An Interactive Mathematical Handwriting Recognizer for the Pocket PC

by

Bo Wan

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
December, 2001

Abstract

Handwriting is the primary input method for hand-held computers because they are too small physically to have keyboards.

To investigate the requirements for upcoming computer algebra systems on hand-held computers, we designed and implemented an application for recognizing on-line handwritten mathematical expressions on the pocket PC. The objective was to translate handwriting mathematical expressions into corresponding presentation MathML, which can be understood by computer algebra systems.

This application consists of three components: 1) a handwriting recognizer for recognizing individual mathematical symbols, 2) a structural analyzer for interpreting and maintaining the relationship between symbols of the expression, and 3) a generator for generating MathML code. Currently it is able to recognize simple expressions including polynomial equations, fractions, trigonometric functions, allowing nested structures. This application could serve as a bridge for mathematical users to interact with the computer algebra systems on hand-held computers.

Keywords: handwriting recognition, mathematical expression recognition, presentation MathML, Pocket PC, computer algebra system

Acknowledgements

First and foremost, I would like to extend my sincerest gratitude to my supervisor, Dr. Stephen Watt, for his consistent guidance, encouragement and support during my graduate studies, and for his time and patience for reading and correcting my thesis.

Many thanks to Mr. Luca Padovani and Dr. Bill Naylor for their precious advice and help during the design and implementation of the program, and for reading and correcting part of my thesis.

Thanks to Mr. Jason Selby and Mr. Yannis Chicha for configuring the development tools and the Windows 2000 operating system that I have used for my thesis, and to Ms. Yuzhen Xie, Mr. Igor Rodianov and Mr. Cosmin Oancea for their important comments.

My thanks also go to Dr. Rob Corless, Ms. Bethany Heinrichs, Ms. Janice Wiersma, Ms. Dianne McFadzean, Ms. Cheryl McGrath, and every other person who has given me help in various forms during my stay in the ORCCA lab and the Department of Computer Science.

Finally, my deepest thanks go to my family, especially my parents, for many years of love and support, and my lovely wife Jenny Li Zheng, for sharing the good and hard times with me over all these years. Without them, this thesis will not be possible.

Table of Contents

Table of Figures

Chapter 1 Introduction

1.1 Why Is Handwriting Important?

Since the middle of the 1990s, the market for small hand-held computers has been seeing a surge in popularity. Small hand-held computers, also known as *Personal Digital Assistants* (PDAs), are portable computers that are small enough to be held in one's hand or carried around in a pocket. Hand-held PCs, pocket PCs, and Palm series devices, *etc.* are all hand-held computers. Among them the most popular today are pocket PCs running Windows CE and PDAs running Palm OS.

Compared with Palm devices, pocket PCs are much more powerful: A pocket PC is equivalent to an Intel Pentium machine in speed, while a Palm device is equivalent to an Intel 386 machine. A Palm OS based device usually has up to 33 MHz, 16-bit CPU, and 2 - 8 MB of memory, while a pocket PC usually has 32/64-bit CPU at a speed of 70 - 206 MHz, and 16 - 32 MB of memory. A pocket PC also has better display resolution (320 x 240) over Palm devices (160 x 160). In fact, Palm PDAs are used typically only as advanced digit organizers. On the contrary, a pocket PC is meant to be a fully functional PC, therefore it catches more interests from the software developers' community.

As PDAs are made smaller to be easy to be carried around, *it is no longer possible for them to physically have conventional keyboards as input devices*. Handwriting is one of the most natural ways for human-computer interaction. Therefore it becomes a primary input method for hand-held devices. Each of these devices has a specially made screen called a *digitizer*. When one writes on the digitizer with a special stylus, the writing is digitized as a sequence of points, at the same time an electronic ink trace is displayed on the digitizer. Recognition software is used to analyze the sequence of points and pass the recognition result to other applications.

1.2 Why Handwriting Math?

The variety of built-in handwriting recognizers shipped with the hand-held computers have very limited function. They were designed with the assumption that people will use these devices for writing short memos, editing simple texts, storing addresses and telephone numbers, *etc*. This assumption is becoming increasingly challenged however. People expect hand-held computers eventually to serve the same purposes as desktop or laptop computers do. Although currently we cannot obtain this goal, hopefully this can be done with the improvement of hardware of hand-held computers. However, the handwriting recognizers may prevent us achieving this goal if we can not improve them. Computer algebra systems provide an example, handwriting math is preferred as input, and this makes it necessary for the recognition of mathematical expressions.

Handwriting for mathematics is a perfect example of non-linear input. The usual input for handwriting cannot be used in this context: mathematical input is a cross between drawing and handwriting. The study of handwriting math will no doubt eventually be related to research on other non-linear input methods, such as chemical formulas, musical notation, and so on.

1.3 Need for Computer Algebra Systems for PDAs

If we take a look at the computer algebra systems like *Maple* or *Mathematica*, widely used in the scientific research and academic world, we are deeply impressed by their functionality. At the same time we can not help thinking: Can we use them on PDAs? For now, the answer is “no”, but with a few advances the answer can be “yes”.

The major limitations that prevent hand-held computers to be able to host this kind of software is their CPU speed, memory and storage capacity, which are now barely powerful enough. Compred with desktop and laptop computers, the CPU speed of PDAs is very slow (206MHz of Compaq iPAQ’s StrongARM processor vs 2GHz of the latest Intel Pentium 4 processor). Also PDAs use expensive flash memory as both memory and secondary storage, which is not practical for computer algebra systems since they require

a great deal of memory and storage. For these reasons there are not yet commercial computer algebra products available for hand-held devices. However, we can expect things to change quickly.

Let's take a very brief look at the record of processor speed. In 1965 Gordon Moore, the co-founder of Intel, observed that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented, and predicted that this trend would continue for the foreseeable future. His prediction is known as *Moore's Law*. In the past three decades, the development of computer hardware almost has followed Moore's Law. For example, the speed of CPUs has doubled almost every 18 months, and this trend still holds right now. Today, Compaq's iPAQ pocket PC is already as fast as the fastest desktop PC (Intel Pentium 200) of four years ago.

From here we can be sure that with the fast improvement of computer hardware, in the very near future, PDAs will be powerful enough to host many applications that they cannot host now, including computer algebra systems. This will no doubt be a great help to the users like physists, engineers, and science students. However, there is a problem that must be solved in order to take advantage of the computer algebra systems on these devices: how to input math in the computer algebra systems on PDAs?

1.4 How to Input Math on PDAs?

Inputting mathematical expressions into a computer is usually more difficult than that of plain text because mathematical expressions typically consist of a two-dimensional layout of special symbols and Greek letters in addition to Roman letters, digits and other symbols. Such a large number of symbols are impossible to input into a computer with the conventional keyboard using single keystrokes.

One way to deal with this problem is to make use of some extra keys in the keyboard along with a set of unique key sequences for representing other special symbols[29]. Another way to deal with this is to mark up the special characters and symbols, as in $\TeX$. These approaches require intensive training and practice, and from the user's point of view are not easy to use.

Certain applications such as mathematical editors and computer algebra systems use equation templates. The templates provided by the application include most of the mathematical elements, such as symbols, delimiters, relations, functions, calculus, and matrices. All the user needs to do is to select the corresponding templates to complete the expression. Figure 1.1 shows the templates provided by Adobe FrameMaker 6.0. This approach is WYSIWYG (What You See Is What You Get), and there is almost no training necessary for the user. It is therefore quite popular. However, the user has to switch between the templates and the editing area all the time. As a result this approach usually slows the user down. Another problem is that the templates take quite a lot display area. For desktop computers or laptop computers, this is not a major problem because the monitors are large enough. But for hand-held computers with very limited display areas, the templates take an area that is intolerable. Obviously, this approach is impratical on hand-held computers.

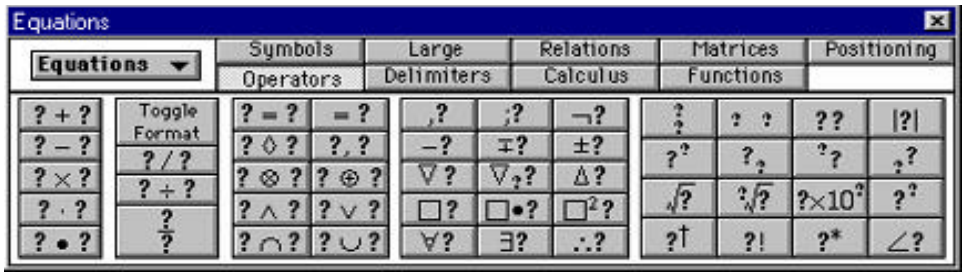


Figure 1.1 The mathematical expression templates as in Adobe FrameMaker 6.0

It could be a good idea to take advantage of the pen-based computing technologies and simply write mathematical expressions on an electronic tablet (digitizer) for the computer to recognize them automatically. This should be the most user-friendly method because writing mathematical expressions on a digitizer is much alike writing on a piece of paper. For hand-held computers, this seems to be the only reasonable option. The problem with this approach, of course, is to find a good recognizer.

1.5 Problems of Existing Handwriting Recognizers

Since handwriting is the primary way of input on PDAs, the preferred method for user to interact with computer algebra systems will also be handwriting. There must be an application that can correctly recognize handwriting mathematical expressions and translate them into a form that can be understood by the computer algebra systems. Unfortunately no such an application is available at this time for the pocket PC environment.

The built-in handwriting recognizers of the hand-held computers are not capable to do this for several reasons:

Firstly, they can only recognize a very limited set of symbols, normally those provided by a standard keyboard. They are not able to recognize most of the mathematical symbols like square root ($\sqrt{\ }$) and integral sign ($\int$).

Secondly, in some recognizers like Windows CE's default recognizer, Roman letters, digits and other symbols are recognized serially in special regions of the input area called a *Soft Input Panel* (SIP), making it impossible to input mathematical expressions.

Thirdly, even if the recognizers allow free writing and can recognize all symbols of an expression, they have no knowledge at all about the spatial relationship between the symbols. For example, *Microsoft Transcriber* is the newest handwriting recognizer on Window CE, it allows free writing and has a pretty good recognition rate. However, it assumes that the handwriting is either a line of text or a picture. A simple expression like "$y=x^2$" will be recognized as "$y=x2$", or sometimes as "$y=$", and treat "x^2" as a picture depending on the handwriting.

The main reason for above problems is that in a mathematical expression, symbols are arranged in a two-dimensional structure, the spatial relationship between the symbols has embedded mathematical meaning. The ability of a recognizer to recognize mathematical expressions depends on the understanding of the spatial relationship. Unfortunately, existing handwriting recognizers are unable to recognize such geometric

relationships. They always assume the input to be linear. That is, characters are written adjacently in the horizontal direction.

1.6 Thesis Objectives

In response to the above problems, we have designed and implemented an experimental pocket PC application for recognizing handwriting mathematical expressions and generating corresponding *presentation MathML* code. The primary objective is to 1) find an effective way to recognize handwritten mathematical symbols, and, more importantly, 2) correctly extract the meaning of the expressions from the spatial relationships of the symbols. Here we use pocket PC as a lower end platform to study the problems of recognizing mathematical handwriting expressions. However, the ultimate goal will be to develop practical mathematical handwriting recognition techniques that can be used on all the platforms.

Our application is developed under Microsoft Embedded Visual C++ 3.0. The target platform is the pocket PC, and the actual device we have used is a Compaq iPAQ 3600 pocket PC running Microsoft Window CE 3.0. Our program is composed of a handwriting recognizer for recognizing individual handwritten mathematical symbols, a two-dimensional structural analyzer for interpreting the relationship between recognized symbols and maintain recognized relationships in a expression tree, and a generator to produce presentation MathML code according to the expression tree.

The application works as follows: it provides the user with a graphical interface which looks like a scratch pad (Figure 1.2(a)). The user can write inside the input area with a special stylus. Every time a symbol is finished, it will be recognized and the result will be shown in the panel named “Recognized”. At the same time, another panel named “Candidates” will show other possible candidates for the symbol. In case a recognition error occurrs, the user can change it easily by picking the correct alternative from the candidate window. Once the symbol has been recognized, the structural analyzer will analyze its relationship with other symbols of the expression and integrate the new

symbol into a proper position of the expression tree. At any time the user can click on the menu "*Edit ® Go MathML*" to view the generated MathML code in a pop up window (Figure 1.2(b)).

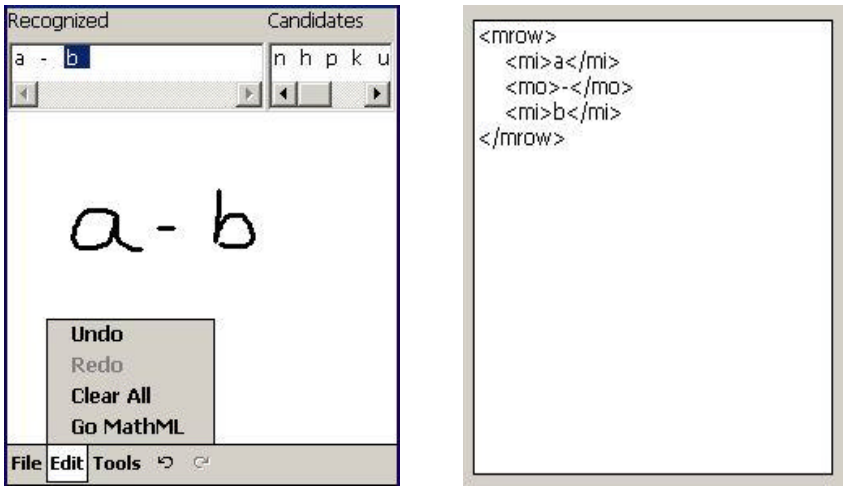


Figure 1.2 Screen shots of the mathematical expression recognizer.
(a) the user interface including the input area and window for recognition result and recognition candidates.
(b) the display of the corresponding generated MathML code

1.7 Organization of the Rest of the Thesis

In the remainder of this thesis we review the problems and existing research related to both handwriting recognition and mathematical expression recognition and also describe and discuss the studies we have performed.

Chapter 2 reviews the problems of handwriting recognition as well as the research results that have been reported on handwriting recognition during the past three decades, with more focus on "on-line" handwriting recognition.

In Chapter 3 we first briefly describe the general design issues such as what development tools to use, how the entire application should be structured, how the user interface should appear, whether to use Unicode or not, *etc*. Then we introduce in detail

the elastic matching method that is used in the handwriting recognizer of this thesis, and the design and implementation of our handwriting recognizer – the *ElasticRecognizer*.

In Chapter 4, we first describe the main features of mathematical expressions and the problems that must be deal with in recognizing their two-dimensional structure. Then we introduce how existing research deals with these problems.

Chapter 5 is dedicated to the details of our approach in the structural analysis procedure, including how to determine the distance and relation between sub-expressions, how to find the proper location and attach a new symbol into the expression tree, and so on.

A relatively short Chapter 6 mainly introduces the approach we use to generate MathML code according to the contents of the expression tree, including postprocessing of the expression tree and MathML code generation.

The last chapter, Chapter 7, addresses the results we have achieved, the existing problems with the application, and future work that should be done for fixing current problems as well as improving performance and usability.

Finally, Appendix A briefly introduces the *ModelBuilder* application that is used to build and maintain models for the *ElasticRecognizer*.

Chapter 2 Handwriting Recognition Review

Handwriting is one of the most natural ways of communication between people. What makes this possible is the fundamental property of handwriting as described by Tappert *et al.* [36]:

> *The fundamental property of writing which makes communication possible is that differences between different characters are more significant than the differences between different drawings of the same character. Some people argue that there are exceptions to this. Since O and 0 (or I and 1) can be drawn identically, although context usually provides information necessary to distinguish letters from numbers. Nevertheless, written communication is not possible without this fundamental property.*

Research on handwriting recognition has been going on for nearly forty years. In 1957, the earliest electronic *tablet* (or *digitizer*) called Stylator was invented [15]. It was followed in 1963 by a better known device called the RAND tablet [13]. These devices were able to detect the *X*-*Y* coordinates of the tip of a writing tool, therefore able to digitize the handwriting into a sequence of points. As a result the research on character recognition began. This activity has been neatly summarized as having "*lasted through the 1960's, ebbed in the 1970's, and renewed in the 1980's*" [35]. With the increasing popularity of hand-held computers, digital notebooks, and advanced mobile phones, handwriting recognition has gained more interest recently. A great many books and papers in this field have been published, and at the same time, more and more commercial products become available. However the handwriting recognition problem is so complex, our understanding of this problem is still far from mature. Up to now there is no solution that could solve this problem both efficiently and completely.

In this chapter we address the problems of handwriting recognition and review the research on these problems.

2.1 On-line and Off-line Recognition

The existing handwriting recognition methods can be divided into two main categories: off-line and on-line recognition. “On-line recognition means that the machine recognizes the writing while the user writes. The term *real-time*, or *dynamic*, has been used in place of on-line” [36]. Off-line handwriting recognition, by contrast, is performed on the scanned image of the handwriting [36].

Off-line handwriting recognition is a subset of *Optical Character Recognition* (**OCR**), which also does recognition on machine-printed characters. Basically, in OCR the characters are either written by a user or printed by a machine (printer, plotter, *etc*.), a scanner is then used to acquire an image of the characters. Recognition is performed on the scanned image.

Both on-line and off-line systems have their advantages and disadvantages.

In the on-line approach, each completed character is recognized immediately, this makes it easy for the recognizer to interact with the user and receive user’s feedback. For example, the user can fix a recognition error right away. On-line systems also have the advantage of being able to capture more information of the writing. The information is usually dynamic, includes the *number* of strokes (a stroke is the writing from pen down to pen up), the *order* of the strokes and the *direction* of the writing for each stroke. It is even possible to get the information of the *speed* and the *acceleration* of the writing within each stroke [34]. This information makes the recognition work easier than off-line recognition. However, due to the nature of interacting with the user, the recognition speed is critical for an on-line system, it must be able to respond fast enough in real time to user’s actions.

In off-line recognition there is neither interaction between the recognition program and the user nor any dynamic information. The recognizer thus has to do more than an on-line recognizer. One advantage is that for off-line systems there are no restrictions on the recognition speed: speed is only a measure of performance, but not of quality. Another advantage is that in off-line systems all symbols can be seen at the same time, and therefore more contextual information can be provided to the recognition process.

Obviously, the on-line approach provides more interaction, and therefore is more user-oriented, and is superior to off-line approach for our application. Based on this observation, in this thesis, the on-line approach is selected.

One issue that needs to be addressed here is that even in an on-line system, there are still two ways to perform the recognition. One way is to perform the recognition "on the fly", as soon as the user finishes a character, it will be recognized. The other way is to perform recognition in batch mode, where recognition will not be performed until all characters have been entered. Our choice was to use the first style because it offers better interaction.

In the past forty years many different approaches have been proposed for character recognition, such as pattern recognition, statistical, syntactic and structural approaches. Some of these are off-line methods but can be easily modified to be used in on-line situations. We outline these on-line recognition methods in the following sections.

2.2 Pattern Recognition Methods

In this category of methods, recognition is performed *statically*. The information of pen motion is not used at all, and the only thing that gets examined is the collection of points left by the pen. This is much like off-line methods, but not exactly the same. The off-line methods have to extract the points from a bit map, at the same time try to get rid of possible noises. For on-line methods this is not a problem.

Methods in this category include bitmap comparison, histogram comparison, and neural networks. Among these methods, the neural networks are the most widely used, and provide the best recognition. The following is a brief introduction of the methods in this category.

2.2.1 Bitmap Comparison

This method compares the bitmaps of both the writing and the models (templates), and calculate the number of pixels that differ as distance [21]. The model that gives the minimum distance is the recognized.

This method is simple and has good recognition speed. However, the recognition rate on handwriting is very poor [22]. It only works well on machine-printed symbols.

2.2.2 Histogram Methods

This method calculates the one-dimensional projection of both the unknown and the model bitmaps, and compare their histograms [21]. The method requires that the histogram must be unique for each character, therefore it is not working well with large set of characters.

2.2.3 Neural Networks

Neural Networks [27] crudely simulate the signal propagation found in human brains. Brain cells are connected together as a network. Each cell has several activation levels, the current activation level is determined by the activation signal a cell receives from other cells. In neural networks nodes are also connected in a network, and the signal transmission is controlled by activation functions. A typical neural network has a layered structure with an input layer for receiving input information, and an output layer for encoding the results in the activation of the output nodes. In between, any number of layers is possible.

In handwriting recognition, the image of the handwriting is given to the neural network, which extracts the features such as size, position, directional features, and shape features from the drawing and feed them to the input nodes. The output layer consists of a vector of nodes, each of which corresponds to a character. The activation value of each output node equals the probability for the corresponding character to match with the drawing.

The main advantage of a neural network is that it is trainable, it can adapt the activation function in order to match the input patterns to output patterns. However, the training process is usually tedious. Futhermore, the networks are usually too large in size to be used on hand-held computers.

Quite a number of mathematical recognition systems, including Ha *et al*. [20], Dimitriadis and Coronado [14], and Marzinkewitsch [26] use neural networks.

2.3 Statistical Approach - Hidden Markov Model (HMM)

HMMs have been used successfully in both speech recognition [2] and handwriting recognition [10]. Simply speaking, an HMM is a finite state automaton in which the transitions between states are probabilistic rather than deterministic. The transition probabilities are based on the probability of occurrence of the various features. In handwriting recognition, the writing is processed to extract its features. The features are then compared to the HMM, which contains nodes for all the characters of the character set. The first terminal node reached is the recognized symbol. Like neural networks, HMMs are also trainable.

2.4 Structural and Syntactical Approaches

These approaches use both structural information and syntactic rules in the recognition. In brief, they use grammar rules to describe different shapes in a formal language, which uses a set of features such as straight line, dot, counter-clockwise / clockwise curve, and loop. as tokens. In the recognition, the features are extracted from the writing and then processed by the grammar.

The most important issue in these systems is to correctly extract the features from the unknown. Usually Freeman's chain code [17] (Figure 2.1) is used to determine how a point is connected to the next in the sequence of points. It consists of eight values, 0 to 7, to indicate the eight possible directions.

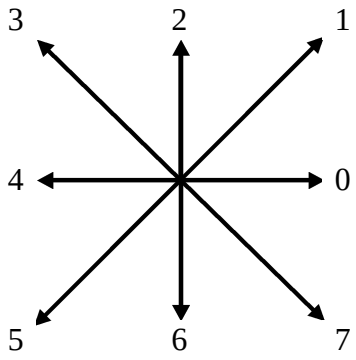


Figure 2.1 Direction values used in Freeman's chain code

In the preprocessing stage, all values that are the same as their preceding ones in the chain code are removed in order to shorten the final chain code without lose important information. This is necessary to improve the recognition speed. The resulting chain code is then analyzed to obtain the primitive features.

This approach is model based: once the primitive feature of an unknown character is obtained, the grammar will compare them with the collection of models to find the one closest in shape. Figure 2.2 shows the structural representation of some characters in Chan and Yeung's system [6].

One problem with this approach is that different people write in different ways, and even the same person may not always write a character in the same fashion. It is impossible to exhaust all the ways to write a character. Futhermore, making a model out of each of the ways means a huge collection of models, which is not good for the performance. For example, Figure 2.3 shows 9 possible ways to write the digit "5" (from Chan and Yeung [6]). Another problem is that analyzing and extracting features from handwriting is very complicated.

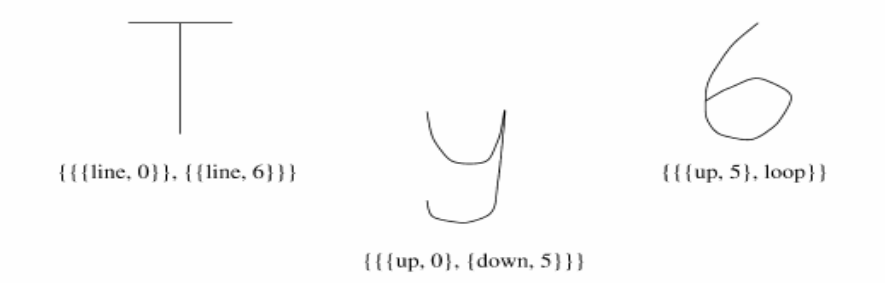


Figure 2.2 Some handwriting characters and their primitive features.
A primitive feature contains both a shape and a direction.

Figure 2.3 Nine possible ways to write the character '5'

2.5 Elastic Matching

Elastic matching is commonly used in on-line handwriting recognition, and many on-line handwriting recognition systems were developed with it [32]. This method was first introduced in 1988 by Tappert [34], who used elastic matching as recognition engine in a run-on handwriting system.

Elastic matching is a model based technique, and it is an application of the dynamic programming algorithm. In this method, the comparison between the unknown character and a given model is done dynamically. Compared with structural and syntactic approaches, this method does not need to do complex feature extraction. In our recognizer, elastic matching is used. In the next chapter we discuss this method in more detail when introducing our handwriting recognizer.

Chapter 3 Handwriting Recognition Design

There are many questions to be answered before we can actually begin writing a program. We have to decide what development tools to use, what kind of user interface should the program provide, how the whole application should be structured, and so on. In this chapter we address these issues first, then move on to the details of our design for handwriting recognition.

3.1 General Considerations

3.1.1 Development Tools

Microsoft Visual C++ and Java are two of the most widely used development tools right now. Both of these are very powerful and easy to use. For example, both language provide many built-in classes which make it very easy to develop GUI (Graphic User Interface) and handle Windows events. There are always arguments going on about which one is better, and the answer is never clear.

One advantage of Java is that it is platform independent; a Java program can run on any kind of machine as soon as they have Java Virtual Machine (JVM) installed. However, at the current time interpreted Java programs are quite slow at runtime for both desktop PCs and PDAs compared with programs written in C and C++.

Compared with Java, Microsoft Visual C++ does not have the luxury of being platform independent. Its program is specific to Windows/DOS platforms. In our case, we chose the Microsoft product for the reasons below. The programming language used is Microsoft Embedded Visual C++.

1. The target platform is a pocket PC, which has a much slower CPU than desktop machines. For example, the fastest pocket PC, Compaq iPAQ machines, have 206 MHz StrongARM CPU, while the fastest Intel CPU today runs at 2.0GHz. Furthermore, handwriting recognition is a computationally intensive procedure. No doubt speed will be critical in our project, and Java will be a bad choice from this point of view.

2. Microsoft Windows CE is currently the most mature operating system for pocket PC, and it is relatively stable. More importantly, it supports Unicode, which is a must for our application since most of the mathematical symbols can only be found in Unicode.
3. A simplified version of Microsoft Visual C++ - Microsoft Embedded Visual C++ is specially designed for developing Windows CE applications.

3.1.2 Dissection of the Application

In order to recognize mathematical expressions, this application needs to do three things: (1) take handwriting input from the user; (2) recognize individual symbols; (3) recognize the relationship between neighbouring symbols.

Our application is designed exactly in this way. It contains a user interface which receives input from the user, a handwritting recognizer for recognizing individual symbols, an analyzer which performs the structural analysis, and a procedure for generating presentation MathML for the recognized expressions. Each part is a module independent from the others, which make it easy to be changed in the future.

3.1.3 The User Interface

The user interface interacts with the user directly. Its main responsibility is to receive handwriting input from the user and to feed back the recognition results to the user. It is required not only to be able to record the data of each symbol, but also be able to keep the two-dimensional information of expressions.

In order to meet these requirements, we decided to use an interface similar to those of drawing applications. Our interface provides a window for the user to write mathematical expressions and two display panels for displaying recognition results and candidates of handwritten symbols. Most of the existing recognizers on pocket PC are not suitable for this purpose because they designate different regions for the input of letters, digits and other symbols. It is impossible in this way to input mathematical expressions since the relationships between symbols are not kept.

Two questions have to be answered at this stage: (1) Should a symbol allowed to be in multi-stroke or must it be in single-stroke? (2) How to distinguish one symbol from another if multi-stroke method is allowed?

- **Multi-Stroke vs Single-Stroke**

Single-stroke symbols are easy from a developer's point of view. The program simply examines pen-down and pen-up events to know when a symbol finishes and the next one begins. The disadvantage is that each symbol has to be mapped to a unique one-stroke alphabet, and this results in somewhat strange looking alphabets. The *Jot* character set (Communication Intelligence Corp.), *Graffiti* (Palm Inc.), and *UniStroke* (Xerox) are all single-stroke methods (There are exceptions, say, *i*, *j* and *x*, *etc.* can be finished in two strokes. However, single-stroke is the dominate case, therefore they are still considered as single-stroke alphabet). Figure 3.1 shows the *Jot* character set. Single-stroke recognition works fine when the symbol set is small in size. However, when it comes to the case of mathematical symbols, single-stroke is not applicable because the set of mathematical symbols has a very large size, it is impossible to find a unique single-stroke representation for each of the symbols. Even if it were possible to design such an alphabet, it would be impossible to remember.

Multi-stroke characters are more user friendly, they allow user to write a character in the usual way without the hassle to memorize the mapping between one-stroke alphabets and their real identities. The problem is that it is hard to distinguish one symbol from another. The general way to deal with this problem is to require the writer wait for a short interval between characters.

- **Our Interface Design**

Our design is in multi-stroke mode. In this design a *stroke* contains a sequence of points and a bounding box (A *bounding box* is the smallest rectangle which entirely encloses a figure, Figure 3.2 shows a bounding box). Each symbol is called a *scribble*, it consists of one or more strokes and a bounding box. In the program there is an internal timer running in the background to distinguish one scribble from another. The timer starts at every pen up, if a new stroke starts before the timer expires, the timer will be cancelled

and the new stroke will be considered part of the same scribble, otherwise it will be considered as the first stroke of a new scribble.

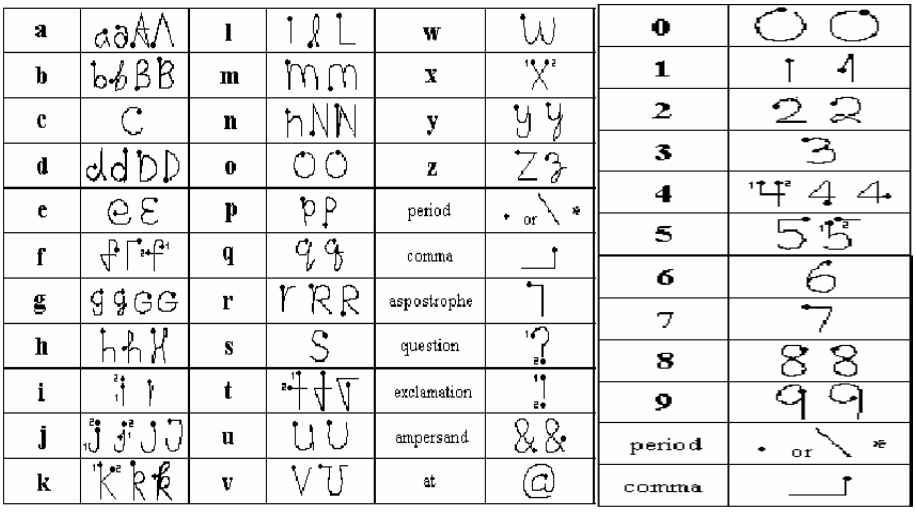


Figure 3.1 The Jot character set. A dot means the start point of the stroke (from *CIC Jot User's Guide*, www.cic.com)

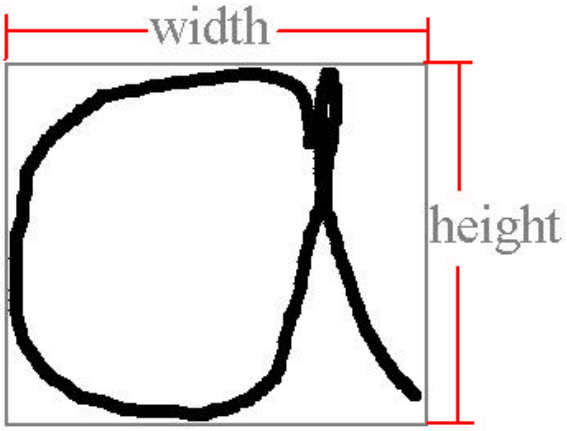


Figure 3.2 A character and its bounding box

Special attention is also paid to the spatial distance between strokes. If a new stroke is far enough from the previous stroke, the new stroke will be considered as the first stroke of a new scribble, even if the interval between the two strokes is within the threshold of the timer. This is reasonable because when two strokes are far apart, they cannot be within the same scribble.

The thresholds for both the timer and the distance between two consecutive strokes are configurable, they can be changed to meet the needs of different writers.

3.2 The Handwriting Recognizer

In the design of the handwriting recognizer, we use an abstract class called *Recognizer*. The actual recognizer used by the system is its subclass called *ElasticRecognizer*, which makes use of the elastic matching algorithm. By this design, it is very easy to change to other handwriting recognition methods.

When a scribble is completed by the writer, it is not ready to be recognized yet. Depending on the algorithm used by the handwriting recognizer, different types of preprocessing have to be performed. Recognition is then performed on the preprocessed data. In the following sections, we introduce the details of preprocessing for elastic matching and the elastic matching algorithm itself.

3.3 Preprocessing

The main purpose for preprocessing is to normalize data, therefore increase recognition rate and speed. For the elastic matching algorithm, three preprocessing steps are needed. They are size normalization, point distance normalization, and smoothing.

3.3.1 Size Normalization

In handwriting recognition, the size or location of a symbol should not affect the recognition result. However, in many algorithms, especially model-based methods, the recognization result depends on the distance between the unknown and the model. The distance itself actually depends on the distance between points of the unknown and the model. In other words, the size and location do affect the result. To solve this problem, it is necessary to performe size normalization on the writing.

This operation rescales all input characters and models to a common size without changing their shape, and at the same time moves their centroid (the point at the center of the bounding box of a character) to the same X-Y coordinate. By doing this, it can be sure

that patterns of the same shape will have the maximum overlap and therefore the minimum distance.

It does not matter what the actual ratio for rescaling is, and where to move the centroid point. What matters is that the operation must be consistent for all the patterns. We arbitrarily chose to rescale all patterns to the size of 100 pixels maximum in height or width, whichever is larger, and move their centroids to the point (50, 50). The calculation is performed using the following equations (Equation 3.1 to 3.7). Here x_c, y_c are the *X*, *Y* coordinates of the centroid, r_x and r_y are the horizontal and vertical radii of the character, $x_{\min}$ and $y_{\min}$ are the minimum *X*, *Y* coordinates of the bounding box, while $x_{\max}$ and $y_{\max}$ are the maximum *X*, *Y* coordinates of the bounding box.

Equation 3.1 $$x_c = \frac{x_{\max} + x_{\min}}{2}$$

Equation 3.2 $$y_c = \frac{y_{\max} + y_{\min}}{2}$$

Equation 3.3 $$r_x = \frac{x_{\max} - x_{\min}}{2}$$

Equation 3.4 $$r_y = \frac{y_{\max} - y_{\min}}{2}$$

Equation 3.5 $$r = \max(r_x, r_y)$$

Equation 3.6 $$x_{new} = 50 + 50\left(\frac{x_{old} - x_c}{r}\right)$$

Equation 3.7 $$y_{new} = 50 + 50\left(\frac{y_{old} - y_c}{r}\right)$$

3.3.2 Smoothing

Writing on the digitizer is not as easy as writing on a piece of paper because the surface of digitizer is usually slippery. This difficulty may introduce noise. On the other

hand, a local sharp change in angle usually causes digitizing noise. Smoothing is a widely used way to deal with this problem.

In smoothing, usually a point is averaged in some way with its neighbouring points [32]. In our recognizer, we also used this strategy by averaging each point with its two neighbors, with the exception that the end points of a stroke will not be processed. The actual calculation is performed as in equation 3.8 and equation 3.9.

Equation 3.8 $$x_i' = \frac{x_{i-1} + x_i + x_{i+1}}{3}$$

Equation 3.9 $$y_i' = \frac{y_{i-1} + y_i + y_{i+1}}{3}$$

3.3.3 Point Distance Normalization

The last step of preprocessing is to normalize the distance between points. The generic way to do this is to start with the first point of a stroke, remove all the points that are at a distance smaller than a given threshold from it, then starts with the next available point and do the same operation, this process goes on until the end point is reached.

This process reduces the number of points involved in the recognition, and therefore reduces the amount of computation and increases recognition speed. This process is performed after size normalization, therefore it is guaranteed that the points in both the unknown and the models are evenly spread out, regardless their original size.

The hardest thing in distance normalization is to choose a proper threshold. If the threshold is too small, the improvement of recognition speed may not be effective, on the contrary, a large threshold may greatly reduce the recognition rate. Experiments by Scattolin [32] showed that for patterns normalized to 100 pixels, the recognition rate is not affected for a threshold up to 6 pixels. In this thesis we follow this result and use 6 pixels as the threshold.

3.4 Recognition by Elastic Matching

Once the preprocessing is done, the *ElasticRecognizer* is ready to perform recognition on the preprocessed scribble with the elastic matching algorithm. This algorithm has several advantages:

1. The research results by Hellkvist [21] showed that systems with elastic matching have higher recognition rate over systems using the Histogram method, Hidden Markov Model, and Neural Networks. Tappert [35] also reported that recognition rate is high for elastic matching.
2. Elastic matching is quite robust, it handles writing errors quite well.
3. Compared with other model-based algorithms especially the structural and syntactic methods, elastic matching requires fewer models. This saves memory space, which is important for PDAs, because on these devices both data storage and computation take place in the flash memory.

3.4.1 Elastic Matching in Detail

Elastic matching is very similar to string matching. They both calculate the minimum distance between two sequences of items by means of the dynamic programming algorithm. Actually elastic matching can be considered a special case of string matching where the strings are sequences of points instead of characters, and the purpose is to find the minimum distance between an unknown string (unknown character) and a collection of reference strings (models).

String matching has three possible operations: **insertion**, **deletion** and **direct matching**. Each of these operations has an associated cost. When matching two strings, we apply each of the possible operations and obtain the smallest total cost. When string matching is done, it is guaranteed that the total cost obtained is the smallest possible distance between the two strings. This makes inexact matching a main feature of the string matching method. Characters between the two strings can have one-to-one mapping which is equivalent to direct matching, or many-to-one mapping which is

equivalent to insertion, and some characters in a string may be skipped therefore do not map to any character in the other string, which is equivalent to deletion.

Elastic matching works in the same way. The Euclidian distance between two points, one from the model and one from the unknown, is used as the cost associated with each of the three operations. The output of the matching procedure is the minimum distance between the unknown character and a given model. Like string matching, elastic matching is inexact, points of the unknown pattern and the model can have one-to-one or many-to-one mapping, it is also possible that some point may be skipped. This is a reason that makes elastic matching robust: “bad” points are usually skipped during the inexact matching.

Figure 3.3 illustrates the distance measurement of elastic matching, in which there are both many-to-one and one-to-one mapping between points of the model and points of the unknown, no points get skipped here.

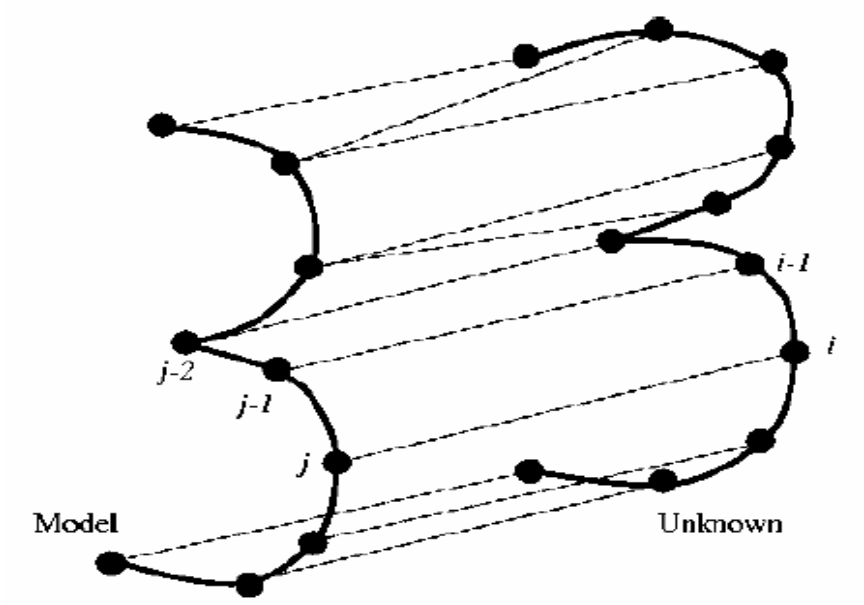


Figure 3.3 The point to point distance measurement in elastic matching.
(from Scattolin [32])

Equation 3.10 shows the calculation of distance between the unknown and the model. In which $D(i, j)$ means the total distance between the unknown point sequence

starting at index i and the model sequence starting at index j. $d(i, j)$ means the distance between the ith point of the unknown and the jth point of the model. The goal is to minimize $D(i, j)$.

Let's take a look at the most general case in this algorithm, the $i > 0$, $j > 1$ case. We can find that the three sub-cases are very similar to the insertion, deletion, and direct matching operations in string matching. When calculating the distance $D(i, j)$, we always have three options: we can match the next point of the unknown to either the next point of the model, $D(i-1, j-1)$, or the same point of the model, $D(i-1, j)$, we can also skip a point in the model, $D(i-1, j-2)$. The minimum of the three cases is selected. This algorithm is called *Dynamic Programming*, which is a general technique for solving discrete optimization problems. In this algorithm, each step of computation depends on the result of the dynamic decision process in the previous step.

In Equation 3.10, the special cases when *i=0* or *j=0* are used to penalize the algorithm when one of the sequences runs out of points. As described by Hellkvist [21], adding these special cases into elastic matching algorithm resulted in better results.

Equation 3.10

$$D(i, j) = d(i, j) + \begin{cases} \sum_{k=0}^{j-1} d(0, k) & \text{if } i = 0 \\ \sum_{k=0}^{i-1} d(k, 0) & \text{if } j = 0 \\ \min \begin{cases} D(i-1, j) \\ D(i-1, j-1) \end{cases} & \text{if } i > 0, \ j = 1 \\ \min \begin{cases} D(i-1, j) \\ D(i-1, j-1) \\ D(i-1, j-2) \end{cases} & \text{if } i > 0, \ j > 1 \end{cases}$$

Note that in Equation 3.10 we represent the algorithm in such a way that is similar to recursion. However a recursive approach would be very inefficient since intermediate values will be evaluated over and over again. So in the actual implementation, we surround this problem by building a lookup table for the result of all the $D(i,j)$ values, therefore each value will be calculated only once.

The distance function d (Equation 3.11) calculates the distance between one point in the unknown and one point in the model.

Equation 3.11 $$d(i,j) = (x_i - x_j)^2 + (y_i - y_j)^2 + C\left|f_i - f_j\right|$$

In this function, two factors have been considered:

1) The first factor is the Euclidian distance between the two points. Strictly speaking, $(x_i - x_j)^2 + (y_i - y_j)^2$ is not really a Euclidian distance but its square. There are two reasons for not calculating the square root: (1) there is no significant improvement in the d function even if Euclidian distance is used. (2) Not performing square root evaluation saves computation time. This is the main reason. Experiments by Scattolin [32] shows that in their elastic matching recognition system, about 60-80% of computation time has been spent on evaluating the distance function. Ignoring square root evaluation step will no doubt improve performance significantly.

2) The second factor is the orientation and curvature of the stroke. This information is also important and should not be ignored. In Equation 3.11, C is an empirically determined constant and f represents the angle of elevation of the tangent to the point. Equation 3.12 shows the calculation of f.

Later in the recognition process, the calculated distance is divided by the number of points of the unknown pattern to acquire the average distance per point. (Equation 3.14, n is the number of points in the unknown, m is the number of points in the model). This makes distance independent of the number of points, and therefore makes it possible to meaningfully compare patterns of different lengths.

Equation 3.12
$$\begin{cases} f_i = \arccos\left(\dfrac{x_{i+1} - x_i}{hyp}\right) & y_{i+1} < y_i \\ f_i = \arccos\left(\dfrac{x_{i+1} - x_i}{hyp}\right) + p & y_{i+1} \geq y_i \\ f_i = f_{i-1} & i = n \end{cases}$$

where

Equation 3.13 $$hyp = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

Equation 3.14 $$\Delta(n, m) = \frac{D(n, m)}{n}$$

It is now clear why and how the elastic matching method calculates the minimized distance between the unknown pattern and a given model. By calculating the minimized distance between the unknown pattern and all available models, we can compare the distances we have obtained and select the one that gives the minimal distance to be the recognition result.

3.4.2 More About the *ElasticRecognizer*

In this thesis, the *ElasticRecognizer* uses exactly the above elastic matching technique. At the initialization stage, the application loads all model files into memory, and preprocesses each of them in the same way it does to unknown symbols. The preprocessed models are used until the program exits.

Whenever a new scribble (symbol) is finished by the user, it is handed over by the application to the *ElasticRecognizer*, which performs the recognition and returns the recognized character as well as a list of candidates to the application. The application then displays the result and candidates in the corresponding area of the user interface.

3.4.3 Handling Recognition Errors

Like every other recognition methods, elastic matching is not error free: Recognition errors always happen. In our case the errors can be divided into three categories:

1. Some characters are not distinguishable at all, context information has to be used to tell one from the other. For example “1” (digit one) and “l” (lower case “L”) are exactly the same. In this example, if the “l” is in “21” we might suspect the character is digit one, while in “seal” we would guess it is the lower case “L”.

2. Some characters can be written in an ambiguous way, which makes it error prone for recognition. For example, in Figure 3.4 the character is in a pattern between “a” and “d”.

3. In our application, a timer is used to distinguish the end of one scribble from the start of another one. Sometimes the user may write too fast, resulting in two intended scribbles being considered as one. It is also possible that the user writes too slow such that strokes of the same character are considered as belonging to two different scribbles.

Figure 3.4 Illustration of an ambiguous character

For the *ElasticRecognizer*, the first two categories of recognition errors are handled by allowing the user to select the correct one from the list of candidates.

The way we handle the third kind of error is to establish *undo* and *redo* features in the application. Therefore the user can undo the writings until the incorrect writing has been erased, then the writer can write again. Furthermore, the threshold for the internal timer is configurable, the user can change the default time interval between strokes to a value that makes him or her comfortable. The problem with this solution is that it is not friendly enough to the user. This issue will be discussed in Chapter 7.

Chapter 4 Review of Mathematical Expression Recognition

Like handwriting recognition, structural analysis of two-dimensional patterns also has a long history, it started in the 1960s. However, only very few researchers paid attention to the special problems in recognizing mathematical expressions [8].

In recent years, the widespread use of the Internet makes it an important way for scientific communication. For example, both online digital libraries and distance learning rely on the Internet. A great deal of existing knowledge, including mathematical knowledge, has to be transformed into electronic form in order to be processed by computers and used through the Internet. The need for transforming the mathematical knowledge into electronic form therefore becomes the primary driving force in the research of mathematical expressions recognition. And the demands for efficient ways to input mathematical expression into computers boost the research in this area even further.

In this chapter we review the current status of research in this area, and leave the details of the corresponding part of our application until next chapter.

4.1 The Problems of Mathematical Expressions Recognition

The ability of a recognizer to recognize mathematical expressions depends on the understanding of the two-dimensional structure of the expressions. This is true whether the expression is written on an electronic digitizer or be a scanned image.

OCR is the most popular method for converting text documents (on paper) into electronic form, but this method cannot recognize embedded mathematical expressions in the document. OCR software generally treats mathematical expressions as unrecognizable pictures. Existing on-line handwriting recognition software has the same problem. Even though they can recognize handwriting quite well, they do not understand the meaning of the expression. The reason for this is that mathematical expressions differ greatly from text. Characters in a text are always placed one after another, and the only

thing needed is to recognize the characters sequentially. In contrast, each mathematical expression contains a two-dimensional structure, in which a symbol may be in the same row with other symbols, be above or under other symbols, or be contained in other symbols, *etc*. Different spatial relations generally encode different mathematical meaning. Therefore, recognizing the structural of an mathematical expression is orthogonal to recognizing individual symbols. However structural analysis can provide contextual information as feedback to improve symbol recognition. To be able to automatically recognize mathematical expressions, a recognizer must be able to understand the two-dimensional structure as well as its mathematical meaning.

Based on the above observations, people have put much effort into mathematical expression recognition, both for off-line (typeset and handwriting) and on-line cases.

4.2 Properties of Mathematical Expressions

As mentioned before, in a mathematical expression the symbols are usually arranged in a complex two-dimensional structure. In addition, the symbols with different roles may have different sizes based on the convention in mathematics. In an expression, usually neighboring symbols are grouped locally into sub-expressions. Neighboring sub-expressions can then be grouped into higher level sub-expressions. The grouping goes on and on, and finally we have a hierarchical structure representing the entire expression.

This grouping of mathematical expressions is very complicated and is not easy to be recognized. This is because "*mathematical notation is not formally defined, and is only semi-standardized, allowing many variations and drawing styles*" [4].

A typical mathematical expression contains two types of symbols. The first type includes all *basic* symbols, the second type includes *binding*, *fence* and *operator* symbols. Each type of symbol has its own grouping criteria. Furthermore, there are two types of operator symbols, namely *explicit* and *implicit* operators. What makes recognition even harder is that some of the symbols may have different meanings depending on their context. In the following we discuss the different types of symbols in detail.

4.2.1 Basic Symbols

Although every symbol has its own meaning, it may represent another meaning when it is grouped together with other symbols. This happens quite often to symbols such as digits and Roman letters.

1. **Digits**: When a couple of digits are of the same size and placed adjacent to each other in a horizontal line, they are normally considered as one unit. Otherwise the symbols are not in the same unit. For example, *123* is considered a single unit that represents an integer, while 12^3 contains two units, namely *12* and *3*.
2. **Roman Letters**: Like digits, when a set of characters are placed adjacently in a horizontal line, they might be considered as one unit. This occurs for function names like *sin*, *cos*, *tg* (or *tan*) and *log* (or *ln*), or keywords like *lim*. Therefore it is necessary to check a group of letters for possible function names or keywords before treating them as the multiplication of individual symbols.
3. **Other Symbols**: Most symbols other than Roman letters and digits should be considered as separate units. However there are exceptions. For example, when a dot is adjacent to digits in a horizontal line, the dot and those digits should be grouped together as one unit that represents a floating point number.

4.2.2 Binding, Fence and Operator Symbols

Symbols in this category have special grouping methods, generally they group the symbols adjacent to them into one unit.

1. **Binding Symbols**: Symbols like $\sum$, $\int$, $\sqrt{\ }$, and fraction bar, *etc.* dominate the sub-expressions around them. For example, in $\sum_{i=0}^{k} i$, $\sum$ dominates the other three sub-expressions *k*, *i*, and *i=0*, and groups them together to represent the summation of integers from *1* to *k*.

   The problem is that for complex expressions, it is usually hard to properly decide the relation between binding symbols and their neighboring sub-expressions.
2. **Fence symbols**: Fence symbols include parentheses (“(” and “)”), square brackets (“[” and “]”), *etc.* These symbols always appear in pairs: an open symbol is

matched by a corresponding close symbol. Each matching pair groups the sub-expressions enclosed in them together as one sub-expression. For example, in the expression *a(b+c)*, the pair of parthenses group the three symbols *b*, +, and *c* as one sub-expression.

3. **Operator symbols**: Operator symbols like +, –, × and /, *etc.* dominate their operands and group the operands together into one unit. For example, in $a+b$, + groups its operand a and b together as one sub-expression, then in $\frac{a+b}{c}$, $a+b$ becomes the operand of the fraction bar, which groups $a+b$ and c together as a higher level grouping unit.

   Each operator also has an associated precedence. We say that operator symbols with lower precedence dominate operator symbols with higher precedence when they are lined up. For example, in expression $a+b\times c$ the + dominates the ×, therefore the × groups b and c, and + groups a and $b\times c$.

4.2.3 Explicit and Implicit Operators

Explicit operators are those operator symbols that are visible in the mathematical expression. As introduced above, operator dominance and operator precendence are used in their grouping rule.

An expression could also contain implicit operators, or **spatial operators**. These operators are not visible: they determine the relationships between symbols simply by their relative positions. For example, in a^2, *2* is the superscript of *a*, representing the square of *a*. While in a_2, *2* is the subscript of *a* representing a variable name.

Determining implicit operators is not trivial. The same spatial relation may represent different implicit operators. For example, in expression $a\frac{b}{c}$, a is adjacent to $\frac{b}{c}$, representing the multiplication of a and $\frac{b}{c}$. But in expression $1\frac{2}{3}$, the same spatial relation means the sum of 1 and $\frac{2}{3}$. In another example, the superscript relation in

expression a^2 means the square of a, while in $\int_0^2 xdx$ the superscript relation between *2* and the integral sign indicates *2* to be the upper bound of the integral operation.

4.2.4 Context-Sensitive Roles

Some symbols play different roles in mathematical expresisons depending on their context. For example, a dot can be a decimal point or a multiplication operator depending on its position and neighboring symbols. For another example, in expression $\int xdx$, dx is part of the integral notation, while in $ay + dx$, dx means the multiplication of d and x.

Mathematical notation has many dialects, and it is nearly impossible for a method to cover all the dialects. As a result, almost all systems are based on a subset of the mathematical notations.

4.3 Processes for Mathematical Recognition

The existing mathematical expression recognition systems require a variety of input data. Some systems work on on-line data, *i.e.* expressions as they are written. Other systems work on off-line data. The off-line systems can further be divided into two categories. One category recognizes the scanned image of mathematical expressions only, while the other category allows mathematical expressions to be mixed with text and pictures (these systems are able to locate mathematical expressions from the documents). Despite the big difference between them, both on-line systems and off-line systems have similar recognition processes.

Understanding a mathematical expression typically involves two stages: symbol recognition and structural analysis. Symbol recognition recognizes individual symbols in the expression, while structural analysis recognizes the embedded meanings from the spatial relationship between the symbols, and maintains the relationships in a corresponding tree structure.

Different systems may carry out the above two stages in different orders. The majority of them perform the symbol recognition first, then the structural analysis. But

there are also systems that do the structural analysis before symbol recognition, or perform the two stages concurrently. Some methods even skip the symbol recognition step, assuming that the symbols have already been recognized prefectly. Figure 4.1 illustrates the recognition process of both on-line and off-line system, with symbol recognition stage prior to the structural analysis stage. The following sections are reviews of these two major stages of mathematical expression recognition.

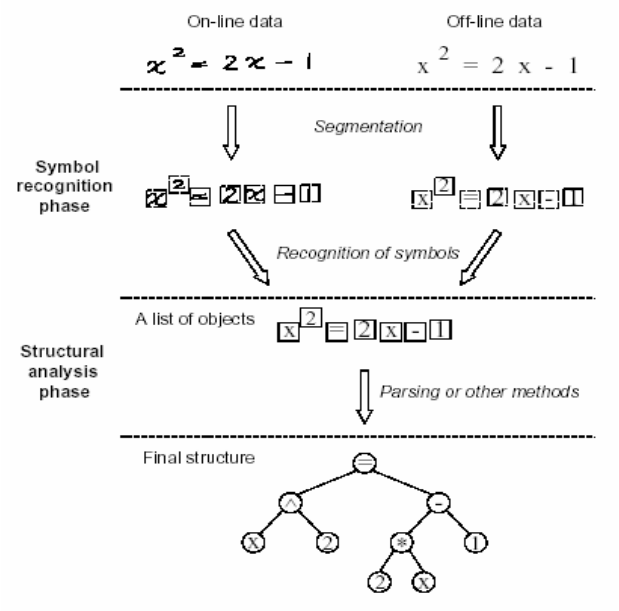


Figure 4.1 Overview of mathematical expression recognition processes. (Chan and Yeung [8])

4.4 Symbol Recognition

Mathematical symbol recognition is exactly the character recognition problem. As discussed in Chapter 3, there has been a long history for research on character recognition, and there are a lot of recognition approaches. Here we will not talk about symbol recognition itself again, but some special issues closely related to symbol recognition: recognizing a large set of symbols, and segmentation of symbols in an expression.

4.4.1 Recognizing Large Sets of Symbols

Most character recognition systems claim to have high recognition rates on typeset or handwritten characters, but their test results are usually based on a limited set of characters, namely, Roman letters, digits, punctuation symbols, and probably some Greek letters. When the set is scaled up to a size that includes all mathematical symbols, these systems probably will not work properly without suitable modification (add more models, more syntax rules, *etc*.).

Currently there are no reported experiments for recognizing large sets of symbols, but theoretically it is possible. This can be demonstrated by the research on recognition of Chinese characters (also known as *Kanji* in Japanese) in both Chinese and Japanese. The research has already gained very impressive results, and there are related commercial products (both OCR and handwriting recognition) available in the market currently with reasonable performance. Since the set of Chinese characters is larger in size than that of the mathematical symbols, the technologies used in these systems will no doubt be very useful for mathematical symbol recognition.

4.4.2 Segmentation of Symbols in Mathematical Expressions

In on-line systems, symbols are written on an electronic tablet stroke by stroke, one symbol after another. The pen-down and pen-up events automatically distinguish one stroke from another, pen-down means the start of a stroke, and the following pen-up indicates the end of that stroke. What needs to be done is to determine whether a stroke should belong to one symbol or another (in multi-stroke mode). The common way is to ask the writer wait a short while after finishing a symbol. The advantage of this way is that the segmentation can be guaranteed to be correct once the threshold of delay between symbols is set properly. In our system, we set up a configurable delay threshold between strokes. If the delay exceeds the threshold then the new stroke is considered as the first stroke of a new symbol. Besides the above method, there are also other ways. For example, Chen and Yin's system [11] uses a much different method. In their system, each stroke is surrounded by a *frame* (actually the same as a bounding box). The X and Y projections of consecutive frames are analyzed for any overlap. Overlapped frames will

be considered as strokes of the same symbol. The system has special rules to unify symbols with disconnected components, such as i, j, ≥, ≤, and =. It also has special rules to handle the square root symbol.

For off-line systems, things are much different. In these systems, an expression comes in as a scanned image, what the system sees is just a two-dimensional array of pixels, there is no information about strokes or symbols at all. The system has to detect and isolate symbols from each other through segmentation, which is not easy to do.

Since the topic of segmentation is not relevant to on-line recognition, we only introduce it briefly. Few methods have been proposed for segmenting symbols up to now, most of them make use of the *X* and *Y* projections to analyze the expression.

Okamoto *et al.* applied "recursive projection-profile cutting" in their system [30,31]. This method cuts the vertical projection profile first, followed by horizontal projection profile cutting on each resulted region. Then this process is applied recursively on all resulted regions until no more cutting is possible. The result is a tree structure, where each node represents an isolated symbol of the expression. The problem with this approach is that it will fail for symbols with disconnected components (i, j, ≥, ≤, and =, *etc.*), and symbols that contain other symbols (*e.g.* the square root).

Ha *et al.* [20] proposed a design similar to the above. In this system, recursive *X* and *Y* cutting is carried on the bounding boxes of primitives. The resulting tree is then traversed to fix errors. Nodes may be split or merged, so nodes in the final tree will be correctly isolated symbols.

Faure and Wang [16] used *X* and *Y* projections to segment handwritten mathematical expressions. Since this method fails for closely-written symbols, symbols with separate components, and symbols that contain other symbols, they used a "mask-removal operation" to fix these errors. This operation first detects "mask symbols" such as square roots, fraction bars, or other long or tall strokes, then remove the mask symbols and apply *X* and *Y* projections again to segment remaining symbols.

From the above we can see that the segmentation problem is easier for on-line recognition systems than for off-line systems. Off-line systems usually have to use

special operations to fix errors of segmentation, but this is not the case for most on-line systems.

4.5 Structural Analysis

Much of the meaning of a mathematical expression is embedded in the spatial relationship between the symbols of the expression. Correctly interpreting the spatial relationships is the prerequisite for a good mathematical expression recognizer, since the final expression tree will be largely based on the spatial relations. Therefore structural analysis, sometimes called symbol-arrangement analysis, is very important in mathematical expression recognition.

Compared with the symbol recognition step, structural analysis seems more important. For example, several systems actually carry spatial analysis before symbol recognition. Some researchers even completely skipped the symbol recognition step and directly perform the structural analysis, assuming that all symbols have already been recognized.

In this section we address the problems of structural analysis first, then introduce the existing works related to this topic.

4.5.1 The Goal of Structural Analysis

Generally speaking, the goal of structural analysis in a mathematical expression recognition system (unless specially specified, we are talking about on-line system only) is to correctly recognize the spatial relationships and implicit operators between symbols. However, how this step should be done depends mainly on the ultimate goal of the entire recognition system, or how much should the system know about mathematics.

4.5.1.1 Systems with No Knowledge about Mathematics

If the main purpose of the system is to recognize and display handwritten mathematical expressions, *i.e.* to be used as a mathematical editor, then the only important information is the two-dimensional arrangement of the symbols. The structural

analyzer only needs to know a very limited set of implicit operators, such as superscript and subscript, there is no need to worry about the mathematical meaning at all. For example, the program does not care about the meaning of the string "*sin*" in the expression.

The on-line mathematical expression recognition system by Chen and Yin [11] is such a system. This system puts a lot of emphasis on stroke grouping (symbol separating) and symbol recognition. Relatively little effort is put into structural analysis. The system recognizes such relationships as *Up*, *Down*, *Superscript*, *Subscript* and *inline* (row), and uses some context information to solve some ambiguity cases, but it does not care about the meaning of the expression. Finally a symbol relation tree is built to display the expression.

4.5.1.2 Systems That Know Mathematics

If the mathematical expression recognition system is expected not only to recognize the expression, but also to be able to know whether the expression is valid or not, then there should be a lot of things for the structural analyzer to do. This means that the system will be largely knowledge driven.

Besides recognizing all symbols, the system has to be able to correctly detect all implicit operators, and check the validity of the expression at the end of the structural analysis. For example, the system will report that the expression $\cos^2 a + \sin^2 = 1$ is invalid since the argument is missing for the function *sin* ($\sin^2 a$ is equivalent to $(\sin a)^2$). If the designers are very ambitious, the system may even have an integrated engine to evaluate the expression and report possible logical errors. For example, the system would report that the proposed equation $1 + 2 = 5$ is invalid since the left hand side is not equal to the right hand side. The result will be an expression tree, ready to be used in either display or mathematical evaluation.

Unfortunately, no such systems are available currently. The reason is that there are too many rules and notational variants involved in mathematics. Developing such a perfect system would be very hard. Even if done, the parsing would be very

computationally expensive. Therefore people compromise between the above two kinds of systems.

4.5.1.3 Systems in Between

This category consists of systems which are a compromise between the above two categories, they have limited knowledge about the common rules and notational variants in mathematics. These systems can usually recognize the implicit operators in an expression, for example, they know that the implicit operator in the expression $1\frac{2}{3}$ is a *plus*, while in $a\frac{b}{c}$ the implicit operator is a *multiplication*. They can also understand function names (*e.g. sin*), keywords (*e.g. lim*), and some context sensitive cases (*e.g. dx* has different meanings in expression $dx+cy$ and expression $\int xdx$). Some systems may also contain limited error detection and correction abilities. However, their knowledge about mathematics will not go beyond these conditions, they may not query an expression such as $1+2=5$. Generally speaking, the structural analysis of these systems results in an expression tree. The tree contains enough information about spatial relationships as well as implicit operators that it can be used as a basis for generating different representations of the expression, such as TEX and MathML.

Several systems have used this approach. Chan and Yeung [5,7] used hierarchical decomposition in their system to parse mathematical expressions. Their system puts a lot of effort on detecting and correcting such syntactic errors as missing function arguments or operands, invalid implicit operators, missing binding or fence symbols, *etc*. Their system can also find some semantic errors, for example, $+ana$ is corrected as $\tan a$. The final result is an expression tree.

Smithies *et al.* [33] developed a handwriting-based equation editor that parses handwriting expressions into a parse tree (expression tree), then the parse tree can be converted into several output formats such as Lisp-like expression, Mathematica and LATEX. This system does not really know much about mathematics, it actually requires the user to realign symbols when the structural analysis result is not correct.

4.5.2 Structural Analysis Problems

How to correctly interpret the spatial relationship between symbols is the major problem of structural analysis. The recognizer has to distinguish all possible relations, such as superscript, subscript, row, above, below, enclose, *etc.* A second aspect of the problem is to correctly detect all implicit operators.

4.5.2.1 Identifying Spatial Relationships

Spatial relationships are critical for the recognition of implicit mathematical operators. Although mathematics is relatively standardized, it allows many variations. There is no specific defintion for the spatial relationships, therefore it is not clear how to identify the relationships.

To make it easy to identify the spatial relationship between symbols, people come up with lots of conventions. For example, the superscript of a symbol should be at the upper-right corner of that symbol, and should have a size smaller than that symbol. It is easy to know what a^2 means. However, the conventions are not formally defined, therefore are not always followed. This is paritcularly true in handwriting recognition systems, where handwritten symbols may be in bad style or erratic, a^2 can be written as a^2, which has no difference between the size of both symbols.

Problems also come from the fact that there is no clear separation of positions between horizontal adjacency and superscript or subscript. For example, in the following five expressions: (1). *a2*, (2). a^2, (3). a^2, (4). a^2, (5). a^2, the spatial relationships between *a* and *2* changed from row to superscript, however the relationship in expression (2), (3), and (4) are hard to determine.

In the Roman alphabet, some letters like *b*, *d*, *h* and *t* contain *ascenders* (an ascender is the portion of a lowercase letter that rises above the main body of the letter). Some letters like *g*, *y*, *j*, *q*, and *p* contain *descenders* (a descender is the portion of a lowercase letter that falls below the baseline). The other letters contain no ascenders or descenders. Ascenders and descenders are not considered to be the main body of a letter. When identifying the spatial relationship between symbols. This issue has to be

considered, otherwise the relationship may be identified incorrectly. For example, the expression in Figure 4.2 can be correctly interpreted as b^p only if we exclude the ascender and descender of the symbols in the spatial relationship identification process.

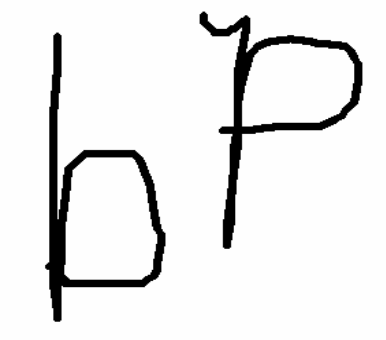


Figure 4.2 Remove ascenders and descenders is essential for structural analysis.

4.5.2.2 Identifying Implicit Operators

Implicit operators, like superscript, subscript, multiplication, matrix, and so on, are indicated by the two-dimensional layout of symbols. The spatial relationship between two symbols, however, is not necessarily enough to determine the implicit operator. For example, in Figure 4.3, symbol *y* could be the superscript of symbol *x* (x^y, an implicit operator *superscript* exists between *x* and *y*), but it is also possible that *x* and *y* form that alignment coincidentally ($z_x y$, no implicit operator between *x* and *y*, instead there exists an impilcit multiplication operator between z_x and *y*). The reason is simple: the baseline of the expression can not be decided locally. We must obtain the baseline globally and decide the spatial relationship based on that.

In some mathematical expressions, presuperscripts and presubscripts are also used. This makes it more complicated. For example, in expression $a_n C_m$, *n* may be a presubscript of *C* instead of the subscript of *a*. In that case there is no implicit operator between *a* and *n*, but an implicit operator between *a* and ${}_nC_m$. Usually these problems are connected with special binding symbols like $\prod$, *C* (combination), *etc.* Therefore when

these symbols appear in an expression, the possibilities of pre-superscript and pre-subscript have to be checked.

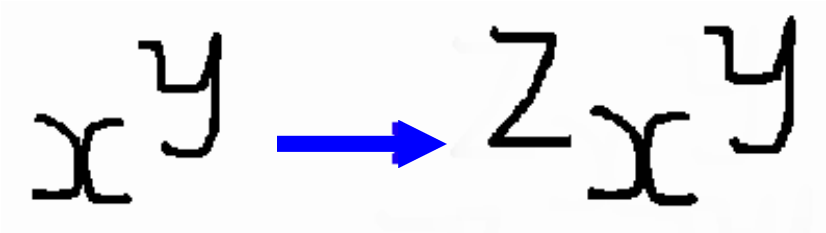


Figure 4.3 Implicit operators should be determined globally

4.5.3 Methods for Structural Analysis

Existing mathematical expression recognition systems use a variety of approaches to analyze the two-dimensional arrangement of symbols. Blostein and Grbavec [4] surveyed the commonly used methods and divide them into four groups, namely, syntactic methods, projection-profile cutting, graph-rewriting, and procedurally-coded rules. In this part we introduce the existing structural analysis methods, following their categorization with an additional part for the methods not included in the four groups.

4.5.3.1 Syntactic Methods

This group claims the most number of structural analysis methods. All methods in this group have used some syntactic grammars.

Anderson [1] is one of the earliest person who used grammars to recognize mathematical expressions. His system assumes that input symbols have already been recognized, and symbols composing integers or real numbers have already been grouped together. Using a top-down parsing scheme, his system starts with one syntactic goal and all symbols of the expression. Each grammar rule starts with a set of symbols and a syntactic goal, and the rule specifies how to partition the set of symbols into several subsets, each with its own sub-goal. If a partition fails, the grammar rule reports failure, causing the parser to try another rule. On the contrary, a successful partition will generate

several sub-goals. The algorithm keeps going until all the sub-goals are satisfied or all possibilities have failed. The main drawback of this method is its inefficiency.

Beláid and Haton [3] came up with a system that uses both a top-down and a bottom-up parser to recognize some simple mathematical expressions. The top-down parser works the same way as Anderson's system, it divides expressions into sub-expressions. The bottom-up parser is used later to combine sub-structures into larger structures. Compared with Anderson's system, this design is more concise.

Chou [12] used a two-dimensional stochastic context-free grammar to recognize noisy typeset mathematical equations. Each production rule of the grammar has an associated probability, and a particular parse tree has a probability that is the product of the probability of all involved production rules. Each production is allowed to use either vertical or horizontal concatenation. The most likely parse is calculated by a dynamic programming algorithm. This approach has a time complexity of $O(n^3)$.

Chang [9] used *structure specification schemes* to analyze the structures of mathematical expressions. Like Anderson, his system assumes that symbols have already been recognized. The structural analysis algorithm of this system put restrictions on the syntactic approach, therefore it is more efficient than Chou's method. The recognition time is $O(n^2)$.

Lavirotte and Pottier [23] used a graph grammar to parse mathematical expressions in documents. The system first extracts expressions from the document and builds a graph, then it uses a graph grammar to parse and generate graphs. Each graph grammar contains a start graph and a set of production rules, the rules are responsible for replacing a matched sub-graph by another one. The advantage of this system is that no backtracking is needed in parsing.

Chan and Yeung [5] used *hierarchical decomposition parsing* to interpret the structure of mathematical expressions. The main techniques of this approach are left-factoring, binding symbol preprocessing, and hierarchical decomposition, for improving efficiency.

4.5.3.2 Projection-profile Cutting

A couple of researchers have used *X*-*Y* projection profile cutting to segment the image of expressions, and build a spatial relationship tree based on the segmentation. These methods usually build a symbol layout tree before recognizing the symbols, and no parsing is involved. However, there are some cases that *X*-*Y* projection cutting can not handle, for example, a square root. Special care must be taken in these cases. Futhermore, in many cases, spatial relationship alone can not provide the correct grouping, especially for context-sensitive relationships. Therefore it is usually necessary to fix errors after the symbol recognition.

Okamoto *et al.*[30,31] applied recursive projection-profile cutting on mathematical expressions. A symbol layout tree is obtained prior to the recognition of the symbols. Special care is taken to handle symbols like square root, which can not be handled by the projection profile cut.

Ha *et al.* [20] applied recursive *X* and *Y* cutting on the bounding box attributes of primitives. The initial expression tree is build in top-down fashion, then it is traversed bottom-up to fix errors.

Wang and Faure [37] proposed a method to automatically decompose expressions into blocks (bounding boxes) by *X* and *Y* projections and label the spatial relationships between symbols. This method analyzes spatial relationship without knowing the identity of symbols, therefore it still works even when some symbols can not be recognized.

4.5.3.3 Graph-rewriting

Graph rewriting is a general computational technique that uses an attributed graph to represent information and uses graph-rewriting rules to update the graph. The main advantage of this approach is that it offers flexible formalism with a strong theoretical foundation for manipulating two-dimensional patterns. Grbavec and Blostein [19] applied this technique on mathematical expression recognition.

Their system assumes that all symbols of the expression have already been recognized and an initial edgeless graph is built upon that. Each node of the graph

represents a symbol of the expression, with an attribute representing the spatial coordinates of that symbol. The role of graph rewriting rules is to add edges that represent potential spatial relationships. Corresponding operator precendences are stored as edge attributes in order to determine how to group symbols into sub-expressions. When a sub-expression is recognized, it is corresponding sub-graph is replaced by a single node representing the sub-expression. The final graph will be a single node representing the expression.

The recognition process involved in this algorithm contains four phases.

1. **Build:** Add edges to represent spatial relationships between symbols.
2. **Constrain:** Use domain knowledge (knowledge of notational conventions) to remove contradictions and resolve ambiguities.
3. **Rank:** Group symbols into sub-expressions according to operator precedence.
4. **Incorporate:** Interprets sub-expressions by replacing a recognized sub-expression with a single node.

This system consists around 60 graph-rewriting rules and provides good recognition results on a small set of test expressions. However the execution is very slow and it could not handle symbol recognition errors.

4.5.3.4 Procedurally-Coded Mathematical Rules

There are also some mathematical expression recognition systems that use procedural code to embody the syntax for structural analysis.

In the system developed by Lee and Lee [24,25], an expression is represented by a list of symbols in random order after symbol recognition. Each symbol is represented by a bounding box. Then procedural code is used to recognize appropriate symbol groups, or sub-expressions. Each recognized sub-expression is replaced by a new bounding box. The final result is one bounding box representing the entire expression.

Okamoto *et al.* [30,31] used recursive *X-Y* projection profile cutting and an extensive array of procedurally-coded recognition rules in the structural analysis. The main role of the procedurally-coded rules is to correct particular recognition errors.

Compared with syntactic approaches, procedurally-coded methods usually have the benefit of fast execution since no expensive parsing is needed. Also procedurally-coded methods may be recursively coded more easily. A main disadvantage is that procedural code is difficult to maintain or scale up since the implicit syntax rules are hard-coded in the recognizer.

4.5.3.5 Other Approaches

Besides the above four groups of methods, other approaches using quite different structural analysis techniques exist which add new ideas to the mathematical expression recognition world.

Fukuda *et al.* [18] introduced the concept of *Mathematical Element* (ME). Each ME may contain a set of symbols. Mutual spatial relationships between symbols are associated with penalty functions. The algorithm calculates the penalty values for different possible configurations of MEs and chooses the minimal one as the final result.

Miller and Viola [28] used convex hulls for grouping symbols and apply A* search to search the best possible interpretation of an expression. Convex hull limits the growth of the number of possible parses of the expression, resulting in an efficient algorithm.

4.5.4 On-line Approaches vs Off-line Approaches

One of the major differences between on-line and off-line mathematical expression recognition systems is that in on-line systems, dynamic information such as timing information can be used. As discussed before, dynamic information helps to distinguish symbols from each other, therefore making the symbol recognition step easier.

At the structural analysis stage, there seems no big difference between on-line and off-line systems. Of the variety of approaches we have introduced, there are no on-line-recognition-specific approaches. On the other hand, approaches for off-line systems can always be used by on-line systems, if necessary we can just ignore the dynamic information. However, in general people will not do this. For example, it will not be a good idea to use "*X*-*Y* projection profile cutting" in an on-line system.

Chapter 5 Structural Analysis Design

In this chapter we introduce our structural analysis method in detail. This includes a brief discussion about the goal of our structural analysis procedure and the selection of algorithms followed by detail of the implementation.

5.1 Overview

5.1.1 The Goal of Our Project

The goal of our program is to interpret the meaning of a mathematical expression, and generate the corresponding presentation MathML code. The generated MathML code is expected to be used as input for future pocket PC computer algebra systems.

In presentation MathML, all implicit operators of a mathematical expression are explicitly represented in the same way as other symbols of the expression. It is essential to recognize the implicit operators correctly, otherwise the generated MathML code will be incorrect. On the other hand, the MathML representation of an expression contains enough information for a computer algebra system to interpret the expression. As soon as the MathML code can correctly represent the expression, the computer algebra system is able to check the validity as well as evaluate the expression for us.

Therefore, in this thesis, we decided to make the structural analysis process be a compromise between the approach taken in section 4.5.1.1 and that taken in section 4.5.1.2. It will have enough knowledge to recognize spatial relationships as well as implicit operators, but will not have to worry about the semantics.

5.1.2 Structural Analysis Methods – Grammars vs Procedural Code

We introduced in Chapter 4 four main approaches for structural analysis, *i.e.* syntactic methods, *X*-*Y* projection-profile cutting (not suitable for on-line systems), graph

rewriting, and procedural code. Among these, we prefer the procedure code approaches to syntactic methods and graph rewriting methods. We explain below why we made this decision.

Syntactic methods and graph rewriting methods represent the majority of structural analysis approaches, and they have a good recognition rate. However, all these methods are good only for static expressions. They begin with the finished mathematical expression and process all symbols of the expression in batch mode. In our project, what we expect is that whenever a symbol is finished, it will be added to the expression tree, *i.e.* process the expression on the fly. This means that the expression tree will keep changing dynamically whenever there are new symbols added to the expression, until the entire expression is finished. In order to use syntactic or graph rewriting methods, we must start parsing the expression anew for every new symbol added to the expression. This may dramatically slow down the whole application due to the fact that grammars are computationally expensive. On the contrary, procedural code executes much faster, and may be coded recursively with ease. This therefore meets our requirements for a PDA better.

5.1.3 Design for the Structural Analysis

To correctly analyze the spatial relations in our design, the program requires that symbols be separated spatially from each other when written. We also assume that symbols are written from left to right, except for parentheses, which can be written after the sub-expression they enclose.

The structural analysis process is based on the bounding boxes of symbols in an expression. The recognized spatial relationships are represented in an n-ary tree structure, in which each node represents either a symbol or an implicit operator.

Whenever a new symbol is recognized, the application will hand it over to the structural analyzer to be analyzed. The analysis is carried out in the following steps: First of all, the program will traverse the expression tree and find the node closest in distance to the new symbol. Then the program will follow the path from the node to its ancestor nodes, then analyze the relationship between the nodes and the new symbol, in order to

try to find out the most appropriate place for it in the expression tree. Next the program will insert it in the expression tree. At the same time it adds the implicit operator involved in the tree if necessary. Every time a new node is inserted into or removed from the tree during tree rearrangement, the bounding boxes of all its ancestor nodes will be updated to reflect the change. This process is repeated until the user finishes the entire expression.

In the next sections we detail our implementation of the structural analysis in the following order:

- the organization of the expression tree.
- the way to find the right place for a symbol in the tree.
- the way to detect proper implicit operators.

5.2 The Expression Tree

The expression tree is the data structure that keeps all the information about the spatial relationships among the symbols. Each node of the tree represents either a symbol or an implicit operator. In the expression tree, leaf nodes contain no spatial relationships, each node represents its own identity, on the contrary, each inner node of the tree contains the spatial relationship between all its subtrees.

Whenever the structural analyzer receives a recognized scribble from the *ElasticRecognizer*, it creates an expression tree node for that scribble. By design, each tree node contains a set of attributes: a label, a scribble, a bounding box, a flag, and a list of its children nodes. We describe these attributes below:

- The **label** represents the identity of the node. For a leaf node, if this node represents an actual symbol in the expression then the label is the same as the recognized symbol. Otherwise, the label would be the name of the implicit operator that the node represents (in the fashion of presentation MathML). For example, the node that represents digit *1* will have a label "*1*", while a node representing the implicit multiplication will have the label "*⁢*".

  For inner nodes of the expression tree, it is more complicated. The label could be the name of an implicit operator (*e.g.* "*msup*" is the label for a node that

represent a superscript relationship), or the name of an explicit operator (*e.g.* "*mfrac*" is the label for a node that represent the division operator), or sometimes the concatenation of its children nodes labels (*e.g.* "*12*" is the label of a node whose two children nodes represent *1* and *2* respectively).

To match the presentation MathML syntax, a couple of rules have been applied when creating a node from a given scribble. For example, the label for fraction bar is "*mfrac*", the label for open parenthesis is "*mfenced*", and the label for the node representing inline relationships is "*mrow*".

- The **scribble** contains all the strokes contained in this symbol, as well as its bounding box and identity. Initially, the symbol is unknown, so its identity is also unknown, the handwriting recognizer then recognizes the symbol, sets its identity, and passes the modified scribble to the structural analyzer.

  In order to preserve the spatial layout of each symbol, in the symbol recognition step all preprocessing operations must be performed on a copy of the scribble, thus leaving the scribble intact. Therefore, when a scribble is passed to the structural analysis step, it contains exactly the same spatial layout information as before, along with a recognized identity.

  For the nodes of the expression tree that represent implicit operators, the scribble attributes are absent since they do not exist in the list of user input symbols.

- The **bounding box** of an expression tree node contains the spatial information of the subtree (sub-expression) that the node represents. For any leaf node, its bounding box is the same as the bounding box of the scribble. We note that bounding boxes of implicit operators do not make sense, and therefore the attribute does not exist. For an internal node, its bounding box is the aggregation of all its children nodes' bounding boxes. By providing both the scribble and bounding box attribute to the node, the application is able to keep the two-dimensional information for both individual symbols and all possible sub-expressions of an expression.

- The **flag** of a node is used to simplify the translation from an expression tree to MathML code when structural analysis is done. By default the flag of each node is true, but in some situations the flag will be modified so that the translation process does not have to consider the corresponding node. For example, children nodes 1, 2 and 3 of the node “123” should not be considered individually, so the flag of these three nodes should be set to false. The program will only consider the node “123” in the translation.
- The **children node list** holds all the children nodes for a node. Due to the fact that some spatial relationship may involve more than two symbols (*e.g.* the example above), the list has no size limit. This attribute is implemented in the fashion of a doubly linked list.

5.3 Locate the Nearest Neighbor (NN) Node

In order to determine the relationship between a new node and other nodes, the first task is to find the nearest node in the expression to the new node. In the following we shall describe the nearest neighbor node as the NN node.

The relationship between the new node and its nearest neighbor is the starting point for the actual spatial relationship recognition procedure. This is because the new node either forms a new sub-expression with its nearest neighbor (*i.e.* they belong to the same subtree), or it belongs to a sub-expression that contains its nearest neighbor as a descendant (*i.e.* the node is in the same subtree with an ancestor node of its nearest neighbor node).

The way to do this is to traverse the entire expression tree, ignoring all nodes that represent implicit operators and calculating their distance from the new node. The node with the minimum distance is selected as the nearest neighbor of the new node. The distance being calculated is the square of the Euclidean distance between the center of the symbols.

5.4 Locating the Correct Position of a New Node

Once the NN node of the new node is found, it is used as the entry point for locating the correct position of the new node in the expression tree. This is the key step in structural analysis.

In a mathematical expression, the most common spatial relationship between symbols are *row* (or *inline*), *over*, *under*, *superscript*, *subscript* and *include*. Among them the *inline* relation is the most common one. There are also other relations such as *presuperscript* and *presubscript*, but they are only used infrequently, therefore we shall ignore them for now.

Determining the right place in the expression tree for a node is not a trivial thing. Usually the information provided by the relation between the node and its NN node is far from enough. The ancestor nodes of the NN node should also be taken into consideration. For example, in the expression a^2b the NN node of "b" is "2", but there is no valid relation between them, the actual relation is between "b" and "a^2" (the parent node of "2").

In our algorithm, which locates the correct position of a node in an expression tree, firstly we check whether the node is in the same row as a sub-expression (of course the sub-expression contains the NN node, this is also true for the following cases); If the answer is "no", check whether the node is in the same column as a sub-expression; If the answer is still "no", then check whether the node is a superscript or subscript of any sub-expression. Whenever such a sub-expression is located, we can perform necessary modification on the tree to attach the node to it.

5.5 Approach for Direction Determination

Correctly determining the position of one symbol to another is critical for correctly interpreting their spatial relationship. This is because the entire analysis process is based on it. However, as discussed in section 4.5.2.1, spatial relations may not have a direct correspondence with the required symbol. For example, there is no clear separation of

positions between horizontal adjacency and superscript or subscript relationships. Conventions in the mathematical world may not be much help either, sometimes it is harder to follow the conventions in handwriting than in typesetting. Superscript, for example, is usually written in a smaller size in typesetting, but this is not necessarily the case in handwriting. Users may find it hard to do this, especially when there are several levels of superscripts as in expression a^{b^c} , also the symbol recognition rate will be heavily affected for very small symbols. With these issues in mind, we generally would ignore such conventions, particularly conventions about symbol size, only in some special situations do we consider them.

Several things are done to make direction determination easy:

Firstly, we take *ascender* and *descender* into account. Ascender or descender does not belong to the main body of a symbol and should be excluded. Strictly speaking, finding the ascender or descender includes the partition of the structure of a symbol, for example the loop strucutre of symbol *"b"* has to be recognized in order to get rid of the ascender (the loop is the main body of "*b*"). Since our design currently does not have the symbol structure analysis feature, we use a relatively naïve approach, where we assume that the ascender or descender takes a fixed portion of a symbol. In the symbols "*b*" and "*p*", *etc.* the ascenders or descenders are fixed as 40 percent of the symbol's height. When calculating directions, only the part corresponding to the main body of a symbol's bounding box is used.

Secondly, we set up some thresholds to help determine the relative position of bounding boxes.

- If the difference between the *Y* coordinate of the bounding boxes' center points is less than one third of the larger box's height, we consider the two boxes to be in a row. Here the *Y* coordinate of a box's center point serves as the box's base line.
- If the *X*-projection of one bounding box is inside that of the other bounding box, the two boxes are considered to in a column (*over* or *under*).

- If the two bounding boxes are neither in a row nor in a column, then we calculate the angle between them. Based on the angle, it can be determined whether the direction between the boxes is superscript, or subscript, *etc.*

Now that we have introduced how the direction between bounding boxes is calculated, we can begin to introduce the way to find the correct destination for a new node in the expression tree. Some special cases must be checked first, then we must check the row, column, superscript and subscript cases, in that order. Presuperscript and presubscript relations are relative rare in mathematical expressions. As mentioned early in this section, they will be ignored for now.

5.6 Special Cases

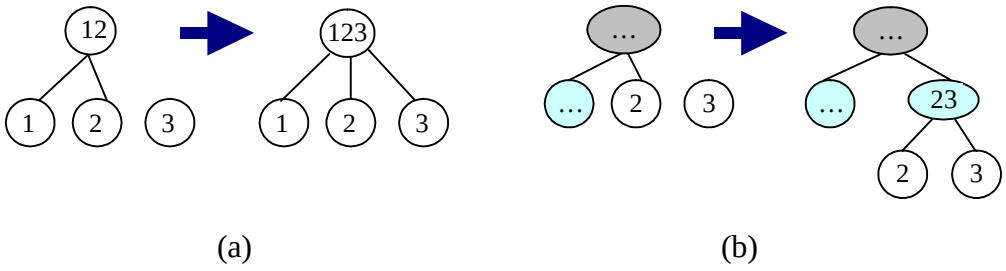
There are a couple of cases that should be taken care of specially. These cases are either very straight forward and therefore can be done directly, or will be interpreted incorrectly if treated normally. Below are some of the special cases. Note that in case 2 to 5, a new node is always in a row with its NN node and is to the right of the NN node.

1. If a symbol is the first of a expression, its corresponding node should be set as the root node of the expression tree, and the analysis is done.
2. If the NN node is the open parenthesis “(”, then attach the new node as a child node to the NN node.
3. If both nodes are digits, or NN node is a digit while the new node is a dot (‘.’), then check the parent node of the NN node.

   If the parent node’s label is a number, it means that the new node is part of that number. So attach the new node as the right most child of the parent node, set its flag to be false, at the same time change the label of the parent to be the concatenation of its original label and the new node’s label. (Figure 5.1a)

   If the parent does not exist, or its label is not a number, it means that the NN node is the first digit of a multi-digit number. A node is created as the parent node of both nodes, it becomes the root of the tree, or it takes the original place of the

NN node. Its label is the concatenation of both nodes, and it contains no scribble. Both of its children will have the flag attribute as false. (Figure 5.1b)



(a) (b)

Figure 5.1 An illustration of forming a number from digit nodes.
When a node is in a row with its NN node and both nodes are digits, they form a number. In both diagrams the node "3" is the new node, "2" is its NN node.
(a) NN node is part of a multi-digit number, attach the new node to its parent.
(b) NN node is a one-digit number, create a multi-digit number from both nodes.

4. If both nodes are Roman letters, or the new node is a letter while the NN node is an **alphabet string** (a string of Roman letters, excluding the special node names such as *mrow*, *msup*, *msub*, *mfenced*, *mfrac*, *etc.*), try to check the parent of the NN node.

   If the parent node does not exist or its label is not an alphabet string, NN node represents the first letter of a string. What we must do is make a new node with both nodes as its children then set its label as the concatenation of the children nodes. This new node will either be the new root of the expression tree, or it will take the original place of the NN node. This operation is very similar to that illustrated in Figure 5.1(b).

   If on the contrary the parent's label **is** an alphabet string, one may be sure that the NN node is part of an alphabet string. The program will process this string. If it does not contain any function name, then attach the new node as the right most child of the parent node. Otherwise, if the parent node contains a function name, then the parent node will be split as follows: every child node representing a letter prior to the function name will be separated from the parent node and becomes its sibling node. Between each of these nodes there is a "&ImplicitTimes;" node representing the implicit multiplication relation between them. What is left of the parent node represents a function name, and the new node will be treated as the

argument of that function. Figure 5.2 illustrates how the parent node “abcos” is split as the implicit multiplication of “a”, “b” and “cos” nodes.

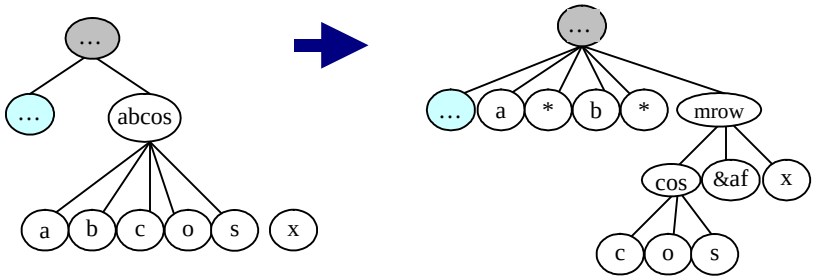


Figure 5.2 An illustration of splitting a node containing function names. In this diagram, the node “x” is the new node, the node “s” is its NN node, the node “*” represents implicit multiplication, and the node “&af” represents “⁡”

5. When the NN node is an alphabet string and the new node is a digit, a fraction bar or an open parenthesis, we must make sure that its NN node is not part of a function name. Otherwise a node split operation must be done on the NN node’s parent as in Figure 5.2, and the new node will serve as the argument of that function. This situation can be found in the sample expressions $\sin 2$, $\sin \frac{a}{b}$, and $\sin(a+b)$.

6. When a dot (‘.’) is at the subscript position of a digit, it can be sure that the dot represents a decimal point. This case is handled exactly the same as in case 3.

7. Special operation is also prepared to handle the situation when an open parenthesis is written later than the sub-expression it encloses.

5.7 Row Direction Check

Once all special conditions have been handled separately, we can make sure that our general approaches will handle other non-special conditions properly.

Among the row, column, superscript and subscript relations, row direction is the dominate one in a mathematical expression. More importantly, this relation can usually involve long range grouping, a node can be grouped with a node that is spatially far away, no matter what relation it has with its NN node. Column relation also has this

feature. So we start with these two directions first, only when the possibilities of long range groupings have been excluded can we consider short range groupings.

The two-dimensional layout of a mathematical expression forms a complicated hierarchical structure, in which the relation between a symbol (and its nearest neighbor in most cases) can not reflect the correct grouping. Quite often a new node is not in a row with its NN node but in fact it is in a row with a sub-expression that contains the NN node. For example, in the expression "a^2+", "+" is not the subscript of "2", instead it is in a row with the sub-expression "a^2". Even when the new node and its NN node are in a row they may not belong to the same place in the expression tree, especially when parentheses are involved. In Figure 5.3(a) the node "+" and "*b*" belong to the same sub-expression "mfenced", while in Figure 5.3(b) the node "+" and ")" belong to different sub-expressions.

We see from the above that the main purpose of the row direction check is to distinguish these different conditions and determine the correct sub-expression to which a node belongs.

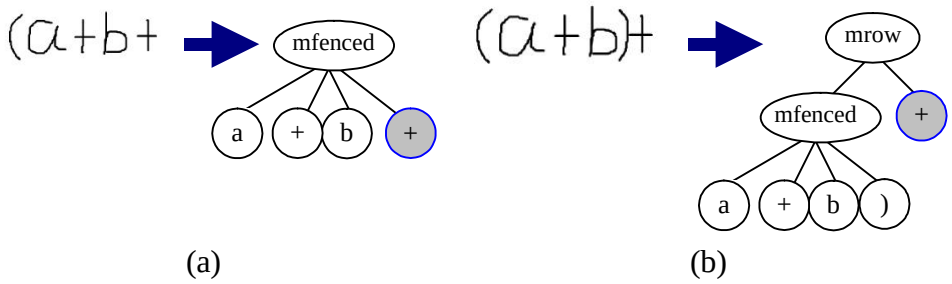


Figure 5.3 Different groupings for nodes in a row.
(a) A node and its NN node are at the same location in the expression tree.
(b) A node and its NN node are at different locations in the expression tree.

5.7.1 The Algorithm

The key operation in this procedure is a routine called *RowParent*. If finds the most applicable node in the expression tree to be the parent node, or occasionally sibling node of a given node.

Below is the algorithm for the routine *RowParent*:

```
Routine RowParent{
  newnode = the new node;
  parent = NN node;
  target = null;
  while(parent != null){
      if(newnode is in a row with parent){
         if(parent is a '(' without a matching ')')
              return parent;
         else
              target = parent;
      }
      parent = the parent node of parent;
  }
  return target;
}
```

This algorithm starts with a given node *newnode* and its NN node *parent*. It first checks their relationship, then checks the relationship between *newnode* and its grandparent node, then its great grandparent node, and so on, until it exits when some conditions are matched, or the root node has been reached. Each time a row relation is found, the algorithm will update the *target* so that it is always the top most node that is in a row with the *newnode*. During this bottom-up procedure, whenever the *newnode* is in a row with an open parenthesis that has no matching close parenthesis, it can be sure that *newnode* is enclosed in the parenthesis, therefore that node will be returned. Otherwise whatever node the *target* refers to will be returned at exit, it could be a node or *null* if in the case that there is no node in the tree in the same row as *newnode*.

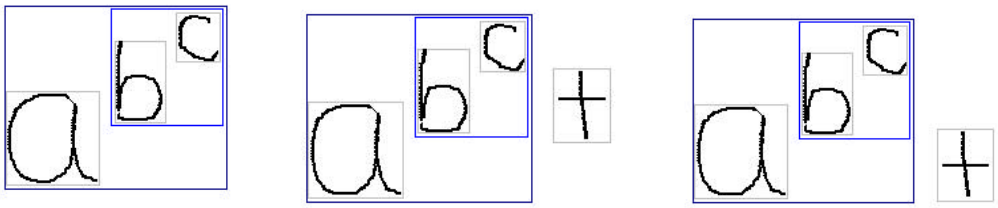
Based on the value returned by the *RowParent* routine, *target*, we may proceed differently. If the value is *null*, *newnode* is not in a row with any node in the expression tree, so it will be passed to the other direction check method that follows. On the other hand, if any node is returned, this node is either the parent node of *newnode* or the sibling node of *newnode*.

In order to know whether *target* should be the parent node or a sibling node of *newnode*, we must check whether *target* is a node that represents a row relation, *e.g.* mrow, mfenced, *etc*. If it is, then it should be the parent node, otherwise a new node has to be created to be the parent node of both nodes.

Both *mrow* and *mfenced* (open parenthesis) nodes represent row relationships, so if *target* is any one of them, *newnode* will be attached to it as the rightmost child node. Besides, if *target* is an open parenthesis and *newnode* is a close parenthesis, the flag of *newnode* should be set to false. On the other hand, if the label of *target* is not "*mrow*", we must create an *mrow* node which has both *target* and *newnode* as children nodes (in that order). It will take the place that *target* had taken before.

5.7.2 Refinement of Bounding Box Operations

The above *RowParent* routine intensively uses the relation between bounding boxes. The bounding box of a sub-expression is the union of all children nodes' bounding boxes. The entire expression's bounding box is the union of all the sub-expression's bounding boxes, as illustrated in Figure 5.4(a).



(a) (b) (c)

Figure 5.4 Cases that bounding boxes fail to correctly reflect the relationship between sub-expressions. We consider some commonly occurring situations.
(a) The bounding box hierarchy of the expression a^{b^c}.
(b) Introduce row relation between '+' and b^c incorrectly considered as row relation between '+' and a^{b^c}.
(c) Introduce row relation between '+' and a^{b^c} incorrectly considered as subscript relation between a^{b^c} and '+'.

One problem the *RowParent* routine may encounter is that when a bounding box becomes large, the baseline of the expression will change (the *Y* coordinate of the centroid serves as the baseline), this causes the routine to be error prone. For example, in Figure 5.4(b) the '+' is in a row with the sub-expression b^c, but its bounding box is actually in a row with the entire expression. On the other hand, in Figure 5.4(c) the '+' is in a row with the entire expression, however its bounding box is actually in the subscript

position of that of the entire expression. Both cases result in an incorrect relationship between bounding boxes.

To solve this problem, we came up with a method to update the bounding box of expression tree nodes conditionally:

1. When the node represents a row relation, *i.e.* it is an *mrow* or *mfenced* node, its bounding box will be updated in this way: horizontally, it is the union of all its children nodes' bounding boxes. Vertically, it is the minimum of the *Y* coordinates of all the children nodes, or 10 pixels, if that is larger. The lower bound of 10 pixels is used to handle situations like the bounding box of the minus sign and the fraction bar. In these cases the height is very small and may affect the accuracy of relation detection.
2. If the node represents column relation, such as *mover*, *munder* and *munderover*, its bounding box will simply be the union of all the children nodes' bounding boxes.
3. If the node represents the superscript or subscript relation, it contains two children nodes where the second child is the superscript or subscript. Its bounding box is updated in a way that the *Y* coordinates will be the same as the *Y* coordinates of the bounding box of the first child node, while the *X* coordinates will be the union of both children's bounding box's *X* coordinates. *i.e.* the superscript and subscript children nodes only contribute their width but not their height to their parent node's bounding box.

By using conditional bounding box updating, each subtree of the entire expression tree may have a different way of updating its bounding box in order to keep the bounding box information meaningful for the sub-expression locally. At the same time solves the problem discussed in Figure 5.4. Figure 5.5 shows the bounding box hierarchy of an expression using conditional bounding box updating.

This *RowParent* algorithm we introduced here works quite well in practice. Let's look at a sample expression "$a^2+((b))+c$", whose tree structure is illustrated in Figure 5.6. When "*2*" is *newnode*, *RowParent* returns *null* indicate that it is not in a row relation with "*a*". When the *newnode* is any of the close parentheses, *RowParent* returns the

mfenced node that contains the matching open parenthesis. In other cases *RowParent* always returns the appropriate parent node.

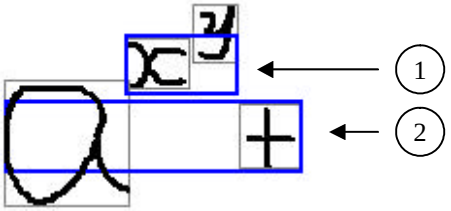


Figure 5.5 The bounding box hierarchy of the expression a^{x^y} by conditional bounding box updating.
Arrow 1 is the bounding box for sub-expression x^y; Arrow 2 is the bounding box of the entire expression.

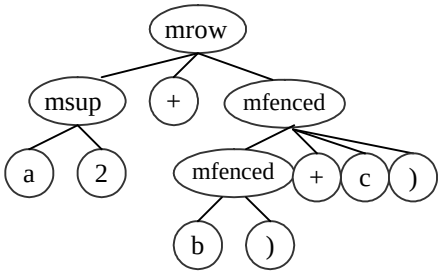


Figure 5.6 The expression tree of expression a^2+((*b*)+c)

5.8 Column Direction Check

If the *RowParent* routine returns *null* for an input node in the above row direction check then that node is not in a row with any other sub-expressions. In this case we consider other possible directions. Amonst these column direction also involve long range grouping. We therefore need to consider it before considering the superscript and subscript relations.

The column direction check procedure tries to determine whether a given node is in a column relation with any nodes in the expression tree. If such a node is found, the given node will be inserted into the tree to form some kind of grouping with the determined

node. Otherwise, this procedure fails and the given node will be considered for other possible relations, namely superscript and subscript.

The idea involved in this procedure is similar to that of the row relation check, however there are more conditions to consider and the involved node rearrangements are more complicated.

5.8.1 Finding the Relevant Parent with the *ColParent* Routine

The key operation is a routine called *ColParent*. This is intended to find the applicable parent node in the column direction for a given node. However the node it returns is not necessarily the parent node of the given node, the reverse is also possible, depending on the symantics of both the given node and the returned node. Below is the pseudo code for the algorithm:

```
Routine ColParent{
    newnode = the new node;
    parent = NN node;
    target = null;
    if(parent is a fraction bar)
       return parent;
    if(newnode is a fraction bar){
       while(parent != null){
         if(newnode is in a column with parent){
           if(parent is a fraction bar and is wider
              than newnode)
                return target;
           target = parent;
         }
         parent = the parent node of parent;
       }
    }
    else{
       while(parent!=null){
          if(newnode is in a column with parent){
             if(parent is fraction bar, integral or
                summation sign)
                  return parent;
             target = parent;
          }
          parent = the parent node of parent;
       }
    }
    return target;
}
```

In this algorithm, initially only a node *newnode* and its NN node are available. If the NN node is a fraction bar then we consider it to be the correct parent node for *newnode*. This is based on the fact that when people write a fraction, they always finish the fraction before they move over to another sub-expression. If the fraction bar is written first, then the numerator or denominator will be written immediately after it, the order does not matter, the fraction bar is always the parent node in the sub-expression.

If the NN node is not a fraction bar, it is necessary to follow up the path from the NN node to its ancestor nodes and find the node that is most appropriate to be the parent node of the *newnode*. This will be done in different ways depending on the *newnode*.

- If *newnode* is a fraction bar, check whether it is in the same column with the parent node of the NN node, or recursively one of its ancestors.

  At each time a match is found, we need to check this node. If it is a fraction bar that is wider then the fraction bar of the *newnode*, it will be returned by the *ColParent* routine. Otherwise the *target* node will be updated to refer to this node. Since this is a bottom-up approach, *target* always refers to the highest level node that is in column relation with the *newnode*.

  Here we take the assumption that when two fraction bars are in a column, the wider one is the parent node of the other. Otherwise it will occasionally be very hard to tell their relationship, especially when there are no other sub-expression that may be refered to detect the correct baseline. For example, the only way to distinguish expression $\frac{a}{\frac{b}{c}}$ and $\frac{\frac{a}{b}}{c}$ is to compare the width of the two fraction bars in each expression. Note that in handwriting math the size of symbols will not be of much help.

- If *newnode* is not a fraction bar, the check starts with the NN node and follow up the path to its ancestor nodes, in exactly the same order as above. Whenever a match is found, if that node is either a fraction bar, or an integral sign or summation, the node is considered to be the parent node of *newnode* and is returned from the routine. If that node is not a fraction bar, integral sign or summation, the

target node will be updated to refer to that node, and the process continues. When the routine exits, the node that *target* refers to will be returned. It represents the top level node that is in column relation with the *newnode*.

5.8.2 Insert the Node into Expression Tree

If the returned value of the routine *ColParent* is a valid node, it is the place where the *newnode* should be inserted into the expression tree. However this is not a trivial thing to do. Two conditions should be considered in order to integrate the *newnode* into the expression tree.

Firstly, the returned node is not necessarily the parent node of the *newnode*, or vice versa. For example, if the fraction bar is written first in the imcompleted fraction $\frac{a}{}$, it will be the node returned by *ColParent*. It is also the parent node of "*a*" (the *newnode*). On the contrary, if "*a*" is written first, it will be returned by *ColParent*, and it will be the child node of the fraction bar (the *newnode*). If neither is the parent of the other then a new node is created as the parent of both nodes.

Secondly, in our system the structural analysis is applied on the fly, whenever a new symbol is written, it is recognized and attached to the expression tree. This causes a problem when a new symbol is introduced, the previous structural analysis may be proved wrong and has to be fixed. For example, when a fraction bar is written beneath the *b* in a^b, the meaning of the expression will change completely. Formerly *b* is the superscript of *a*, now (in $a\frac{b}{}$) *b* is the numerator of a fraction and the expression is the implicit multiplication of *a* and the fraction. This kind of problem must be dealt with correctly.

In our algorithm, we first check whether the previous analysis result is wrong, if it is, then the expression has to be rearranged to fix the error. Otherwise it is not necessary to modify the previous analysis results. In both conditions, we use the value (symbol) and size information of the nodes to decide how to add the *newnode* to the tree. We detail our methods below.

Is Rearrangement Needed?

When the *ColParent* routine returns a node that is in a column with the *newnode*, we must find out whether the previous analysis result is wrong or not, but how? We observed that this only happens if the previous analysis result is a superscript or subscript relation. For this to happen, if the *newnode* is in a column with the superscript or subscript node, it must be in a row with the parent node or an ancestor node of the superscript or subscript node. In above example the fraction bar is in a row with the symbol *a*, which has the same baseline as the sub-expression a^b due to our conditional bounding box updating method.

Therefore, we start with the parent of the node returned by *ColParent* routine, we shall call it *parent*. If it is a superscript (*msup*) or subscript (*msub*) node, then we check whether it is in a row with the *newnode*. If the result is positive then we stop the check (here we say a match is found). Otherwise we do the same check on *parent*'s parent node, grandparent node, and so on if necessary. Whenever a match is found during the process the check will terminate. Let's call the matched node *match* for convenience. If no match has been found at all, we assume that the *newnode* does not affect the result of previous structural analysis.

Rearrangement Cases

Expression tree rearrangement is required if a match is found in the above check. The first step should be to detach the *parent* node from the expression tree and group it with the *newnode*. There are three ways to group them together based on their symbols and size.

1. If both *parent* and *newnode* are fraction bars, the wider one is considered to be the parent of the other.
2. If either but not both of *parent* and *newnode* is a fraction bar, then the fraction bar is considered to be the parent node.
3. In case neither of *parent* and *newnode* is a fraction bar, then we take the one with a larger height to be the base. If the other one is above it, we create a new node *mover* as the new parent of both nodes, otherwise we create a new node *munder* as the

parent of both nodes. To be consistent with presentation MathML, in both conditions we make sure that the one with the larger height is the first child.

After detaching the node *parent*, what is left of the expression tree may or may not need modification before the sub-tree formed by *parent* and *newnode* is added, depending on the situation.

1. After detaching the *parent*, its original parent node has to be deleted. Figure 5.7 shows such an example. When the expression changes from a^2 to $a\frac{2}{}$, the relation *msup* is no longer valid. So after detaching the *parent* and grouping it with *newnode* (Figure 5.7(b)), the node that *match* refers to (*msup*) is deleted, its child node *a* has been moved to occupy the vacancy it leaves. The *match* also changes its reference to this node (Figure 5.7(c)). After the rearrangement the expression tree correctly reflects the structure of the expression (Figure 5.7(d)).

2. After detaching *parent*, its original parent node will not be deleted because that relation still exists between other nodes. Figure 5.8 illustrates a sample expression of this kind. When the expression changes from a^{234} to $a^{23\frac{4}{}}$, the node *4* does not belong to the superscript relation any more. In the rearrangement, *4* is removed from the tree and grouped together with the fraction bar, while the node *234* has been modified to reflect this change.

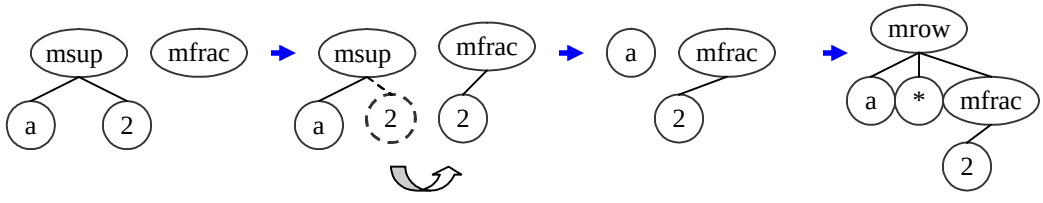
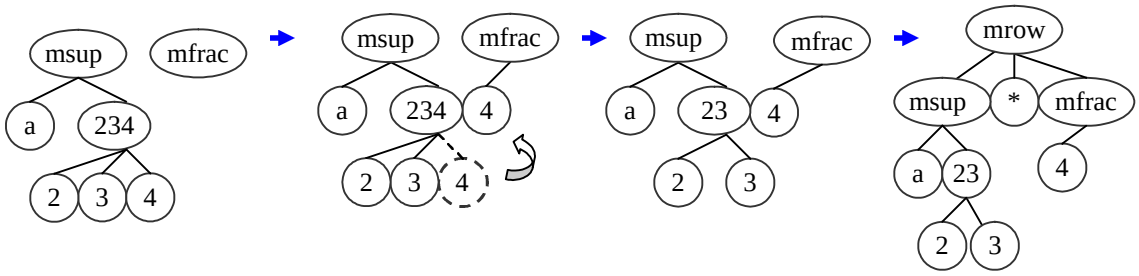


Figure 5.7 The steps of expression tree rearrangement from "a^2" to "$a\frac{2}{}$".
Node *mfrac* is the *newnode*, node *2* is the *parent*, node *msup* is the *match*, and node * means implicit multiplication. The diagrams reflect the following stages:
(a) before rearrangement;
(b) detach *parent* and group it with *newnode*;
(c) delete *parent*'s original parent node *msup* and reset *match* to refer to *parent*;
(d) after rearrangement.



(a) (b) (c) (d)

Figure 5.8 The steps of expression tree rearrangement from "a^{234}" to "$a^{23}\underline{4}$".
Node *mfrac* is the *newnode*, node *4* is the *parent*, node *msup* is the *match*, and node * means implicit multiplication. The diagrams reflect the following stages:
(a) before rearrangement;
(b) detach *parent* and group it with *newnode*;
(c) modify *parent*'s original parent node to reflect the change;
(d) after rearrangement.

In both conditions stated above, the last step of the rearrangements (Figure5.7(d) and Figure5.8(d)) have been done in the same way. It is guaranteed that *match* and the sub-tree containing *parent* and *newnode* must be in a row relation. So we check whether the parent node of *match* represents a row relation, if it does we can insert the other node to the right of *match*. Otherwise create a new "*mrow*" node as the parent node of both nodes, with *match* as the first child. This is placeed at the original position of *match* in the expression tree. Finally, new node will be created for any implicit operator and inserted between the two nodes as a sibling node.

Non-Rearrangement Cases

If there is no need to rearrange the expression tree the task is much simpler. Firstly detach the *parent* from the tree and group it with *newnode* in exactly the same way as in the rearrangement cases. Then put the newly formed sub-tree back where the *parent* node was before it got detached. This completes the task.

5.9 Superscript and Subscript Direction Check

We can be sure that there is no long range grouping for the given node if both the row direction check and column direction check fail to find an appropriate parent node from the expression tree, for a given node. In this case, the most likely grouping will be

directly between the node and its NN node. It is only necessary to check for short range grouping in the case of superscript and subscript relationships. Compared with the row and column direction checks, superscript and subscript directions are much simpler to handle.

5.9.1 Superscript Direction Check

If a node is the superscript of its NN node, we need to create a new node "*msup*" to reflect this implicit operator. The "*msup*" node has two children nodes, the superscript node will be the second child for sure, however, what will be the first child depends on the context. Generally there are four possibilities.

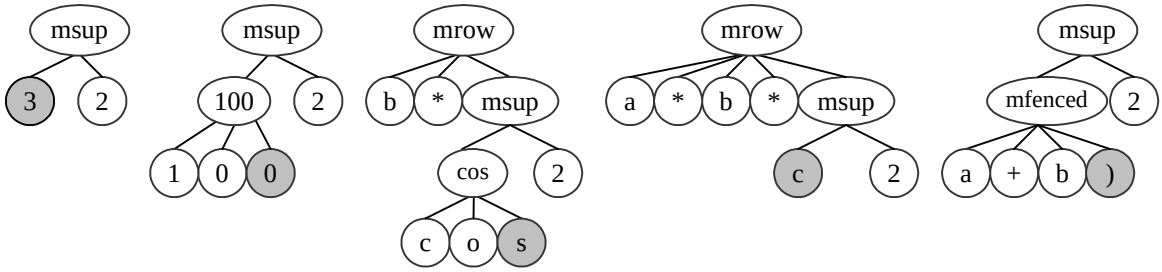
1. If the NN node is a digit, we need to check its parent node first. Either the parent node does not exist, or the parent node does not represent an integer or float number, one may be sure that the NN node represents a one-digit number, therefore the NN node is set as the first child of the "*msup*" node. On the contrary, if the parent node represents a number, we know that the NN node is only part of that number, therefore the parent node is selected as the first child of the "*msup*" node. Expression 3^2 and 100^2 belong to these two possible cases respectively (Figure 5.9 (a), (b)).

2. If the NN node is an alphabet string, there are also two options, depending on the parent node of the NN node.

   When the parent node does not exist, or it is label is not an alphabet string, it means that the NN node itself is a one-letter string. Therefore the NN node is chosen as the first child node of the "*msup*" node.

   However, if the parent node does represent an alphabet string, tree rearrangement is needed. The parent node will be split into several nodes as described in section 5.6 (special case 4). Whichever node contains the NN node (itself or another node that has it as a child node) will be chosen as the first child node of the newly created "*msup*" node. For example if the expression $bcos^2$ is input to the node split algorithm the results is two nodes, *b* and *cos*. The node *cos* will be the first child of the "*msup*" node since it contains the NN node *s* (Figure 5.9 (c)). While in abc^2, the

node split algorithm returns three nodes *a*, *b* and *c*. Node *c* will be the first child of "*msup*" (Figure 5.9 (d)).



(a) 3^2 (b) 100^2 (c) $bcos^2$ (d) abc^2 (e) $(a+b)^2$

Figure 5.9 Expression tree after superscript direction check.
In all these cases "*2*" is the node to be checked, the shadowed node is its NN node. The '*' node represent implicit multiplication. The diagrams reflect the following cases:
(a) NN node is a one-digit number.
(b) NN node is part of a multi-digit number.
(c) NN node is part of a function name.
(d) NN node is a letter that is not part of a function name.
(e) NN node is a close parenthesis.

3. Special care must be taken when the NN node is a close parenthesis. Usually a pair of matching parentheses along with their enclosed sub-expressions are considered as one unit. Therefore, in this situation we will fetch the parent node of the NN node and make it the first child of the "*msup*" node (in our system, the parent of a close parenthesis node is guaranteed to be the node that contains the matching open parenthesis). In expression $(a+b)^2$ the first child node of "*msup*" is the sub-expression $(a+b)$ as a whole (Figure 5.9 (e)).

4. In all other cases, the NN node itself will be chosen (same as Figure 5.9(a)). An example expression is a^2.

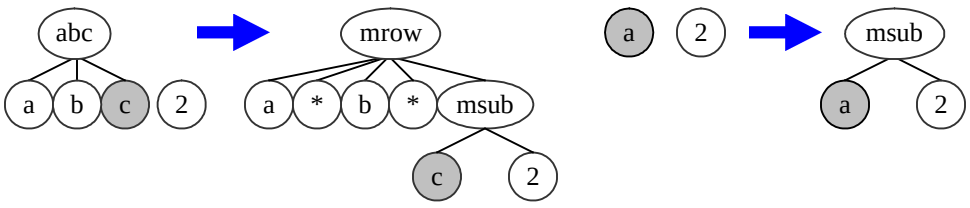
5.9.2 Subscript Direction Check

Subscript direction check is quite similar to superscript direction check except that it is impossible for parenthesis to have a subscript, and it is very rare for a digit to have a subscript, *eg.* 5_2. We note that this is possible, for example $a2_2$ is a valid variable name, or 5_2 to mean 5 modulus 2. We will not consider this case for now.

The subscript relation is represented by an implicit operator node "*msub*". When a node is in the subscript position of its NN node, such an "*msub*" node has to be created first. Like "*msup*", it also has two children, among which the second child is always the subscript node, and the NN node is always the node being subscripted. However, in some cases rearrangement is needed for the expression tree. Roughly speaking there are two cases in subscript check:

1. If the NN node represents an alphabet string, we should check its parent node. If its parent also represents an alphabet string, then node split is necessary to isolate the NN node from its parent node, and make it the first child of the "*msub*" node. Note that it is possible for a function to have a subscript, like in $log_2 a$, however we leave this situation for now.
2. In all other cases, no tree rearrangement is necessary.

Figure5.10 shows examples for the above two cases.



(a) abc_2 (b) a_2

Figure 5.10 Expression tree after subscript direction check.
In all these cases "*2*" is the node to be checked, and the shadowed node is its NN node. The diagrams reflect the following stages:
(a) Node split is involved in the process.
(b) No node split is involved in subscript check.

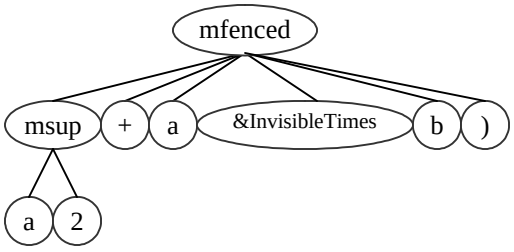
Chapter 6 MathML Generation

In our system, the hierarchical structure of the expression tree is designed in such a way that it will be trivial to translate the tree into presentation MathML. Firstly, MathML keywords are used as the label of nodes as much as possible. For example, *mrow* is used to represent row relation, *mfenced* is used to represent fenced row relation, *mfrac* is used to represent fraction, and so on. Secondly, all implicit operators of the expression are explicitly represented as nodes in the expression tree. Thirdly, a flag has been set for all nodes that are not important for MathML code generation in order to mark them out.

6.1 Generate MathML with Preorder Tree Traversal

With these features, MathML code can be generated very easily. A preorder traversal of the expression tree with minor changes will suffice. In our system, during the preorder traversal, nodes that have been flagged as above (in section 5.6 and 5.7.1) are ignored, the labels of other nodes are used either as tag names or values. For example, *mrow*, *mfenced*, *mfrac*, *msup*, *etc.* are used as tag names. Some other nodes' labels are used as tag values, different tag names are added to them depending on their types. For example tag name *mn* is used for numbers, *mo* is used for operators and implicit operators, *mi* is used for variables. Figure 6.1 shows the relationship between the expression tree and the presentation MathML code for the expression $(a^2 + ab)$. The MathML code generated by our traversal method will be exactly the same as that in the figure.

It is therefore critical that the expression tree represents an expression correctly in order that we generate error-free MathML code. Although in the structural analysis step we have already considered and added as many implicit operators as we can, we may still miss some implicit operators just like an ignorant human reader may miss some esoteric implicit operators. There are even some implicit operators that can not be identified until the last minute. No doubt a final check is necessary before generating the MathML code.




```
<mfenced>
   <msup>
       <mi>a</mi>
       <mn>2</mn>
   </msup>
   <mo>+</mo>
   <mi>a</mi>
   <mo>&InvisibleTimes;</mo>
   <mi>b</mi>
</mfenced>
```

Figure 6.1 The corresponding expression tree and presentation MathML code of expression (a^2+ab). Note that the node ‘)’ has been flagged as false.

6.2 Final Check on Expression Tree

The final check contains a couple of operations, including merging or splitting some neighboring nodes, as well as implicit operator modifications. Some of the steps in the final check, especially the implicit operator modifications, are highly experimental and perhaps should not be part of the structural analysis at all, but rather should more properly be part of a semantic analysis phase. Since we do not have a separating semantic analysis phase we made these part of the structural analysis for now. This issue is discussed in Chapter 7.

6.2.1 Split Nodes When Necessary

In our algorithm, whenever a symbol is written next to an alphabet string, we check whether that string contains a function name, if a function name is found then the alphabet string node is split, otherwise the node will be left alone.

In the final check step we must split all alphabet nodes that are not function names, as well as the nodes that contain function names but are actually not. For example, in expression $a+tg$ the *tg* has no arguments therefore should be considered as the implicit multiplication between *t* and *g*, instead of the trigonometirc function *tg*. It is performed in this way: for each node that contains a function name, check whether its next sibling node contains a label “*⁡*”, a positive answer means that the node we are checking is really a function name, and we should ignore that node. On the contrary, a

negative answer means that the node is not really a function name, therefore that node will be split and treated as the implicit multiplication of all children nodes of that node (Figure 6.2).

6.2.2 Merge Nodes When Necessary

In the expression tree of an expression, letters and digits are always represented by different nodes. This results in the separation of letters and digits in the same variable, like in the expression $a11+a21=a31$. In the final check this problem must be resolved. What our system does is to check whether an alphabet string node has an adjacent integer node, if it does then we merge the two nodes together by deleting the alphabet string node and making it the leftmost child of the integer node. Sometimes it is also necessary to remove the parent node if they are the only children nodes of that node. Figure 6.3 illustrates examples for both cases.

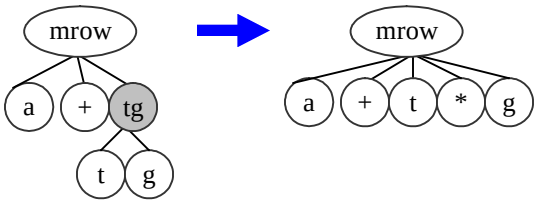


Figure 6.2 Split alphabet string node that is not really a function name in the final check. Node '*' means "⁢"

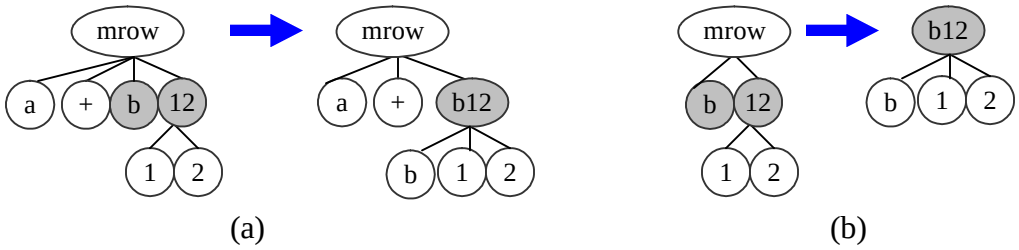
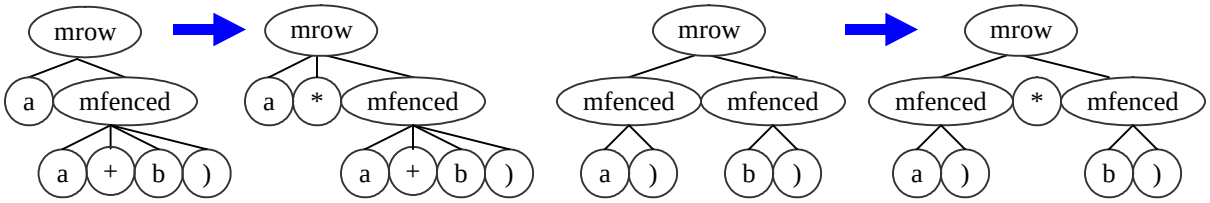


Figure 6.3 Merge alphabet string node with its adjacent integer node to form a node representing a variable.

6.2.3 Add Missing Implicit Operators

Besides the above conditions, the final check step also identifies other possible implicit operators. For example, when an *mfenced* node is adjacent to a node representing a number, a letter, another *mfenced* node, a fraction, *etc*. There should be an implicit multiplication between them. Figure 6.4 shows two of these cases.



(a) *a(a+b)* (b) *(a)(b)*

Figure 6.4 Add necessary implicit operators to the expression tree.
In both diagrams the node "*" always represents "⁢"

After the final check, we can make sure that the preorder traversal of the expression tree can generate correct presentation MathML code easily.

6.2.4 Identify Ambiguous Implicit Operators

Some implicit operators can not be identified until the last step. For example, in the expression $a\frac{b}{c}$ the implicit operator between a and $\frac{b}{c}$ is implicit multiplication, but in the expression $2\frac{3}{4}$ the implicit operator between *2* and $\frac{3}{4}$ is an implicit plus. It is impossible to identify the implicit operator until the fraction is finished.

In our initial design, in the final check, if a number is found to be followed by a fraction whose numerator and denominator are both numbers, we would like to insert a "*&InvisiblePlus;*" node between them. Otherwise we insert a "*⁢*" node. It is possible that an expression can be treated in a different manner before and after it is finished, but this presents no problem since a final check operation must be performed every time before the MathML code is regenerated. Unfortunately the current MathML version (MathML version 2.0) does not support "*&InvisiblePlus;*", and therefore in this

thesis no implicit operators are added for both $2\frac{3}{4}$ and $a\frac{b}{c}$ cases (to keep consistency, although we could have inserted "*⁢*" for the $a\frac{b}{c}$ case).

We conclude that the MathML standard is lacking an invisible plus operator, it is our hope that this shall become part of a future standard. Currently alternative markup for presentation of $2\frac{3}{4}$ (*ie.* $2+\frac{3}{4}$, with an invisible plus) is to associate the presentation with the corresponding semantical meaning as follows (not supported by our program for now):

```
<semantics>
   <mn>2</mn>
   <mfrac>
      <mn>3</mn>
      <mn>4</mn>
   </mfrac>
   <annotation-xml encoding=MathML-Content>
      <apply>
         <plus/>
         <cn>2</cn>
         <apply>
            <divide/>
            <cn>3</cn>
            <cn>4</cn>
         </apply>
      </apply>
   </annotation-xml>
</semantics>
```

Chapter 7 Existing Problems and Future Work

At this stage, our system still has many open issues. In the handwriting symbol recognition part, the speed is quite slow, and the recognition rate becomes worse when the symbol set size becomes large. In the structural analysis part, simple polynomial expressions, fractions, integrations and summations can be recognized quite well. However, complicated expression may not be recognized correctly, and such expressions as square roots, matrices, *etc.* can not be recognized at this time. In the following we shall address these problems and discuss possible solutions for them. Also we introduce strategies to further improve the system.

7.1 Handwriting Recognition Problems

Our *ElasticRecognizer* is model based on the approach that whenever a new scribble is delivered to the *ElasticRecognizer*, it is compared with all available models and the model that gives the minimum distance is picked as the result. *Elastic matching* is a computationally intensive algorithm, where the speed depends on the number of models. Our testing program is currently provided with 52 models, the speed seems acceptable in the Windows CE simulator on a Pentium III 800MHz desktop computer with 256MB memory, but when it is run on the iPAQ3600 pocket PC (StrongARM 206MHz processor, 32MB memory), the speed is too slow for the system to be usable in real life.

It is not possible to exclude the possibility of irrelevant symbol (model) appearing to be the recognition result, when the model size becomes large. This implies that the chances of recognition error may also become large as the model size becomes large.

Both of the above problems are connected with the size of the models used in the system. However, a large model set enables the recognizer to recognize more symbols. One possible way to surround this dilemma would be to introduce structural information of the symbols, to the recognition scheme.

Different mathematical symbols have different structures. Some of them are quite alike, especially in handwriting, for example *a*, *d* and *q*. While some of them are totally different, for example, *a* and =. If we can make use of the structural (*e.g.* shapes) information of the symbols and classify the symbols with different structural information into different categories. When recognizing a symbol, the recognizer can first extract the structural information from the symbol, then search the collection of categories to find the category or categories that have matching structural information. The elastic matching recognition will be based only on models in the matching categories. This will drastically reduce the amount of computation and therefore speed up the recognition significantly. Furthermore, since irrelevant symbols belong to different categories, they are not considered in the recognition, this reduces the chance of recognition errors.

Another way to improve symbol recognition is to provide feedback from structural analysis. The context information recognized in the structural analysis could be very useful for handling symbol recognition errors or ambiguities.

It is also possible to improve the *elastic matching* algorithm itself. In this algorithm each point of the model is considered to be of equal importance, and each point being matched in the unknown contributes an equivalent amount to the total distance. Scattolin [32] proposed an improved *Weighted Elastic Matching* algorithm that adds different weights to each point of the model, and therefore allows the most significant distinctive portions of the characters take on more importance. As a result the weighted elastic matching algorithm results in better recognition rate.

7.2 Structural Analysis Problems

Our structural analysis algorithm is in an early stage of development and there are still a lot of features that our structural analysis algorithm can not recognize. These include square roots, matrices, presuperscript and presubscript. Also the system assumes that symbols are written from left to right in the horizontal direction, except for parenthesis. All these problems should be taken into account in order that the system would be fully functional.

Our structural analysis algorithm is based on the relationship between bounding boxes of symbols in the expression. However, some relationships in mathematical expressions are so complicated that it is not easy to identify them through bounding boxes. In the following we discuss problems relating square roots and matrices.

7.2.1 Square Root

When an expression contains square roots, the square root's bounding box actually overlaps the bounding boxes of all the symbols inside the square root. Recognition is even more complex, because the centroid of the square root's bounding box is in the baseline of the sub-expression inside it. When new symbols are added to the sub-expression, it is quite possible that the algorithm will treat the square root as its NN node. Figure 7.1 illustrates this case, in which the centroid of the square root turns out to be the NN node of the +. This causes failure of the structural analysis algorithm.

Another problem which is also caused by a special feature of the square root's bounding box is the following. Since the centroid of the square root's bounding box is in the middle of the sub-expression inside it, the relationship between the square root sub-expression and other sub-expressions will be hard to correctly identify. This can be clarified with the expression $\sqrt{a+b}-$. In this expression '-' is in row direction with $\sqrt{a+b}$, however, the node closest to it is not the square root sub-expression, but the node '*b*'. This will result in the '-' being interpreted as part of the sub-expression *a*+*b*.

To solve this problem, it is necessary to set up some special rules such that when square root is involved in the expression, specially designed techniques instead of general techniques are used to check for relationship between nodes.

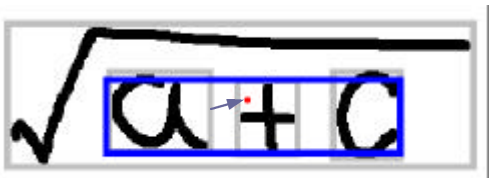


Figure 7.1 The bounding box hierarchy of the expression $\sqrt{a+c}$.
The dot pointed to by the arrow is the centroid of the square root.

7.2.2 Matrices

Compared with square root, matrix is even more difficult to interpret. Firstly, in an $n \times m$ matrix, the *nm* elements are evenly distributed in *n* rows, with *m* elements per row. In most cases, when a symbol is added to the matrix, it is usually very hard to determine its closest neighbor because it is about the same distance from both the element above it or the element to the left of it (suppose the matrix is filled from left to right and top to bottom). For example, in the expression showed in Figure 7.2, the element *7* is the same distance from *3* and *6*.

The bounding box for a matrix should be the union of the bounding boxes of all the elements and both square brackets. However, the distance between its centroid and other sub-expressions will be much greater than the distance between its square brackets or some of its elements and those sub-expressions. As in Figure 7.2, the distance between the $\times$ and the right bracket of the left matrix is much smaller than the distance between the $\times$ and the left matrix itself. This problem has to be solved in order to correctly interpret the relationship between matrices and other sub-expressions.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \times \begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}$$

Figure 7.2 An expression that contains matrices.

One suggestion for solving the structural analysis problem for matrices is this: Format special rules to handle symbols within a matrix and outside a matrix separately. Inside a matrix, consider horizontal relationship between symbols only, except the first element of each row, this excludes the interference from symbols in the column direction. When the matrix is finished, it will be treated as one unit. When determining the relationship between the matrix and other sub-expressions, the bounding boxes of all elements inside the matrix will not be considered, only the bounding box for the entire

matrix is considered. The matrix is thus treated as a single symbol, therefore guaranteeing that the relation between the matrix and other sub-expressions can be correctly interpreted by our algorithm.

7.3 Future Improvement

Currently, our implementation for structural analysis of mathematical expressions uses a general tree. It contains its own methods for tree editing and traversal, which is sufficient for the purpose of structural analysis and MathML code generation. However there are disadvantages.

The main disadvantage is that it is not easy to modify expressions. Currently, to modify an expression, the user has to apply the *Undo* operation to go back to the symbol that should be changed, erase that symbol and write a new one. However, all the symbols that are involved in the undo operations are also erased, and the user has to write them again. This means that the system must perform symbol recognition and structural analysis on all these symbols again. This is very time consuming and is not a user-friendly strategy.

Our mathematical expression recognition system is a compromise between the approach taken in section 4.5.1.1 and that taken in section 4.5.1.2. Right now the structural analysis handles semantic analysis as well as spatial relationship analysis. It would be better to make it modularized by separating the semantic part and make it an individual semantic phase.

It is highly desirable to make expression modification more efficient and user-friendly, *i.e.* allow editing of individual symbols in the expression without affecting other symbols. This can be done in an event driven manner like this: whenever the user intend to change a symbol in the expression, he/she can click that symbol on the digitizer and the program will ask the user whether he/she wants to change that symbol. Once the request is confirmed the clicked symbol will disppear and the user can write down a desired symbol in that place. When the new symbol is done an event will be fired. It contains enough information for the system to know how to locate the symbol in the

expression tree and how to change it, therefore the changes in the expression can be reflected in the correct tree node without affecting other nodes of the expression tree.

Another disadvantage is that currently our system can not support dynamic symbol rewriting, that is, when a symbol is recognized, it will be replaced by a typeset one at the same location, with the correct font and size. The way to do this would be to link the expression tree nodes with font objects, and each font object knows how to redisplay the node it is linked to.

Our design for the expression tree is purely for converting handwriting mathematical expressions to presentation MathML. Although it contains structural relationships of the expression as well as semantic information like implicit operators, it would be hard to use the expression tree for other purposes. For example, render it to TEX, or display it as a tree view, *etc.* with different rendering methods. We realize this problem and intend to move the design to this direction in the future. By doing this our system will be more extensible and more flexible.

Mr. Luca Padovani is working on MathML rendering in our lab, the Ontario Research Center for Computer Algebra (ORCCA) at UWO. His work keeps synchronization between a presentation MathML tree and a rendering tree. Any changes in the MathML tree can be reflected in the rendering tree automatically. Since the rendering tree contains the semantics of the expression, it is able to render an expression in many ways. For example, it can save the information as presentation MathML code that can be accepted directly by computer algebra systems, and it can also render the information into a tree view structure. This design can also be extended very easily to other kinds of rendering ways that perform different tasks.

Mr. Padovani's MathML tree is based on the DOM (Document Object Model), a standard for representing an XML document in tree structure. The DOM contains a complete set of methods for modification of the tree. Also each DOM tree node can be linked to other objects, or to an event. So both the expression modification problem and the symbol redisplay problem discussed above can be solved without changing the tree design.

If we modify the design of the general expression tree in some way and link it to Mr. Padovani's work, then our mathematical expression recognizer will be much more useful. Because the rendering tree can use many different rendering ways to satisfy different recognition needs. The question is: what modification do we need to do for this purpose? There are two options:

1. We can change the expression tree to be a DOM MathML tree. In this way our expression tree can be directly synchronized with Mr. Padovani's rendering tree. A possible problem with this is that maintaining a MathML DOM tree is not easy, especially when *undo* operation is performed. In this case it would be very hard to rearrange the MathML tree. Futhermore, keeping the synchronization between the MathML tree and the rendering tree is also hard, and it is even harder when *undo* operation is performed.
2. An alternative way is to keep the current expression tree. However, we can generate a MathML DOM tree from the expression tree. When rendering is needed we can generate a MathML tree, which can be linked to the rendering tree. In this way we do not have to worry about the synchronization between the MathML tree and the rendering tree. The disadvantage of this approach is that we have to keep three trees in the program therefore consume more of the limited resources.

No matter which of the two ways (or some other possible ways) we may choose to modify the expression tree design, from the above we comprehend an overall view of an ideal handwriting mathematical expression recognition system. We summarize this as follows:

1. Individual handwritten mathematical symbols are recognized dynamically, with contextual hints.
2. For each recognized symbol, structural analysis is carried out to interpret its relation with other symbols, and the symbol is added to the expression tree correctly. This expression tree can be either a DOM MathML tree or is able to generate a DOM MathML tree.

3. A rendering tree is used to render the DOM MathML tree. It is synchronized with the DOM MathML tree all the time and is able to render it in many different ways.
4. Each of the renderers would perform a different task. One renderer may generate presentation MathML code, which is accepted as input by computer algebra systems. Another renderer is responsible for generating a tree view of the expression on the screen. Yet another renderer is able to generate corresponding $\TeX$ code for the expression, and so on.

References

1. R. Anderson. Two-dimensional Mathematical Notation. In K.S.Fu, editors, Syntactic Pattern Recognition, Applications, pages 147-177. Springer Verlag, New York, 1977.
2. L.R. Bahl, F. Jelinek and R.L. Mercer. A Maximum Likelihood Approach to Continuous Speech Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 5:179-190, March 1983.
3. A. Beláid and J.P. Haton. A Syntactic Approach for Handwritten Mathematical Formula Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 6(1):105-111, 1984.
4. D. Blostein and A. Grabvec. Recognition of Mathematical Notation. In H. Bunke and P.S.P. Wang, editors. Handbook of Character Recognition and Document Image Analysis, pages 557-582, World Scientific, Singapore, 1996.
5. K.F. Chan and D.Y. Yeung. An Efficient Syntactic Approach to Structural Analysis of On-line Hand-written Mathematical Expressions. Pattern Recognition. 33(3): 375-384, 2000.
6. K.F. Chan and D.Y. Yeung. Elastic Structural Matching for On-line Handwritten Alphanumeric Character Recognition. Technical Report CS98-07, Department of Computer Science, Hong Kong University of Science and Technology, March 1998.
7. K.F. Chan and D.Y. Yeung. Error Detection, Error Correction and Performance Evaluation in On-Line Mathematical Expression Recognition. Pattern Recognition, 34(8): 1671-1684, 2001.
8. K.F. Chan and D.Y. Yeung. Mathematical Expression Recognition: A Survey. International Journal on Document Analysis and Recognition, 3(1): 3-15, August 2000.
9. S.K. Chang. A Method for the Structural Analysis of Two-dimensional Mathematical Expressions. Information Sciences, 2:253-272, 1970.

10. M. Chen and A. Kundu. A Complement to Variable Duration Hidden Markov Model in Handwriting Recognition. In ICIP'94 (International Conference on Image Processing), pages 174-178, 1994.
11. L.H. Chen and P.Y. Yin. A System for On-Line Recognition of Handwritten Mathematical Expressions. Computer Processing of Chinese and Oriental Languages, 6(1): 19-39, 1992.
12. P.A. Chou. Recognition of Equations Using a Two-dimensional Stochastic Context-free Grammar. In Proceedings of the SPIE Visual Communications and Image Processing IV, 1199: 852-863, Philadelphia, 1989.
13. M.R. Davis and T.O. Ellis. The RAND Tablet: A Man-machine Graphical Communication Device. In Proceedings of Fall Joint Computing Conference, pages 325-331, 1964.
14. Y.A. Dimitriadis and J.L. Coronado. Towards an ART Based Mathematical Editor, that Uses On-line Handwritten Symbol Recognition. Pattern Recognition, 28(6):807-822, 1995.
15. T.L. Dimond. Devices for Reading Handwritten Characters. In Proceedings of Fall Joint Computing Conference, pages 232-237, 1957.
16. C. Faure and Z. Wang. Automatic Perception of the Structure of Handwritten Mathematical Expressions. In R. Plamondon and C. Leedham, editors, Computer Processing of Handwriting, pages 337-361, World Scientific, Singapore, 1990.
17. H. Freeman. Computer Processing of Line Drawing Images. ACM Computing Surveys, 6(1): 57-98, 1974.
18. R. Fukuda, S.I.F Tamari, *et al.* A Technique of Mathematical Expression Structure Analysis for the Handwriting Input System. ICDAR'99 (International Conference on Document Analysis and Recognition), 28:131-134, 1999.
19. A. Grbavec and D. Blostein. Mathematics Recognition Using Graph Rewriting. In Third International Conference on Document Analysis and Recognition. Montreal, pages 417-421, 1995.
20. J. Ha, R.M. Haralick and I.T. Phillips. Understanding Mathematical Expressions From Document Images. ICDAR'95, 26:956-959, 1995.

21. S. Hellkvist. On-line Character Recognition on Small Hand-Held Terminals Using Elastic Matching. Master's Thesis, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, 1999.
22. R.H. Kassel. A Comparison of Approaches to On-line Handwritten Character Recognition. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.
23. S. Lavirotte and L. Pottier. Mathematical Formula Recognition Using Graph Grammar. In Proceedings of the SPIE, 3305:44-52, San Jose, 1998.
24. H. Lee and M. Lee. Understanding Mathematical Expressions in a Printed Document. In ICDAR'93, Tsukuba, Japan. pages 502-505, 1993.
25. H. Lee and M. Lee. Understanding Mathematical Expressions Using Procedural-Oriented Transformation. Pattern Recognition. 27(3):447-457, 1994.
26. R. Marzinkewitsch. Operating Computer Algebra Systems by Handprinted Input. In Proceedings of ISSAC, pages 411-413, Bonn, Germany, July 1991.
27. A. Meyer. Pen Computing - A Technology Overview and a Vision. Department of Computer Science, University of Zürich, Switzerland, 1995. (http://www.amug.org/amug/sigs/newton/nanug/PenReport/NewPenCom.html)
28. E.G. Miller and P.A. Viola. Ambiguity and Constraint in Mathematical Expression Recognition. In Proceedings of the Fifteenth National Conference on Artificial Intelligence. pages 784-791, Madison, Wisconsin, 1998.
29. Y. Nakayama. Mathematical Formula Editor for CAI. In Proceedings of the ACM SIGCHI Conference on Human Factors in Computer Systems, 387-392, Austin, Texas, 1989.
30. M. Okamoto and B. Miao. Recognition of Mathematical Expressions by Using the Layout Structure of Symbols. In Proc. First International Conference on Document Analysis and Recognition, Saint Malo, France, pages 242-250, September 1991.
31. M. Okamoto and A. Miyazawa. An Experimental Implementation of Document Recognition System for Papers Containing Mathematical Expressions. In H.

Baird, H. Bunke and K. Yamamoto, editors, Structured Document Image Analysis, pages 36-53. Springer-Verlag, 1992.

32. P. Scattolin. Recognition of Handwritten Numerals Using Elastic Matching. Master thesis, Department of Computer Science, Concordia University, Montreal, Quebec, 1995.
33. S. Smithies, K. Novins and J. Arvo. A Handwriting-Based Equation Editor. In Proceedings of Graphics Interface ’99, Kingston, Ontario, pages 84-89, 1999.
34. C.C. Tappert. Recognition System for Run-On Handwritten Characters. United States Patent, 4731857, March 1988.
35. C.C. Tappert, C.Y. Suen and T. Wakahara. The State of the Art in On-Line Handwriting Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(8): 787-807, 1990.
36. C.C. Tappert, C.Y. Suen, *et al.* On-line Handwriting Recognition – A Survey. The 9th International Conference on Pattern Recognition, 2:1123-1131, 1988.
37. Z. Wang and C. Faure. Structural Analysis of Hand-Written Mathematical Expressions. In Proceedings of the 9th International Conference on Pattern Recognition, pages 32-34, Rome, Italy, 1988.

Appendix A: The *ModelBuilder* Application

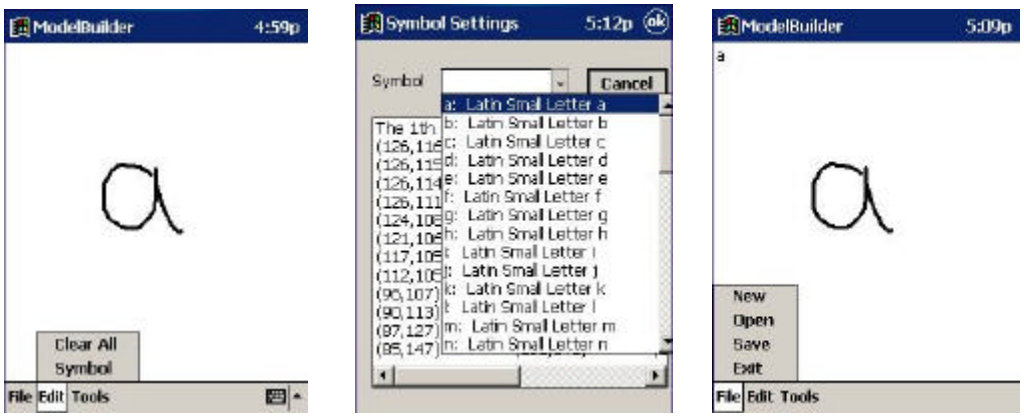
The handwriting recognizer, *ElasticRecognizer*, is model based. It is necessary to find a way to create models as well as modifying existing models. For this reason we designed and implemented a simple application called *ModelBuilder*.

The *ModelBuilder* is also implemented with Microsoft Embedded Visual C++ using MFC. The user interface is very similar to that of the mathematical expression recognizer. Basically it is a drawing board for user to write on (Figure A.1(a)). A user can write a symbol on the writing area, then select the menu *Edit®Symbol* to enter the *Symbol Settings* window (Figure A.1(b)). This window displays the points of all strokes of the symbol, and provides a list of available symbols. Once the corresponding symbol is chosen, it is displayed at the upper left corner of the writing area (Figure A.1(c)), and a link has been established between the handwritten symbol and its identity. The model can then be saved to disk as a text file. In the *ElasticRecognizer,* the default location for model files is the "`\My Documents\MdlBase`" of the pocket PC, therefore it is desired to save model files into that directory in order for the *ElasticRecognizer* to load it.

It is true that file I/O is slow, especially when there are a large number of models. However this design is better than hard coding every model. Our application is user-dependent, a user can create and maintain his/her own models to make sure that his/her handwriting be recognized well. With hard coding, there is no way to do that without changing the source code, which is impossible. By storing model files on disk, a user can simply add or delete files in the directory and the change can be automatically reflected in the recognition. Futhermore, it is only necessary to load the models once at the initiation stage of the application, then the models are ready to be used until the application exits. So a little delay at the beginning will not be a big problem.

It is worth mentioning that the way we create models here is only a basic one. A bad model may hinder the recognition. It is necessary to carry out some studies on this issue, but we did not due to time restrictions. The Master thesis of Scattolin [32] has a

chapter discussing model selection problems and solutions. This would be a good reference for future improvement of the *ModelBuilder* application.



(a) (b) (c)

Figure A.1 Screen shots of building a model for the Roman letter “a” with the *ModelBuilder*.

(a) Write the model in the input area.
(b) Select the corresponding identity for the model.
(c) The model is ready to be saved to physical storage

Vita

Name:	Bo Wan
Place of Birth:	Nanchong, China
Post-Secondary Education and Degrees:	The University of Western Ontario London, Ontario 2000 - 2001 M.Sc (Computer Science) Sichuan University Chengdu, China 1991 - 1994 M.Sc. (Genetics) Sichuan University Chengdu, China 1987 - 1991 B.Sc. (Microbiology)
Honors and Awards:	Special University Scholarship The University of Western Ontario 2000 - 2001 Dean's Honor List Standing The University of Western Ontario 1999 - 2000
Related work Experience:	Teaching Assistant The University of Western Ontario 2000 - 2001 Research Assistant The University of Western Ontario 2000 - 2001 Software developer Student Development Center The University of Western Ontario 2000