 Graphics Interface '99 Paper

Filename: `gi99_ppr.ps`

This paper was presented at Graphics Interface '99, Kingston, Ontario. It is written by myself, Kevin Novins, and James Arvo (1999). It covers a subset of the material presented in this thesis, focusing mainly on the new user interface techniques developed. Requires a Postscript capable viewer or printer.

Appendix B

Ethical Statement

The following pages contain the ethical statement that was prepared in accordance with the regulations at the University of Otago. Upon approval of this, the user testing was able to proceed. Ethical considerations in user testing are discussed in Section 5.3.

Freehand Formula Entry System

Ethical Statement

Steve Smithies

In accordance with Page 5 of the "Handbook for Masters' Degrees", this is an ethical statement regarding how I intend to carry out a user study on the system I have created as part of my Master's Thesis. This summarises the guidelines and conditions under which I will be performing this user testing, and specifies how the users will be treated.

The system being tested is one for entering formulae into a computer, using a pen and drawing tablet. The study involves asking a number of participants to try out the system, gathering their opinions of it through a questionnaire, and finding any possible flaws in my system.

The data gathered is intended to prove or disprove the claim that my "Freehand Formula Entry System" is better than existing formula entry systems, to gauge in general how good it is, and find any shortcomings that it may have.

- **Use of the data gathered.** The findings and opinions of the users are intended to be reported in my final Masters Thesis, and as part of papers submitted to journals and conferences about the system. There are no ulterior motives for this study, and everything that we are testing will be made clear to the participant.

- **Privacy and anonymity.** Although I am intending to video-tape the participants, screen, tablet and keyboard, their anonymity will be preserved. The video-tapes will be viewed by those involved with evaluating the system (primarily myself), but published details will be totally anonymous and untraceable to the original subject. If the subject strongly objects to being videotaped, I will not do so.

- **Informed consent.** The user will have the testing process that they will be going through explained to them in advance. Participants will be advised that the study involves use of a program on a computer. It will be effectively "normal" computer use, with no additional physical or mental stress applied. Although there will be evaluation of their performance and behaviour, I will be judging the system and not them.

- **Participation.** Participation in this is to be totally voluntary. People will be only doing this of their own free will, and are able to stop at *any* time.

Student's Signature ________________________________

Supervisor's Signature ________________________________

Head of Department's Signature ________________________________

Appendix C

Participant Consent Form

The following pages contain the consent form that participants were required to read and sign before taking part in the user testing. A discussion of informed consent is in Section 5.3.

Participant Consent Form

User testing of the Freehand Formula Entry System

Steve Smithies

1. Background

For my Masters thesis, I have created a system which enables a user to enter formulae into a computer by using either a mouse or a pen and tablet.

You are being invited to take part in user testing, which means that you will be asked to try out my system, comment on it, and compare it to any other systems you know of.

The aim of this study is to test the system that has been developed. I am not testing you, the user. Any positive or negative feedback that you give will help me to evaluate and develop the system.

2. The Process

- **Introduction.** You will be first given an introduction to the system so that you know how it works and how to use it. This will be essentially a tutorial on using the system.

- **Testing.** There will then be a set of tasks for you to complete. The tasks will involve entering and modifying formulae, within the capabilities of the system. You will work through these with as little assistance from the tester as possible, though the tester will be available if you get completely stuck or frustrated.

  Unless you have any objections, I will be videotaping the computer's screen and keyboard as you work so that later analysis may be performed.

  There are no ulterior motives to this study. Everything we are testing will be made clear to you.

- **The questionnaire.** During and after testing the system, you will be asked to fill out a short question sheet about how you found the system.

  You will be given the questions before you begin, and you can choose to answer them either as you go, or at the end.

- **After testing.** Upon completion of the tests, you may be asked additional questions by the tester. You will also be given the opportunity to discuss the system with the tester.

Once you're gone, the video tape will be reviewed to analyse any areas in the system that caused you trouble.

Once I have tested a sufficient number of users, all the results from the question sheets will be collated and analysed.

Findings from this study will be used as part of my final Masters thesis, and will also be used in papers submitted to journals and conferences. In no way will you be mentioned by name, or have any findings traceable back to you.

3. Participation

Your participation in this is totally voluntary, and you are free to stop at *any* time, no questions asked.

By signing below, you are stating that you have read and understood the above, and agree to participate in the study.

Participant's Name ____

Participant's Signature ____

Appendix D

Anonymous Questionnaire

The following pages contain the anonymous questionnaire that was filled out by the participants in the user study after doing the testing, described in Section 5.4. The answers that participants gave on this questionnaire are included in Appendix F.

Thank you for taking part in my user testing. During or after your use of the system, please answer the following questions by either circling the appropriate answer, or writing in the space provided.

Answers you give here are completely confidential, and will not be seen until all the user testing is finished and all answer sheets have been gathered together.

Questions

Have you used or seen this system before? **No** **Seen** **Used**

	Never					**All the time**
How much do you normally use computers?	**0**	**1**	**2**	**3**	**4**	**5**

What is your occupation?

What is your overall impression of this system?

What did you find to be the best parts of the system?

What did you find to be the worst parts of the system?

Comparisons

Have you ever used any other formula entry systems before? **Yes** **No**
(For example: Microsoft Word, Mathematica or LaTeX)

If you answered "No" to the previous question, please skip the remainder of this section and go onto the to the comments section.

Which other systems have you used?

	Never					**All the time**
How regularly do you use the other systems?	**0**	**1**	**2**	**3**	**4**	**5**

	Worse					**Better**
How would you rate the style of interaction of this system against others you have used?	**0**	**1**	**2**	**3**	**4**	**5**

Which aspects of this system made it better than others you have used?

Which aspects of this system made it worse than others you have used?

Comments

Are there any other comments that you wish to make?

Appendix E

Oral Questionnaire

The following questions are the questions that I asked the participants in the user testing, after they had used my program and filled out the anonymous questionnaire. The answers given to these questions are included in Appendix G.

Oral Questionnaire

User testing of the Freehand Formula Entry System

What did you think of the way that the system annotated and boxed the characters you wrote as you entered them?

When it didn't get all the strokes right for a particular character, how did you find the "modify stroke groups" method of drawing a line through the strokes to collect together/break up the strokes?

When the system misrecognised your character, given that the stroke groupings were correct, what was your opinion of the "modify character" system for choosing the correct replacement for it?

When the system was unable to recognise your formula, how useful was the system in helping you to correct it?

Appendix F

Anonymous Responses

This appendix shows the responses given by users in the anonymous questionnaire that they filled out after testing my system. Duplicate answers have been removed, or noted where appropriate. Annotations to answers in square brackets are comments made by me that weren't part of the original answers.

Have you used or seen this system before?

- No (5 participants)
- Seen (2 participants)
- Used (1 participant)

How much do you normally use computers? (0 to 5 scale)

- 4, 5, 4, 5, 5, 4, 5, 4, 3

What is your occupation?

- Student. (5 participants)
- Atomic Physicist.
- Analytic Mathematician.
- Maths Postgrad Student.

[Two of the "Student" responses were high school students (4th to 6th form), the others were university students (Year 3 or above).]

What is your overall impression of this system?

- Concept great.
- Interesting and potentially useful, but it will have to handle significantly more complex formulae for it to become useful to the average physicist or mathematician.
- Good, with a lot of potential.
- Useful - needs a few small tweaks to make it really nice → though so do most programs.
- Once tricks are mastered, the system is quite easy to use.
- Good. Fun to play with. Can see how it *would* be useful.
- Good. Easy to learn and almost no learning curve.
- Seems potentially useful, problem with speed, really need to be able to write it in faster than can type i.e.: TEX for it to be useful.

What did you find to be the best parts of the system?

- Recognition.
- The fact that it works. The "organising the equation" graphics [formula display window] was fun to watch.
- Not having to use LATEX.
- Flexibility - able to write reasonable scribbly without problems → also not written perfectly straight.
- Drop-drown list for wrong recognition of characters, quick re-recognition of groups.
- Simple modification system.
- Drawing rather than typing. I also like the modify stroke groups bit. Plus pretty formula at the end.
- The interface.
- Doing super/subscripts it recognised very well, and the ease at which you can modify the input.

What did you find to be the worst parts of the system?

- Parsing.
- Handwriting recognition. Some slight differences between the way the computer sees the equation and the way the human does.
- Character recognition and speed.
- Fiddly with formulae to get them right without feedback as to where to change.
- Getting spacings/relative locations right, [character recognition] interrupting drawing.
- Bad symbol recognition.
- Waiting for it to find a correct solution, and how the strokes keep losing bits. [character recognition interrupting drawing.]
- Occasional crashes.
- The "4" fixation, the aforementioned slowness, and the understandable limited vocabulary - e.g. Greek letters, etc.

Have you ever used any other formula entry systems before?

- Yes (7 participants)
- No (2 participants)

Which other systems have you used?

- LaTeX, LyX, MSEE
- LaTeX, MSEE, Texturer
- LaTeX, MSEE
- LaTeX, MSEE (eughh!)
- MS Equation Editor.
- LaTeX, Mathematica

How regularly do you use the other systems? (0 to 5 scale)

- 0,3,1,1,1,3
- 1, unless I'm writing up.

How would you rate the style of interaction of this system against others you have used? (0 to 5 scale)

- 5,3,5,3,5,4
- Basically neutral because I know LaTeX. versus MSEE, its a 5. anything is.

Which aspects of this system made it better than others you have used?

- The use of a writing tablet against a keyboard as the preferred type of input.
- Not having to remember symbol names or syntax.
- More natural - writing rather than typing w[sic] abnormal strokes.
- More intuitive - easily written - less restrictive.
- More intuitive, less need to remember tags.
- More visual representation of equation.
- It is easier to visualise what you want by putting it in the right place, rather than learning all the commands, etc.

Which aspects of this system made it worse than others you have used?

- It uses equipment that is not common in the public.
- The character recognition system wasn't too hot.
- Not being able to write at full speed and having the system miss characters.
- Slower than LaTeX - if you know what you want.
- Tolerance to lazy [writing]. I'm sure the user would improve over time :).
- Slow at generating with LaTeX.
- Simplicity of maths available.

Are there any other comments that you wish to make?

- The formulae will still need tweaking to look good.
- A bigger tablet would be good. Hot keys for mode swap. [the participant possibly didn't find the mode changing option on the menu with the hot keys beside them.]
- It should automatically go into draw mode when starting an new screen.
- Seems like it should be real good, allows intuitive feel and to actually try and do solutions as you work.

Appendix G

Oral Responses

This appendix presents the answers, given by the participants in the user testing, to the oral questions that they were asked after using the system and filling out the anonymous question form.

What did you think of the way that the system annotated and boxed the characters you wrote as you entered them?

- Good. It showed you what it recognises straight away.
- Helpful.
- Not distracting.
- Boxes not obviously separate in (for example) $\sqrt{2}$. "Otherwise Cool". Darker boxes as they overlap, perhaps.
- For things like $\sqrt{2}$, boxes overlap and are indistinguishable. (Several people have encountered this problem.)
- Too much grabbing too quickly. When it decided to start recognising what the user had written, the system ended up swallowing up strokes that were part of the character that the user was entering at the time. [This was a bug. Now fixed.]
- How about letting the user do all their writing then invoking the recogniser afterwards?
- "Reasonable intuitively way."

- "Showed me what it was thinking."
- Distracting/annoying? → No.
- Good that it shows you what it has done, and that it has had an attempt at recognising your characters.
- The colour scheme makes it hard to read the character overlaid on the box/strokes. (mentioned twice)
- Distracting? → No.
- Helps you see how things are grouped.
- Tended to group things badly.

When it didn't get all the strokes right for a particular character, how did you find the "modify stroke groups" method of drawing a line through the strokes to collect together/break up the strokes?

- "Great."
- "Intuitive."
- "Quite good."
- Easy to use/understand? → "Yes."
- "Favourite bit". Liked drawing lines through everything. Enjoyed it.
- Scribble thing was good. Easier than drawing a box around it.
- "Pretty good."
- "Quick and easy."
- Very easy to use.
- "Fun."
- Good that the single thing was able to both split and combine strokes.
- Nice that could either squiggle over things or draw a line.

- “Good.”
- Especially easy to put =’s back together when the two strokes weren’t grouped properly.
- “Easy to regroup things.”
- Useful.
- Having a different coloured stroke when you were in this mode is a good visual cue.
- Easy to understand and use? → Yes.
- Was nice that could draw a squiggle to indicate which strokes, rather than drawing a box around things. Made it easy to indicate what was wanted, and avoid other stuff. Would be good for select + move.

When the system misrecognised your character, given that the stroke groupings were correct, what was your opinion of the “modify character” system for choosing the correct replacement for it?

- Liked it.
- “Good system” → could do with more alternatives than the 5 that it gives.
- “All right” maybe some easy way of marking stuff on the fly as you’re writing, that you need to come back to later.
- Good. Didn’t like typing things in though. Buttons for *everything* (including letters, numbers, etc.) would be good. If going to do formula entry with pen, should make it possible to use only the pen and not need to touch the keyboard at all.
- “That was good.”
- “Very handy.”
- Should add to the modify character entry thing a toolbar thing with the entire alphabet and all symbols.

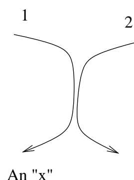
- Pretty good.
- Without changing mode, click on the character in the top left to bring up the pop up box with alternatives. While using the system, the user tried to do this.
- "Good. I liked the pop up menu."
- This is an essential thing.
- Sometimes the alternatives provided on the pop up menu didn't seem to be obvious/reasonable alternatives for what was written, and occasionally an obvious alternative wasn't there at all.

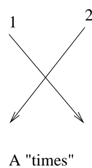
When the system was unable to recognise your formula, how useful was the system in helping you to correct it?

- Need a reasonable understanding of how the system works to be able to fix problems.
- It wasn't useful at all.
- The formula graph display need to be made more "user friendly." Too cluttered. Less lines would be good. [i.e. the weight lines]
- Totally unhelpful. Only because of [the observer's] help that knew what to do.
- "Could be more useful."
- Should come back with suggestions of where you could put things to make it clearer.
- Had to fiddle to get equations to work.
- Didn't help much.
- After a bit of use, you would start to learn the tricks necessary to make it (i.e.: not being able to parse) less of a problem.
- The formula graph display window was useful as it showed what it was doing wrong.
- Wasn't always obvious what was needed to fix things.

Other comments

- Minor problem: Using delay to determine when you had finished writing was annoying
- It would be good to see, as you were writing your formulae in, the associations that would be make in the final parsing of the formula. e.g. what would become the integrands of an integral.
- It would be nice to teach it about mathematical conventions to make it nicer/easier to use.
- The difference between an "x (the letter x)" and a "$\times$ (times) is:





- Need to be able to enter formulae split over a number of lines [this feature needs to be provided].
- Going to need to be able to handle a lot more complex formulae before this will become something of use to a "real" mathematician.
- How about gridding the screen, then getting the person to write in the grid-squares?
- As well as making the character in the top left a magic thing for changing the character, could also maybe do something similar for stroke groups.
- Would be good to have a thing to find out how you should draw a character. (Smartboard has this.)
- Make the drawing cursor look like a pen.
- Autoswitch to draw mode on file→new.
- Hold down shift (perhaps) to get to modify stroke groups mode.
- Could also use the squiggle concept for selecting strokes or characters to move, instead of using a rectangular region.

- Needs a better colour scheme.
- Colour coding of the stroke drawn was good for showing the different modes that the system was in.
- Would be good to have on the fly training of the character recogniser.
- Idea: If you draw a new stroke that intersects with, or passes near to something already written (even if recognised), then re-run the recogniser with the additional stroke on the strokes involved.
- The way that people conceptually *see* things is different to how it is written or typeset. For example, the limits on $\int_b^a c$ are *seen* as being "above" and "below", although they are typeset to the side of the integral sign.
- Selecting stuff with a rectangular area should only select the things that are fully enclosed by the rectangle. This would enable you to pick out things inside other things (e.g. the z in $\sqrt{z}$).

  [This leads to the idea... When you're selecting stuff, there are two sort of actions. First, there is a tap to select a single item, and there is a boxing-of-an-area to select multiple stuff.]
- The ability to interact with the formula graph display as it was parsing would be a good idea, so that you could indicate to it which things should be together if it was getting them wrong, or taking a long time. Would involve drawing the actual strokes on the formula graph display as well.

Other observations made by the observer

- Many people try to use the delete or backspace key to cut.
- One user called "modify stroke groups" "group characters".
- One user commented, and also noticed with other users, that it was tempting to go back and fix up characters that had bits missed out. Of course, this confuses the character recogniser.

Appendix H

Raw Data

This Appendix shows the raw data gathered by analysing the recordings of the computer's screen during the user testing. Conclusions from and graphs of this data are presented in Section 6.1.

H.1 Error Rates

The following table presents the total number of characters written across all five formulae and the number and percentage of misgrouping and misrecognition errors.

Test User	Characters Written	Total Misgrouped	Total Misrecognised	Percentage Misgrouped	Percentage Misrecognised
1	80	8	22	10%	28%
2	97	19	24	20%	25%
3	53	10	12	19%	23%
4	72	8	18	11%	25%
5	66	7	11	11%	17%
6	81	17	16	21%	20%
7	54	9	16	17%	30%
8	80	12	16	15%	20%
All	583	90	135	15%	23%

H.2 Parsing Attempts

This counts the number of parsing attempts each user took to parse each of the formulae. Note that participant 7 did not enter the second or third formulae, and participant 1 didn't attempt to parse the first or second formulae. If they had attempted to parse them, these formulae would have parsed first time.

Test User	Formula 1	Formula 2	Formula 3	Formula 4	Formula 5
1	-	-	4	2	3
2	1	1	4	3	4
3	1	2	2	2	1
4	1	2	5	3	4
5	1	1	1	2	4
6	1	1	1	4	9
7	1	-	-	2	1
8	1	1	2	4	5
Average	1.0	1.3	2.5	2.8	3.9

H.3 Drawing and Correction Times

This table presents the time spent making entering formulae and correcting errors. Each of the columns numbered from 1 to 8 represent a participant in the user testing.

The activities are: drawing, which is drawing strokes; grouping, making regrouping squiggle-selects, or looking for grouping errors; recognition, using the pop up menu to correct recognition errors, or looking for character recognition errors; and parsing, waiting for the parser.

	Activity	1	2	3	4	5	6	7	8	Average
Formula 1	Drawing	21	8	18	15	12	11	9	7	12.6
	Grouping	0	5	8	6	6	0	0	0	3.1
	Recognition	0	2	0	0	0	3	0	0	0.6
	Parsing	-	0	0	0	0	0	0	0	0.0
Formula 2	Drawing	26	16	17	56	23	22	-	17	25.3
	Grouping	8	14	17	10	0	0	-	19	9.7
	Recognition	0	2	13	14	3	3	-	18	7.6
	Parsing	-	0	0	2	0	13	-	0	2.1
Formula 3	Drawing	62	29	38	57	43	53	-	29	44.4
	Grouping	16	0	9	0	11	6	-	19	8.7
	Recognition	8	24	16	42	0	13	-	27	18.6
	Parsing	81	0	11	23	0	11	-	2	18.3
Formula 4	Drawing	59	30	57	52	36	41	48	86	51.1
	Grouping	13	18	10	12	4	17	9	9	11.5
	Recognition	15	45	48	55	15	63	43	38	40.3
	Parsing	14	9	5	12	5	20	0	11	9.5
Formula 5	Drawing	68	34	85	48	60	74	66	39	59.3
	Grouping	7	29	14	47	15	12	24	17	20.6
	Recognition	17	21	13	28	27	39	38	15	24.8
	Parsing	18	5	0	24	78	32	0	106	32.9
All Formulae	Drawing	236	117	215	228	174	201	-	178	192.7
	Grouping	44	66	58	75	36	35	-	64	54.0
	Recognition	40	94	90	139	45	121	-	98	89.6
	Parsing	113	14	16	61	83	76	-	119	68.9