CENTER FOR
MACHINE PERCEPTION

CZECH TECHNICAL
UNIVERSITY IN PRAGUE

MASTER THESIS

2D Grammar Extension of the CMP Mathematical Formulae On-line Recognition System

Eva Gallardo Pérez

qqperez@cmp.felk.cvut.cz

CTU–CMP–2009–03

January 21, 2009

Available at
ftp://cmp.felk.cvut.cz/pub/cmp/articles/hlavac/GallardoPerez-TR-2009-03.pdf

Thesis Advisor: Hlaváč Václav, Průša Daniel;

This research work was supported by the CVUT Media Lab foundation, and the project CAK.

Research Reports of CMP, Czech Technical University in Prague, No. 3, 2009

Published by

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

2D Grammar Extension of the CMP Mathematical Formulae On-line Recognition System

Eva Gallardo Pérez

January 21, 2009

I would like to thank my advisors Václav Hlaváč and Daniel Průša for providing the motivation and the resources for this reserach to be done.

In addition, I would like to thank Michal Havlena, Michal Jancosek, Jef Vandemeulebroucke and Marian Uhercik for their helpful suggestions for improvements to the thesis, Akihiko Torii, Vladimir Smutny and Pavel Krsek for their willingness to help me solve unexpected problems during my work and Eva Matyskova and Radka Kopecka for being supportive colleagues.

I would like also to thank my parents, J.Eduardo Gallardo and Purificación Pérez, my grandparents Pedro Pérez, Modesta Arcediano and my sister Raquel Gallardo for their unwavering support during this work and in all my academic pursuits.

Finally, I would like to express my deepest gratitude to Baptiste Breda, Felix Dent, Florence Reith and Yanick Slikboer for all the encouragement and support they have provided throughout this project.

Abstract

In the last years, the recognition of handwritten mathematical formulae has recieved an increasing amount of attention in pattern recognition research. However, the diversity of approaches to the problem and the lack of a commercially viable system indicate that there is still much research to be done in this area. In this thesis, I will describe the previous work on a system for on-line handwritten mathematical formulae recognition based on the structural construction paradigm and two-dimensional grammars. In general, this approach can be successfully used in the anaylysis of inputs composed of objects that exhibit rich structural relations. An important benefit of the structural construction is in not treating symbols segmentation and structural anaylsis as two separate processes which allows the system to perform segmentation in the context of the whole formula structure, helping to solve arising ambiguities more reliably. We explore the opening provided by the polynomial complexity parsing algorithm and extend the grammar by many new grammar production rules which made the system useful for formulae met in the real world. We propose several grammar extensions to support a wide range of real mathematical formulae, as well as new features implemented in the application. Our current approach can recognize functions, limits, derivatives, binomial coefficients, complex numbers and more.

Contents

List of Figures

1 Introduction

The input of mathematical expressions into computers is more difficult than the input of plain text because special symbols are present, symbols have different size, formulae have a rich internal structure given by mathematical conventions and the spatial arrangement in a plane. The handwriting recognition yields a natural, practical way of entering mathematics into a computer system.

We present a method for on-line mathematical formulae recognition that can be successfully used in the analysis of images containing objects that exhibit rich structural relations based on the structural construction paradigm and 2D-grammmars. An important benefit of the structural construction is in treating the symbol segmentation in the image and its structural analysis as a single intertwined process. This allows the system to avoid errors usually appearing during the segmentation phase.

This thesis contributes to the incremental work on a system for on-line mathematical formulae recognition. Principles of the system have been already published as well as a pilot study supporting very basic mathematical constructs has been implemented [18, 19]. The main goal of the thesis is to extend the application to be able to cope with more realistic formulae, see Figure 1.1

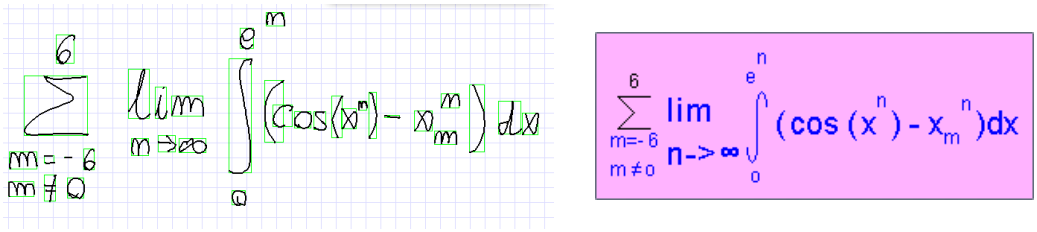


Figure 1.1 Example of a successful recognition of a complex mathematical formula.

1.1 Motivation

The problem of recognition of handwritten expressions has long been a focus of study in the field of pattern recognition. Research in this area has been driven by a desire to combine the natural advantages of handwritten input, including a simple interface, with the data processing capabilities of computers. There exists a number of commercially successful products which are available to recognize a user's handwriting and use this ability to perform simple tasks such as scheduling appointments and writing memos. However, most scientists and engineers are

unable to take advantage of these products for their technical work due to the lack of effective algorithms for interpreting more complex handwritten expressions, particulary equations, diagrams, graphs, and other mathematical forms. Compared to the effort put into the recognition of printed and cursive prose, the recognition of more complex forms has received only minor attention in pattern recognition research. In addition, the diversity of approaches to the problem and the lack of a commercially viable system indicate that there is still much research work to be done in this area.

As more powerful computers with better displays and input devices become available, demand will increase substantially for software systems which can work with the type of handwritten data that one would find in a research notebook or technical document. Mathematical expressions are a natural place to begin such research since they are critical to all technical writing and there already exists a vast amount of literature on recognizing handwritten letters and words, major subcomponents of these expressions. Combining mathematical expression recognition capabilities with existing algebra solving software, graphing programs, and simulation systems would be a firtst step towards a superior user interface for doing technical work with a computer.

1.2 Overview

Formulae contain significant structural information which is expressed as 2D spatial relations. Thus, mathematical formulae recognition is an appropiate application area for testing *2D grammars*-based approach. In a mathematical expression, characters and symbols are spatially arranged as a complex two-dimensional structure, possibly of different character and symbol sizes. This makes the recognition process more complicated even when all the individual characters and symbols can be recognized correctly. Moreover, to ensure that a mathematical expression recognition system is useful in practice, its recognition speed is also an important issue to consider. On the other hand, mathematical formulae follow quite strictly established formalism. This formalism opens the door to use tools from the theory of formal languages on both syntactic and semantic level.

Recognition of mathematical formulae can be basically divided into two groups: *off-line* recognition and *on-line* recognition.

By off-line recognition we mean recognition of scanned formulae. In this case, the input is a picture, a two-dimensional array of pixels. Scanned pictures typically contain printed formulae, but handwritten expression can be considered as well. This method serves to digitize scientific documents or notes.

On the other hand, on-line recognition is being used to process handwritten formulae entered through tablets. The input is a sequence of strokes. Elements of the sequence are sorted by the time of creation. Each stroke is a sequence of points written by one move of the pen. Since the pen is driven by hand, on-line recognition focuses on handwritten formulae.

In this thesis, we deal with on-line formulae recognition. There are many contributions concerning this topic, several mehtods have been designed or adopted for the formulae recognition, a nice survey can be found in [8].

Most of the known methods for mathematical formulae recognition, like a typical system of pattern recognition, follow a two-phase procedure:

1. *Symbol recognition phase*: Detection of individual symbols by image segmentation and labeling symbols using pattern recognition techniques.
2. *Structural parsing phase*: Structural analysis of relations among labeled symbols.

1.3 Thesis structure

In this thesis, I focus on examination of a general formulae recognition process and used methods, which are implemented in real interface applications, and in the structural analysis aspect of mathematical expression recognition.

Firstly, a general description of formulae recognition process is given in Chapter 2. Afterwards some related known works of others where an application for formulae recognition is available are listed and described in Chapter 3. Main ideas behind our method are briefly summarized in Chapter 4. Some of the general problems that have to be overcome during the structural analysis stage are discussed in Chapter 5. Used grammar and grammar productions for parsing real mathematical formulae are also described in Chapter 5. Finally, I present and discuss experimental results, as for example Figure 1.1, which are then followed by some concluding remarks (Chapter 6, 7, 8).

2 General Description of Formulae Recognition Process

Formulae recognition typically consists of two major stages: *symbol recognition* and *structural analysis*.

The symbol recognition stage is precessed by *segmentation*. The goal of the segmentation is to divide black pixels of the input picture (off-line recognition), respectively strokes (on-line recognition) into groups, where elements of a group form exactly one character. It is typically done by comparing coordinates of the elements and grouping the nearest ones together. Character recognition is done for these groups. If the results of character recognition indicate that a grouping of elements was not correct, the grouping can be modified and the character recognition done again (so, more iterations 'group elements, recognize characters' can be possibly done during the symbol recognition stage). There are also methods that perform the segmentation and character recognition simultaneously.

The output of the first stage consists of detected symbols and also their coordinates and sizes (i.e. each occurrence of a symbol is assigned by the size and coordinates of the bounding rectangle, coordinates of some important feature points of the symbol, etc.). For each segmented symbol, more than one possibility which characters it matches best can be computed (each possibility being assigned by probability or penalty). Such data are the input for the structural analysis that tries to find out relationships among the symbols and builds a derivation tree representing the structure of the input expression. To achieve this, several grammar formalisms have been developed or adopted. They include formalisms resembling extensions of two-dimensional context-free grammars. Graph grammars appear also quite often.

A grammar oriented approach is connected with a usage of some parser, but not all methods of the structural analysis are based on parsing. Some authors have developed systems, where rules for building a derivation tree are procedurally coded.

Both stages, parsing and character recognition, are completely separated and independent. The approach proposed by M.I. Schlesinger in [20] is unique. It performs the stages simultaneously by one process. A similar approach has also been studied in [4]. It seems, there are not many other systems based on this idea mentioned in the literature.

2.1 Structural Analysis

As has been already mentioned, most of the methods for the structural analysis of mathematical formulae are grammar based. Here are two examples of grammars used in some systems:

In [14], so called *geometric grammar* are considered. These grammars allow to define the following relationships among the symbols: up/down (one symbol is under another one), left/right, in/out (convenient to describe e.g. the square root symbol having an operand inside it), upper-right/lower-right (e.g. to describe 'power of' relationship).

The second example comes from the system described in [17], where parsing is based on *graph grammars*. Assuming the segmentation and recognition of symbols have been done, a graph for the recognized symbols is constructed. Nodes of the graph correspond to the symbols. Two symbols are connected by an evaluated edge if the two symbols can be in some direct relationship in the formula (e.g. one symbol is a subscript of the other). This decided by comparing coordinates of the symbols. Edges are evaluated by this type of the relationship. Nodes are labelled by the symbol, they correspond to.

Structure of an expression is described using the graph grammar. This grammar is context sensitive. Rules of the grammar transform a subgraph into a single node. These nodes are labelled by nonterminals. The parsing is performed by a bottom-up approach. Nodes of the created graph are being reduced till there is only one node (labelled by the initial symbol of the grammar).

A different, non-grammar based, method for structural analysis was proposed in [7]. In this case, a network (graph) is constructed. Nodes represent detected symbols. They are linked by several edges with labels and costs representing possible relations of the pair of symbols. The network can have multiple edges between two nodes. They represent the ambiguity of the decision of he relation between the symbols. Edges are calculated based on the distance between the symbols and their position. The result of the recognition corresponds to the spanning tree with minimum cost.

2.2 Segmentation

A survey of character segmentation methods is given in [2]. The methods used for the off-line recognition can be divided into the following groups:

- Classical approaches, in which segments are identified based on character-like properties. The picture is being cut into components that most probably form a character. Decisions how to cut are made by evaluating neighborhoods of black pixels, shapes of connected areas these pixels form, etc.
- Recognition-based segmentations (template matching), the picture is searched for components that match known characters.
- Methods, in which words (formulae) are being recognized as a whole. In this case, the segmentation and character recognition are merged into one process.

The third group of methods is mostly represented by *Hidden Markov Models* (HMMs). These models were succesfully applied in the area of speech recognition and thus adopted by some authors to be used in *Optical Character Recognition* (OCR) systems for which they provide the advantage of a simultaneous segmentation and character recognition. An example can be found in [9]. HMM is a statistical model. It requires training data to be trained on. Moreover, suitable features need to be extracted from the input pictures before they are processed by HMMs.

In general, the segmentation is easier in the case of on-line recognition (in contrast to off-line recognition) since the pixels are already divided into components (strokes). It is sufficently to group only strokes that should form a character together. The algorithm finding the minimum spanning tree appears in the literature to solve this task. (An example can be found in [14]). The segmentation is done as follows. Each stroke is assigned by a coordinate corresponding to the center of the stroke. A graph is constructed: for each stroke, there is a node representing it and each two strokes are connected by an edge evaluated by the distance between strokes' centers. Minimum spanning tree is found in the graph. During the segmentation, only strokes neighboring in the spanning tree are considered to be possibly parts of one character.

2.3 Recognition of Segmented Symbols

Methods for character recognition can be categorized as follows:

- Template matching
- Structural methods
- Statistical methods
- Neural network based methods

The first method compares an input image respectively a group of strokes to be recognized to templates representing particular symbols. Using some metric, the distance between the input and a template is measured and the template matching best is found.

Structural approaches look for a presence of selected features in the input. The considered features are like the number of corners, loops, etc. Result of the recognition is calculated based on the detected features. A decision tree can be build to help in this process. Structural approaches are rare, there are only few systems based on such ideas.

Statistical and neural network based methods seem to be more common than the previous two methods. They are suitable for recognition of printed as well as handwritten characters. Both methods are very similar and share mostly all stages of the recognition process. The difference between them is in usage of a statistical, respectively neural classifier. In general, the methods can be described as follows:

- Classifier is trained on a set $\{t_i, c_i\}_{i=1,\ldots,n}$, where t_i are input pictures respectively groups of strokes, each of them containing a symbol belonging to class c_i.

- Trained classifier takes an input x and finds a class c the input fits best.

Usually, some important features are extracted from an input and they are passed to the classifier instead of the whole input. Input pictures for off-line recognito can be of different size, they also contain a huge number of pixels, these are the reasons why feature extraction is needed.

Choosing a good type of features to extract seems to be an important part determining the quality of the method. Requirements on the features can be summarized as follows:

- The number of features should not depend on the size of the input.
- Visually similar inputs should lead to similar features.
- The features should be capable of discriminating among the given classes of pictures.

One of the advantages of the statistical, respectively neural based methods, is that the feature extraction as well as the training process are independent on particular characters. This is not true in the case of template matching, respectively structural methods, where a template, respectively a set of rules, needs to be designed for each symbol.

3 State-of-the-art

3.1 Overview

Interest in developing pen-computing applications has been growing significantly during the last years, due to, among other factors, the introduction of devices such as Personal Digital Assistants (PDAs) or Tablet PCs. The main characteristic of such devices is that they use the stylus as input tool, being a 'natural' substitute for keyboard and mouse.

The data generated by users wirting with a stylus on an electronic device is known as *digital ink*, and the process of writing is called *on-line handwriting*. The minimal unit which forms digital ink are *strokes*, which are sequences of pints generated between *pen-up* and *pen-down* events, at regular time intervals.

The main objective in pen-computing is not only handling digital ink 'as is' but also its conversion into another data structure that can be automatically processed by computers.

Character recognition, as the most common type of a symbol recognition problem, has been an active research area for more than three decades [15, 25]. Structural analysis of two-dimensional patterns also has a long history [16].

3.2 Existing Applications

There is a variety of systems based on handwriting developed at research institutions. We could mention these as recent work on user interfaces for on-line handwriting:

3.2.1 The Natural Log System

The Natural Log system is a user-dependent system developed by Matsakis [14, 13], (see Figure 3.1).

To classify on-line symbols, he constructs a high-dimensional normal distribution, which describes the population of each class. A symbol label corresponds to the class that has the maximum probability. Low probability values are used to reject symbols which can represent potential errors in the handwriting. The procedure to recognize a given mathematical expression begins by finding an optimal grouping of the written strokes into isolated symbols. The final grouping of strokes is determined by evaluating all possible groupings and taking the one which minimizes a sum-cost function. This function is the sum of the log likelihood of the classifier output of each symbol in the current partition. To make the optimization of the cost function manageable, its evaluation is constrained by the

minimum spanning tree of strokes, considering the centers of strokes' bounding boxes as nodes of a completely connected weighted graph. Different combinations of subtrees of the minimum spanning tree are evaluated and the optimal one is taken as the final segmentation result.

The structural analysis in this system consists of locating a 'key' symbol usually an explicit mathematical operator. Once the key symbols are located, the parse algorithm proceeds to find their corresponding operands, and partial subexpressions are formed. The procedure is applied recursively until no more key symbols are found. The algorithm is extended to support parsing of superscripts, e.g. to non-explicit operators, but no support for subindexes is offered.

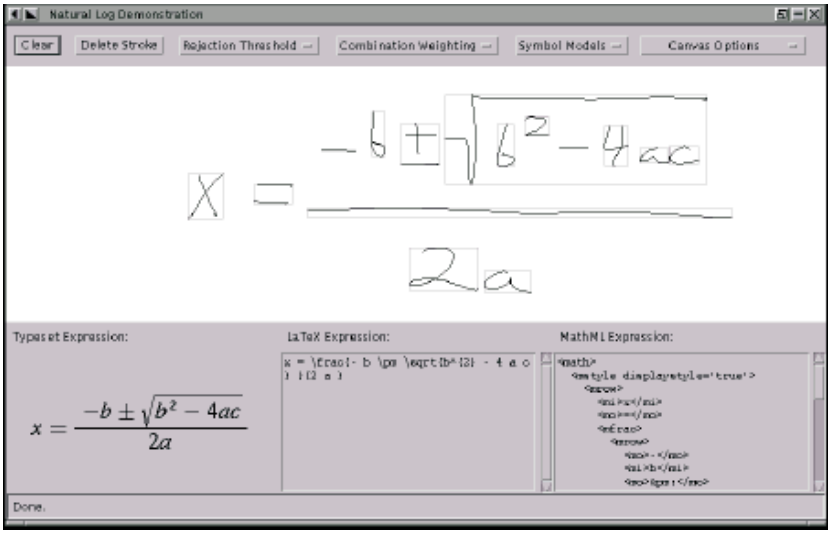


Figure 3.1 The Natural Log System.

3.2.2 Freehand Formula Entry System

The FFES is a pen-based equation editor developed by Smithies and Norvins [21, 1].

The classification of symbols is done by using the nearest-neighbor method. The developers use confidence information supplied by the classifier to group strokes. Their method proceeds by generating all possible combinations of a fixed number of strokes (by default they take a maximum of four strokes), which potentially can constitute a single symbol. Once a single symbol is classified, the confidence level of a combination corresponds to the lowest output of the classifier. Finally, the group with the highest confidence is taken and the first symbol in the group is returned and considered a correctly recognized character. The procedure is repeated, once again, when a fixed number of input strokes is reached. For the structural analysis, they first used a method based on graph rewriting, Zanibbi [27] modified the program to use a structural analysis method developed by him, (see Figure 3.2).

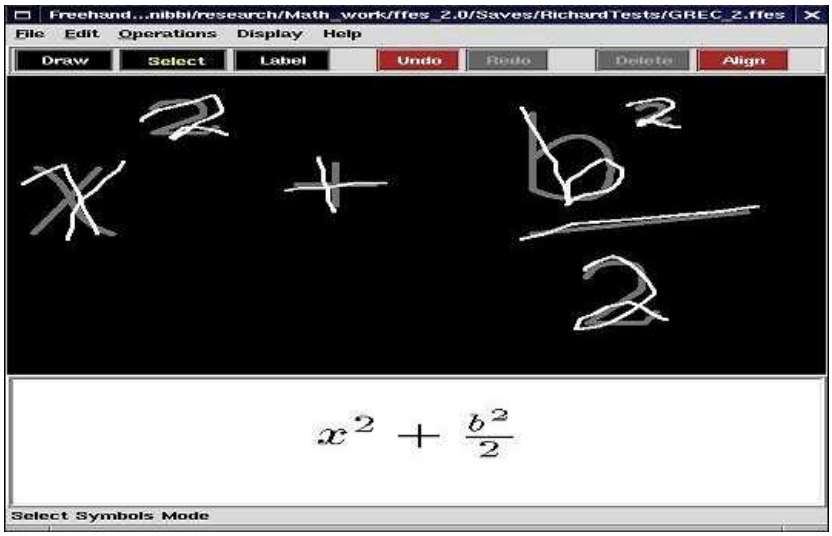


Figure 3.2 Free Hand Formula Entry System.

3.2.3 Infty Editor

Infty Editor [23, 22] is a system specialized for creating mathematical documents, which contains a real-time recognition system for mathematical expressions.

The recognition system combines segmentation and recognition of characters to remedy difficulties in structural analysis due to irregular symbol position and size. The rewriting puts symbols into extendable symbols and non-extendable ones. The former can be extended to form other symbols by adding more strokes, while the latter cannot. For example, F can be extended into E. When a stroke is classified as non-extendable, the classification result is rewritten by the computer in the drawing area using a predefined prototype. If a stroke is classified as an extendable character, the system waits for the next strokes. The classification result is written automatically if a predetermined time interval has elapsed or the expected number of strokes is reached.

3.2.4 MathJournal

Wenzel and Dillner [26] describe another system, MathJournal. The interface is very similar to Microsoft's Journal program which is included in the operating system, (see Figure 3.3). The recognition capabilities of this program are similar to the ones of MathJournal. It operates as a normal pocket calculator, the operations are done after recognizing a handwritten arithmetical expression.

MathJournal uses the recognizer integrated in the operating system for the classification of isolated handwritten characters. Although it is possible to recognize special mathematical symbols and constants, they are limited to the ones recognized by the Microsoft API.

Relevant aspects of this system are its 'solution engines'. They process the recognized expressions in numeric, graphic, or symbolic formats. Diagrams, such as function tables, are processed and plotted by using curly braces and arrows

as gestures. Similar gestures are used for the solution of equation systems or for plotting functions.

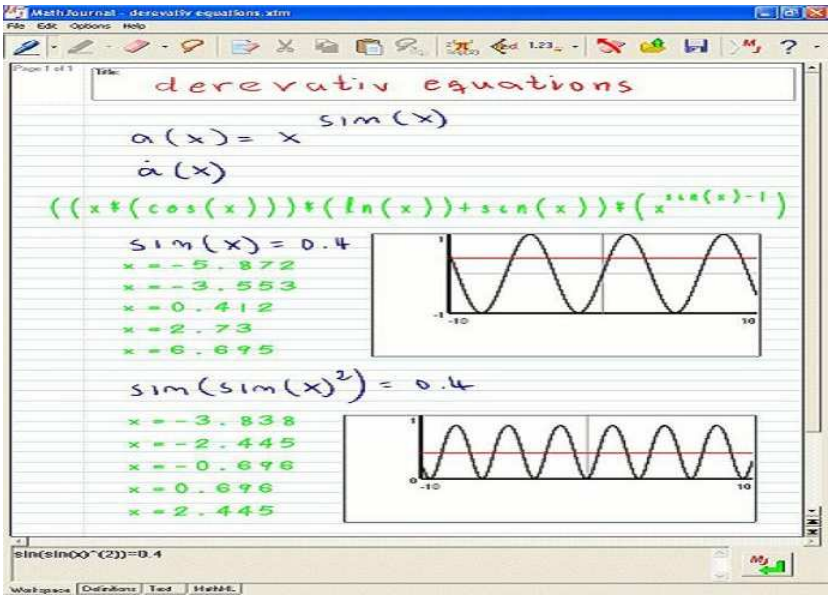


Figure 3.3 MathJournal.

3.2.5 MathPad

MathPad [12] is a system for the creation and exploration of mathematical sketches. The main objective in this user interface is to facilitate the user-editor interaction using sketches.

A set of gestures is used to interact with the recognition and processing engine included in the system. The basic interaction is the drawing of a lasso gesture followed by a tap. This action requests the system to recognize the selected strokes. It is used also for the association of variables and stroke grouping. Drawing the equals sign followed by a tap or by the minus sign allows the solution of equations or factorization of expressions. Other gestures are used for deletion of ink, association of recognized expressions, and angle association, (see Figure 3.4, show the different gestures used in MathPad).

The system is writer-dependent to guarantee a more accurate symbol recognition. Handwriting is recognized using hybrid recognizer. First, a dynamic classifier which calculates the similarity with prototypes is combined with a statistical classifier, to obtain information to use once again dynamic programming for the fine classification.

3.2.6 PenCalc Project

PenCalc is a handwriting-based calculator program that can recognize handwritten mathematical expressions and perform calculations accordingly [3]. Recognition is limited to simple mathematical expressions, (see Figure 3.5).

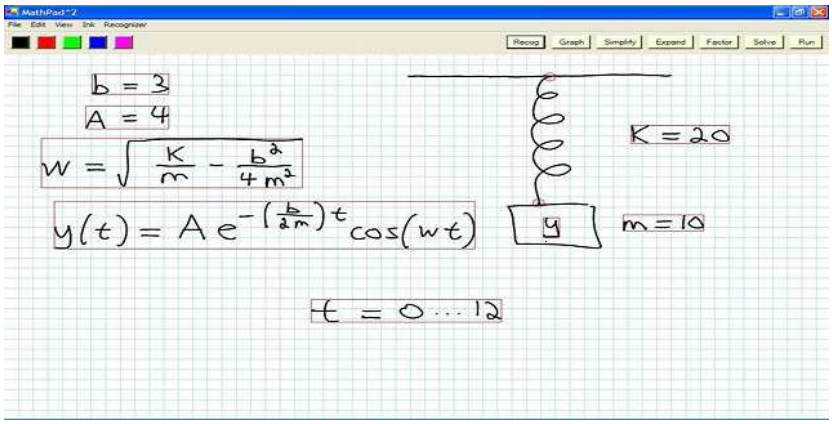


Figure 3.4 MathPad.

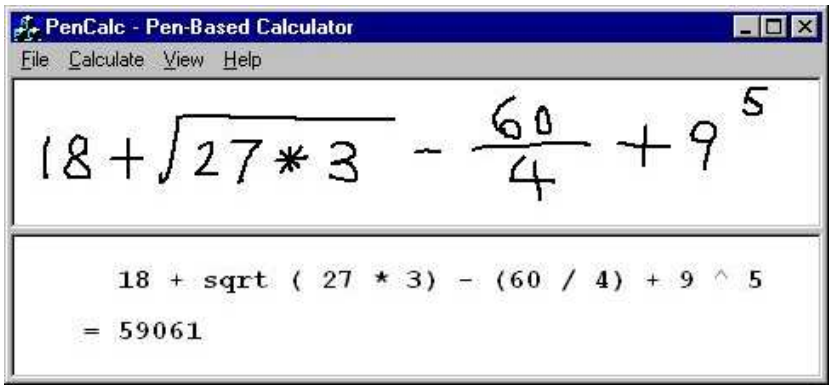


Figure 3.5 PenCalc Project.

3.2.7 MathForEngine

MathFor is a set of Java libraries recognizing mathematical notation from digital-ink documents [24].

4 Notes on Our System (Previous Work)

The presented method for on-line mathematical formulae recognition is based on the structural construction paradigm and two-dimensional grammar as presented in [20]. We use a kind of 2D grammar to express spatial relations among elementary mathematical symbols, and a parsing that ensures we obtain a syntactically correct 2D structure. The segmentation and resolution of ambiguities that can arise during symbol recognition are fully driven by the parsing algorithm. This is one of the main achievements and the main difference compared to other existing methods. Our *elementary symbol detection phase* searches for each group of strokes which can form an elementary symbol (we use OCR based on elastic matching). It is done without any knowledge of the formula structure. The result is a set of elementary symbol candidates where particular candidates can share any number of strokes. The *structural analysis phase* takes these candidates and decides which of them really represent a symbol in the written formula and determines mathematical relations among them. Parsing is polynomial in the number of strokes and thus makes it possible to have a responsive implementation. Details on the method are described in [19].

So far, the system supported only few simple mathematical constructs as: binary operations, power to operator, squares, fractions and subscripts. Moreover, most of them were only supported in a limited way. In this thesis, I report the grammar extensions we made to support a wide range of real mathematical formulae, as well as new features implemented in the application. Our current approach can recognize functions, limits, derivatives, binomial coefficients, complex numbers and more, see Chapter 6.

4.1 Formulae Recognition Task

The task of the formulae recognition is to produce a tree over elementary symbols that represents the formula structure in the input sequence of strokes. An example is shown in Figure 4.1.

Requirements for the used method are to be able to deal with the following situations:

- Symbols touching vertically or horizontally.
- Symbols split into several components.
- Ambiguities.
- Misplaced symbols.

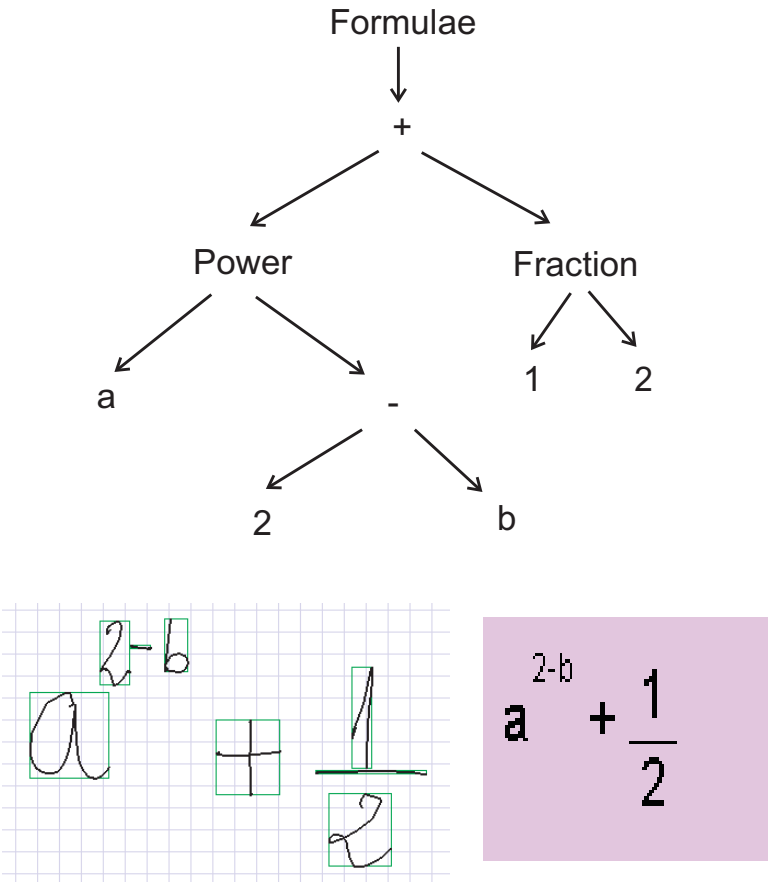


Figure 4.1 An example of an input and its output of recognition.

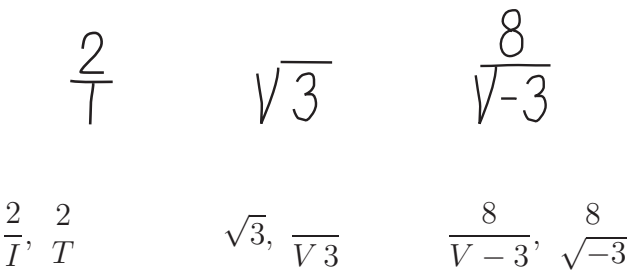


Figure 4.2 Examples of some possible segmentation ambiguities. The pair under each hand-written formula describes an expected interpretation followed by an incorrect interpretation.

These cases can make the symbol segmentation hard, as it is demonstrated in Figure 4.2. There are two recognition results bellow each image, the first one obtained after the correct segmentation, while the second one after an incorrect segmentation (the segmented symbols does not form a valid formulae in this case). Figure 4.3 shows another examples causing difficulties for the recognition. Case (a), illustrates a formula with touching. Case (b) split symbols. Case (c) illustrates a fraction line and a minus sign represented by the same symbol, the

meaning is being given by the context. And finally, case (d) shows and additional symbol C included into the formula by a mistake. We requiere our method to exclude such a misplaced symbol and recognize the formula composed of the other symbols.

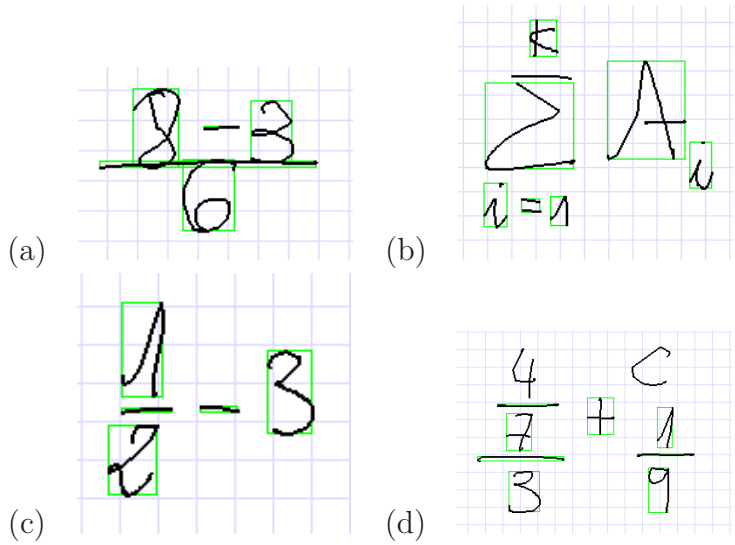
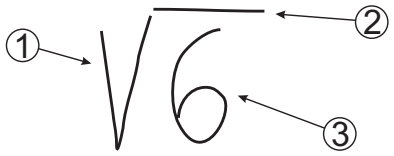


Figure 4.3 Examples of some cases we would like to handle.

4.2 Applied Method

The main ideas taken from the structural construction, which are followed in the used approach, can be expressed in the following way: perform 'rough segmentation' of the input sequence of strokes. For each possible elementary symbol (terminal), find all occurrences of it and, based on the terminal occurences, let the structural analysis decide, the structure of which formula fits the input sequence of strokes best and how does the symbol segmentation looks like. Terminals are detected using an OCR tool and a suitable strategy that chooses rectangular areas of the input for evaluation. For example, up to five terminals can be detected in the first formula in Figure 4.2 variables T and I, number 2 and a fraction line that can be interpreted as a minus sign as well. Each of the occurrences is assigned by a penalty (computed by the OCR tool) determining quality of the recongized symbol. The terminals, represented by their boundind boxes, labels and penalties, are processed by a grammar-based structural analysis. During a bottom-up parsing process, bigger rectangular areas labeled by grammar non-terminals are being derived, each derived area being assigned by a penalty again.

The used software consists of two independent layers. The first layer performs the *terminals detection*, while the second one is responsible for the *structural analysis*. The used structural analysis is driven by a 2D grammar defining suported mathematical formulae. The parsing algorithm is of a general nature and it can



symbol	variable V	number 6	fraction line	minus sign	square root
strokes	1	3	2	2	1, 2

Figure 4.4 Candidates find for a sequence of three strokes.

be used to recognize another types of structures but mathematical formulae if supported by a proper grammar.

4.2.1 Elementary Symbols Detection

A *stroke* is a finite sequence $s = ((x_1, y_1), \ldots, (x_k, y_k))$ where each $x_i, y_i \in \mathbb{R}$.

Informally, a stroke is a sequence of points in a plane. Strokes data are usually produced by tablets. Except coordinates, it is also possible to obtain information on a pen pressure and time of creation, however, we do not use this in our method. A sequence of strokes $S = (s_1, \ldots, s_n)$ forms an input to the recognition process.

The purpose of the elementary symbols detection phase is to detect candidates for the formula elementary symbols. To suppress the large number of possibilities, a strategy has been implemented where only some groups of strokes are evaluated by an OCR tool for on-line character recognition. The strategy is based on the assumptions that one stroke cannot be a part of two or more different symbols and that a symbol is formed of at most 4 strokes. Moreover, we choose only groups of neighboring strokes.

The used OCR tool is a modified existing application, freely available at [6]. It is based on a simple extraction of *features* from the input sequence of strokes, which allows to recognize symbols varying in size. The k-nearest neighbor *classifier* is implemented to classify the extracted vectors.

We use labeled groups of strokes to represent results returned by the OCR tool: V is the set of all symbol names learned by the tool, l corresponds the recognized symbol, p expresses a reliability of the recognition (a lower value implies a bigger similarity to learned patterns) and M is a record storing information on the symbol's base line, logical center and bounding box (its size and coordinates).

A *labeled group of strokes* over S and a set of labels V is a tuple (S', l, p, M) where $S' \subseteq S$, $l \in V$, $p \in \mathbb{R}^+$ is a *penalty*, M is a *metrics record*.

For each S', the tool returns up to 5 best matches. Furthermore, no result where the penalty exceeds a global threshold constant is returned. Finally, we can define a set of candidates $\mathcal{C}(S)$ containing all elements (S', l_i, p_i, M_i) returned by the OCR tool when it runs for each $S' \in \mathcal{C}_1(S)$. A candidates example is represented in Figure 4.4.

4.2.2 Two-dimensional Grammars

A *2D co-ordinal grammar* is a tuple $\mathcal{G} = (V_T, V_N, A_0, \mathcal{P})$, where
- V_T is a finite set of terminal symbols (terminals)
- V_N is a finite set of non-terminal symbols (non-terminals)
- $A_0 \in V_N$ is the starting non-terminal symbol (axiom)
- $\mathcal{P}$ is a finite set of 2D productions

Given a set of candidates $\mathcal{C}(S)$ and a 2D co-ordinal grammar (modelling mathematical constructs in this case), the task of the structural analysis is to incrementally derive new labeled groups of strokes over S and $V_T \cup V_N$. The result is chosen among those derived elements (S', N, p, M) where $N = A_0$. Details on this will be given in Section 4.2.3. Informally, the goal is to minimize p and also the number of points in $S \setminus S'$.

I describe the form of productions in $\mathcal{P}$ and rules of how they are used to derive new elements. There are two types of productions:
1. $(N \to l)$
2. $(N \to A \oplus B, \sigma, \pi, \mu)$

where
- $N \in V_N$, $l \in V_T$ and $A, B \in V_T \cup (V_N \setminus \{A_0\})$
- σ : is a *spatial constraint function*
- π : is a *penalty function*
- μ : is a *metrics record composition function*

Let

$$R_1 = (S_1, l_1, p_1, M_1) \qquad R_2 = (S_2, l_2, p_2, M_2)$$

be labeled groups of strokes. A production of the first type is just a simple renaming. If there is $(N \to l_1) \in \mathcal{P}$, it is possible to derive (S_1, N, p_1, M_1) from R_1. A production of the second type is used to derive a union of two elements: $(N \to A \oplus B, \sigma, \pi, \mu) \in \mathcal{P}$ can be applied on R_1, R_2 if

$$S_1 \cap S_2 = \emptyset \ \wedge \ l_1 = A \ \wedge \ l_2 = B \ \wedge \ \sigma(R_1, R_2) = \text{true}$$

The following labeled group of strokes is derived

$$R = (S_1 \cup S_2, N, p_1 + p_2 + \pi(R_1, R_2), \mu(M_1, M_2))$$

I give some details on σ and π we use in used implementation. Let $\text{box}(S')$ denote *bounding box* of a sequence of strokes S'. For R_1 and R_2, production's spatial constraint evaluates mutual position of $\text{box}(S_1)$ and $\text{box}(S_2)$. First, a rectangle C is computed which of size and position is relative to $\text{box}(S_1)$. And second, it is checked whether some specific point of $\text{box}(S_2)$ (called a *reference point*), determined based on M_2, is contained in C. This is showed in Figure 4.5 which shows an example of R_1 and R_2. The reference point, denoted by F, is required to be located in C.

A constraint can be quite general, we do not require the bounding boxes to touch each other, they can overlap or even more, one bounding box can be included in the other one, as it is used to model the spacial relations between square roots and their arguments.

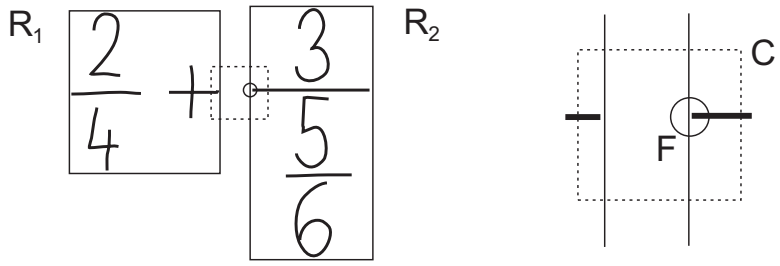


Figure 4.5 Example of a spatial constraint evaluation. On the right, there is a zoomed reference point F and constraining rectangle C. Related production is (BinaryOperation $\rightarrow$ ExpressionFollowedByBinaryOperator $\oplus$ Expression, σ, π, μ).

$\pi(S_1, S_2)$ is computed based on the distance between F and the center of C. Let (f_1, f_2), resp. (c_1, c_2) be coordinates of F, resp. the center of C. Then

$$\pi(S_1, S_2) = w_1 \cdot |f_1 - c_1| + w_2 \cdot |f_2 - c_2|$$

where $w_1, w_2 \in \mathbb{R}$ are production dependent weight constants.

$\mu(M_1, M_2)$ computes a metrics record of the derived mathematical structure in terms of M_1 and M_2.

4.2.3 Structural Analysis. Parsing Algorithm

This section describes the parsing algorithm used by the structural analysis phase:

Let $\mathcal{G} = (V_T, V_N, A_0, \mathcal{P})$ be a 2D co-ordinal grammar and $S = (s_1, \ldots, s_n)$ an input sequence of strokes.

1. Initialize a list $\mathcal{L}$ by all labeled groups of strokes found by the terminals detection phase, i.e. append all elements in $\mathcal{C}(S)$ to $\mathcal{L}$.
2. For each $(S', l, p, M) \in \mathcal{L}$ and each $(N \rightarrow l) \in \mathcal{P}$, append (S', N, p, M) to $\mathcal{L}$.
3. Iterate through elements in $\mathcal{L}$, repeat the following procedure till the end of $\mathcal{L}$ is reached.

   Let $R_1 = (S_1, l_1, p_1, M_1)$ be the current element in $\mathcal{L}$. For each production $(N \rightarrow A \oplus B, \sigma, \pi, \mu) \in \mathcal{P}$ where $l_1 = A$, and each

   $$R_2 = (S_2, l_2, p_2, M_2)$$

   such that

   $$\sigma(R_1, R_2) = \text{true} \wedge S_1 \cap S_2 = \emptyset \tag{4.1}$$

   Let $\overline{p} = p_1 + p_2 + \pi(S_1, S_2)$. Check if $\mathcal{L}$ contains an element

   $$R = (S_1 \cup S_2, N, p, M)$$

   for some p and M.

   3.1 If R is found and $\overline{p} < p$, append $(S_1 \cup S_2, N, \overline{p}, \mu(M_1, M_2))$ to $\mathcal{L}$ and remove R from $\mathcal{L}$, otherwise do nothing.

   3.2 If there is no R, append $(S_1 \cup S_2, N, \overline{p}, \mu(M_1, M_2))$ to $\mathcal{L}$.

Handle analogously the case $l_1 = B$.

4. Compute a list $\mathcal{L}_1$ such that

$$\mathcal{L}_1 = \left\{ (S', A_0, p + \sum_{s \in S \setminus S'} |s|, M) \mid (S', A_0, p, M) \in \mathcal{L} \right\}$$

Among elements in $\mathcal{L}_1$, find that one with the lowest penalty and return it as the result.

To have the description readable, I have omitted implementation specific details speeding-up the algorithm:

To search for R_2 in step 3 effectively, we use a data structure storing points in a plane, supporting to query for all points located inside a given rectangle. About the *orthogonal range searching* is spoken in [5]. A query is processed in time $\mathcal{O}(\log n)$, where n is the number of stored points.

In the implementation, we have extended the condition (1) in step 3 to be more restrictive. In addition to (1), when the applied production does not model a relation where box(S_1) can be located inside box(S_2) (like e.g. a square root argument can), or vice versa, it is required that there is not any stroke $s \in S \setminus \{S_1 \cup S_2\}$ which of all points are located inside box($S_1 \cup S_2$). This requirement substantially reduces the number of derived labeled groups of strokes, on the other hand, it still preserves the correctness of the structural analysis.

During the algorithm, whenever a new labeled group of strokes is appended to $\mathcal{L}$, information on the production and groups of strokes used to derive it is recorded. This helps to construct the resulting derivation tree.

5 Mathematical Notation and Extension of 2D Grammar

There are several typical ambiguities in mathematical notation which have to be taken care of. Following paragraphs give a survey of them together with our proposed approach leading to correct recognition.

5.1 Mathematical Notation: Characteristics and Problems

Mathematical notation is a kind of two-dimensional notation which helps mathematicians to communicate and visualize concepts and ideas. Although the notation is a language used in many areas of science, no formal definition in terms of syntax and semantics as a two-dimensional language exists.

In a mathematical expression, characters and symbols can be spatially arranged as a complex two-dimensional structure, possibly of different character and symbol sizes. All the characters and symbols, when grouped properly, form an internal hierarchical structure. However, proper grouping of symbols in a mathematical expression is not trivial. Firstly, there are two types of symbols. One type includes all basic symbols and the other includes binding, fence and operator symbols. Each type of symbols has its own grouping criteria. Secondly, there are also two types of operators, namely, *explicit* and *implicit* operators. Explicit operators are represented by operator symbols while implicit operators by spatial operators. Thirdly, some symbols may represent different meanings in different contexts. These properties together make their recognition process very difficult even when all the individual characters and symbols can be recognized correctly.

5.1.1 Grouping symbols

Obviously every symbol has its own individual meaning. However, in a mathematical expression, sometimes there is a need for grouping some adjoining symbols together to represent another meaning. For example, the digits 2, 6 and 3 have their own meaning as digits, but if they are of the same size and lie in the same line, they can represent the integer value 263. By varying size and location they can also represent 2^{63} or $2^{6}3$. Similarly, we can group some letters to represent function names, like 'log' or 'cos'. The following are some general rules:

1. Digits together usually form a unit when they are of the same size, adjacent to each other, and written on the same horizontal line.

2. Some letters together may form a unit. Before considering a group of letters as a concatenation of variables representing their multiplicative product, we have to first check whether they together form a function name.
3. Symbols other than letters and digits should be considered as separate units.

5.1.2 Determining relationships among symbols

The presence of some symbols in a mathematical expression may invoke some special grouping methods. The following are three types of such symbols:

1. Some fence symbols, such as parentheses, group the enclosed units into one single unit. For example, $(a + b)$ is a unit which holds the sum of a and b.
2. Some binding symbols, like fraction line, $\sqrt{}$ and $\sum$ dominate their neighboring expressions. For example, in $\sum_{i=1}^{10} i$, the subexpressions 10, $i = 1$, and i are bound to the symbol $\sum$ which together give meaning to the expression as the sum of $1, 2, \ldots, 10$. However, deciding proper relationships among binding symbols and their neighboring subexpressions becomes non-trivial in some nested expressions, like: $\sum_{i=1}^{10} \frac{i}{a+b}$ and $\frac{\sum_{i=1}^{10} i}{a+b}$
3. The ideas of operator precedence and operator dominance can also be used for grouping units. For example, in $a + \frac{b}{c}$, the meaning becomes $\frac{(a+b)}{c}$ if — dominates + in this case.

5.1.3 Explicit and implicit operators

Explicit operators are operator symbols. When consecutive operator symbols exist in an expression, we can apply operator precedence rules to group the symbols into units. However, when those operator symbols are not lined up, we have to use the concept of operator dominance. For example, in $a + \frac{b}{c}$, the meaning is $a + \frac{b}{c}$ due to the fact that the operator + dominates —, when − and + share the same baseline. However, in $\frac{a+b}{c}$, the meaning becomes $\frac{(a+b)}{c}$ since — dominates +, because + is above − in this case. In some mathematical expressions, there are also implicit operators. Implicit operators (also called spatial operators) determine the relationships between symbols simply by their relative positions. For example, in a^{-2}, -2 is the superscript representing the inverse square of a. However, in a_{-2}, -2 is the subscript of a representing only a variable name, and moreover $a - 2$ can be also be used to represent the subtraction of a and 2.

5.1.4 Context-sensitive roles

Some symbols in mathematical expressions may play different roles in different context. Here are some examples:

1. A dot in an expression can be a decimal point or a multiplication operator depending on the position of the dot and its neighboring symbols.
2. A horizontal line may be a fraction line or a minus sign depending on the length of the line and whether there are symbols above and below the line.

3. The same group of characters can sometimes have different meanings in different contexts. For example, dx is a part of the integral notation in $\int x^2 dx$ but it represents the multiplication of d and x in $dx + py$.

In addition, mathematical notation is not completely standardized and many dialects are used by scientists. Some authors try to describe mathematical notation [10] for solving problems of typesetting and for automatic processing of mathematical notation [11]. However, similar to natural languages, it is nearly impossible to design a universal grammar to cover all the dialects. As a result, almost all systems are based on a subset of the mathematical notation only. Additionally, irregular handwriting aggravates the ambiguities described before and makes it harder to group symbols and to distinguish relations among them. It results in layout problems affecting the recognition of the whole expression.

Our goal is to develop a grammar which would be used to recognize without difficulties, in real time and with a suitable user's interface complex mathematical expressions like

$$\sum_{\substack{m=-6 \\ m\neq 0}}^{6} \lim_{n\to\infty} \int_0^{e^n} (\cos(x^n) - x_m^n)\, dx. \tag{5.1}$$

Figure 1.1 shows this formula written by hand as an input to the system and the related recognition result in a printed form.

5.2 Our Approach for Extension of 2D Grammars

The fact that the subject of recognition is a mathematical formula is not used in any special way to aid in its recognition. Indeed, the parsing algorithm would work well for any handwritten inputs exhibiting some implicit structure. A good example could be musical scores or electronic circuits. However, if a syntactic or semantic meaning is to be given to symbols, knowledge of the structure of mathematics must be incorporated into the recognition algorithm. Inspiration for this task may be drawn from typesetting languages, since they were designed with the specific intent of formalizing the relationships that appear on printed pages. Additionally, having a structure based on a typesetting language makes it easier to generate output for existing typesetters.

5.2.1 Two-dimensional Grammar

In this section we give details on how to use 2D grammars formally defined in Section 4.2.2. Many of the ambiguities commented in Section 5.1 can be avoided by ensuring that the input is be parsed using a two-dimensional grammar. In this grammar, each basic mathematical construct is modelled by a production type reflecting spatial relationship among construct's components. The types can be combined with one another to produce more complex formulae. The spatial relationship between any two 'bounding boxes' of symbols is classified as one of 6 qualitative types: in/out, up/down, left/right, upper left/lower right, upper right/lower left, or identical.

The context of a symbol can be then used to determine which character it represents. For example, the characters − and 'conjugate sign' both use the same symbol. However, within the framework of this grammar, a 'conjugate sign' needs to have a simple character below it, while a − may not. These two characters can then be easily distinguished in an expression such as $\sum_{i=1}^{n}(x_i - \overline{x})$. Some characters can be easily confused with others of the same grammar type such as the characters O and 0. In this case, the grammar is unable to disambiguate the interpretation since they are both simple characters.

We will illustrate the mechanism behind our grammar productions through some examples.
General schemes of our fundamental production types are shown in Figure 5.1.

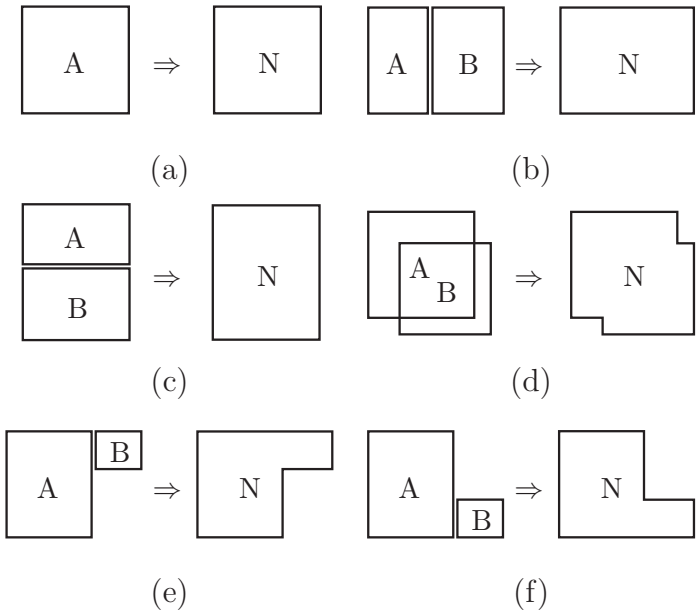


Figure 5.1 Fundamental production types.

(a) $N \longrightarrow A$
To redefine terminals or non-terminals.
example: ' x ',

```
Variable -> [x]
Term -> Variable
Expr -> Term
Formulae -> Expr
```

(b) $N \longrightarrow A|B$
To define left/right relationship (A B).
example: ' x, y, z ',

```
SeqTerms -> Seq|Term
Seq -> Term|CommaD
Seq -> Seq|Seq
```

(c) $N \longrightarrow \frac{A}{B}$
To define up/down relationship.
example: ' $\frac{x}{y}$ ',
`Fract -> Expr/LowerFract*`
`LowerFract -> Line*/Expr`

(d) $N \longrightarrow A\%B$
To define in/out relationship.
example: ' $\sqrt{x}$ ',
`Sqart -> SqrtSign%Expr`

(e) $N \longrightarrow A \hat{} B$
To define superscript relationship.
example: ' x^y ',
`Power -> Term^Expr`

(f) $N \longrightarrow A\_B$
To define subscript relationship.
example: ' x_y ',
`Subscript -> Variable_Variable`

To represent our two-dimensional grammar in a text file, we use the following format and notations:

1. Terminals are in square brackets and non-terminals without brackets.
2. New relations between terminals are redefined by '`->`'.
3. The productions follow one after another, as a list, without any termination sign.
4. The first line contains the name of the initial non-terminal, in our case, it is '`Formulae`'.
5. Sign '`*`' is used to indicate the expression dominance in top/bottom relationships (i.e. to set the baseline of a fraction).
6. It is possible to add line comments, each comment is preceded by '//'.

Using the given 6 types of productions, we can express all common cases of constructs that appear in mathematical formulae. Combining these general forms in the right way and creating new grammar productions leads to grammar extensions that help us to recognize more mathematical constructs. For example, we have extended `Sum` to be able to recognize a sequence of sums or sums with several conditions, see Figure 5.2.

```
Sum->SumExpr|Expr
Sum->SumSimple|Expr
Sum->SumSign|Expr
Sum->SeqSum
SumExpr->Expr/LowerSumPart*
LowerSumPart->SumSign*/Formulae
SumSimple->SumSign*/Formulae
SeqSum->SeqSumExpr|Expr
SeqSum->SeqSumSimple|Expr
SeqSum->SeqSumSign|Expr
```

```
SeqSumExpr->SumExpr|SeqSumExpr
SeqSumExpr->SumExpr|SumExpr
SeqSumSimple->SumSimple|SeqSumSimple
SeqSumSimple->SumSimple|SumSimple
SeqSumSign->SumSign|SeqSumSign
SeqSumSign->SumSign|SumSign
```

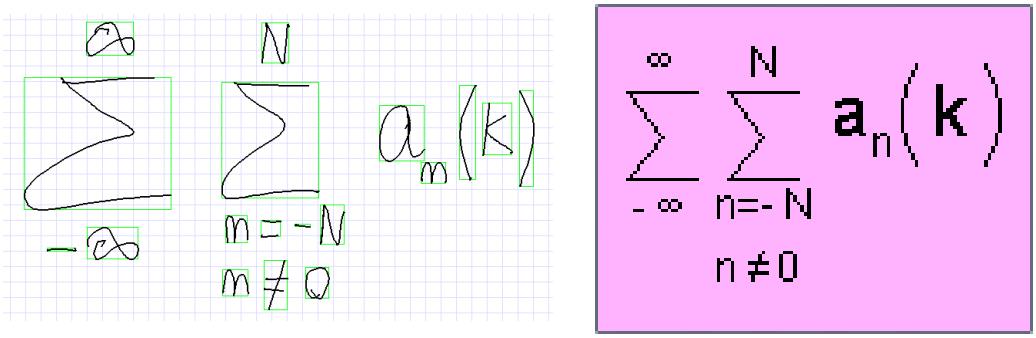


Figure 5.2 Productions related to *Sum* extensions.

We make use of precedence productions together with the sign dominance [*] to group the symbols into units.
The parsing is done with a bottom-up approach. We can demonstrate this on an example. Let us consider formula $2*x+1$. Assuming that all elementary symbols are successfully detected by OCR tool, the parsing proceeds as follows: [$*$] and [x] are combined into $*x$ as a new subgroup, further, this subgroup is extended to $2*x$. Then, [+], [1] are combined into $+1$, and finally, both these groups form $2*x+1$. See the right-hand part of Figure 5.3 for an illustration of the parsing process in the diagram tree.

The presence of brackets, fraction lines, square root signs and other similar mathematical constructs helps us to group units of symbols to larger units as a result of their relationships. For example, in

$$\lim_{n\to\infty} 1+n$$

$\lim_{n\to\infty} 1$ is recognized as a *Limit* and grouped together with $+n$ in a *Binary-Operation*.
On the other hand, in

$$\lim_{n\to\infty} (1+n)$$

$1+n$ is recognized as a unit grouped from an *Expr* and *LimExpr*, *Limit*.

5.2.2 Implementation Details

The application is implemented in *Java*, using *NetBeans* IDE. Grammar definition is stored in one text file having `.gram` extension. We have also implemented support in *NetBeans* that identifies files of this extension and provides basic

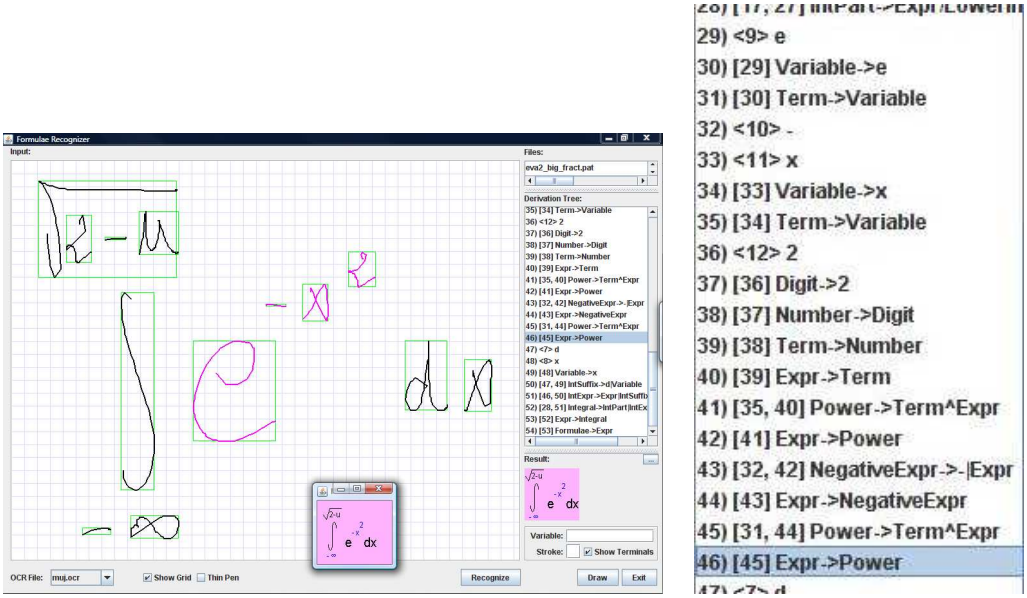


Figure 5.3 Example of structural analysis.

syntax coloring for them in editor. With this coloring support, we have developed a more natural programming style of productions creation. Our grammar is highly comprehensible by any interpret and very efficient from the implementation point of view. The productions are extremely simple as is apparent from the previous snippet of the grammar file.

As for the OCR part, we should emphasize that our system is not limited to a single user. Instead of being invariant to different users' writing styles, we have developed a user adaptive system. By simply selecting the user, the training data to this particular user is applied. This data is collected using a digitalizing tablet or a mouse in a custom application called *OCR Training* (see Figure 5.4). It contains a dictionary of supported elementary symbols. A user writes up to 8 etalons per each of the symbols and produces OCR data file which can be identified and used by the recognition application.

5.2.3 Graphical User Interface

We have developed a comfortable graphical interface. The user can select his personal handwriting style and write mathematical formulae on a grid or plain screen. The system collects the strokes data from a mouse, an electronic-pen, or a tablet, sends the data to the Formulae Recognizer engine, and finally displays results in two forms. Firstly, there is a detailed derivation tree, and secondly, the result is visualized as an image of the recognized expression, typeset in a printed form.

The user interface allow to browse formulae images, run the recongition on them and display results. Except the results, the interface also provides information helping to undersand and tune the process of structural analysis. The system also allows to enter and evaluate queries on the mentioned derivation trees - the

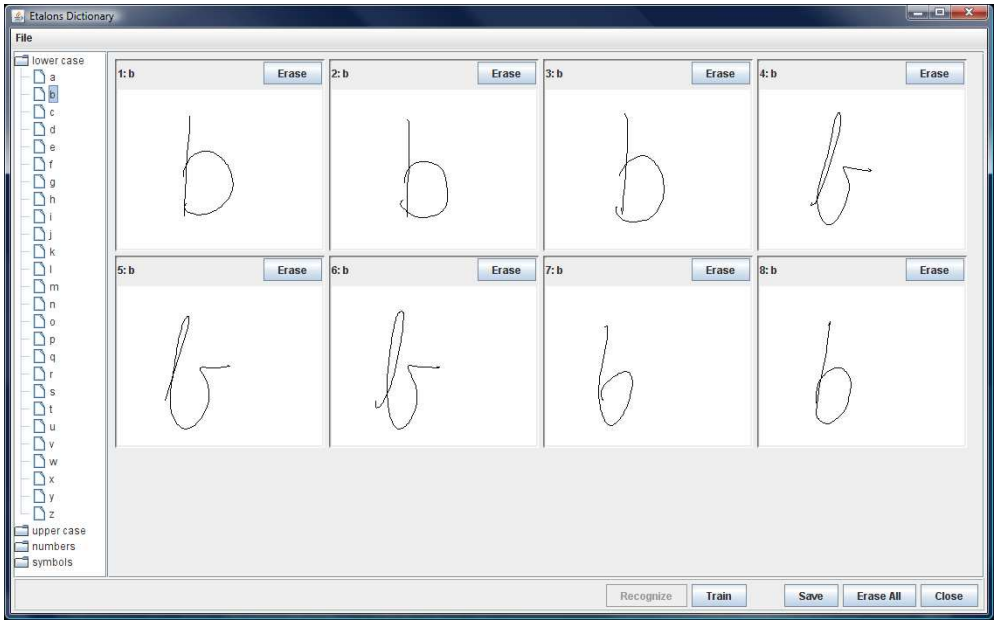


Figure 5.4 OCR-Training.

user can query for derived groups of strokes labelled by a specific non-terminal, penalties of related derivations, etc. This helps us to understand and tune the process of the structural analysis. A screenshot of the application is shown in Figure 5.5.

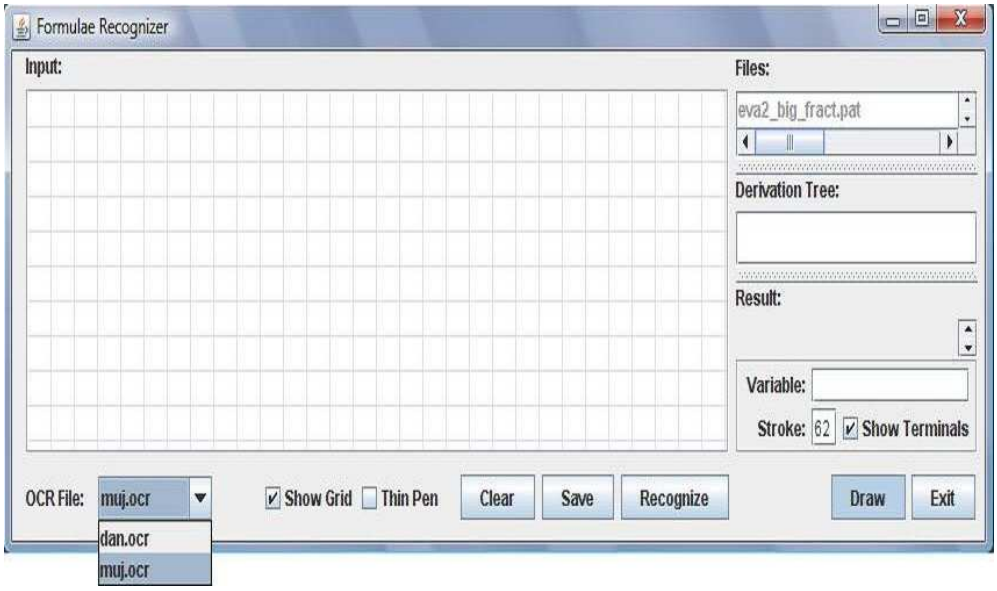


Figure 5.5 Screenshot from the user application.

6 Results

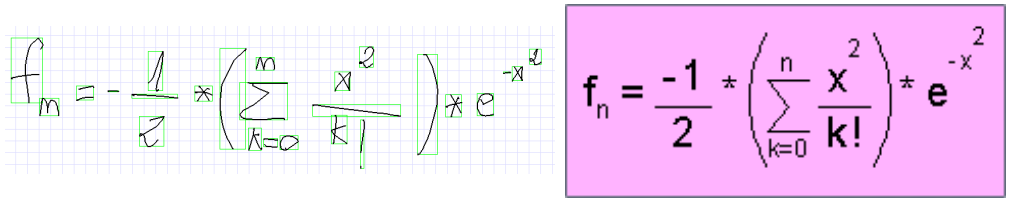
We have developed a grammar for our software that recognizes difficult expressions as we proposed in Expression 5.1. It is possible to recognize functions, limits, derivatives, vectors, complex numbers, binomial coefficients, etc. Figures 6.1, 6.2 and 6.3 show a list of examples which are successfully recognized.

On the other hand, Figure 6.4 contains some examples which sometimes fail to be recognized. Occasional incorrect formulae recognition is due to erroneous grouping, (see (a)) or interpretation of ambiguously placed boxes, (see (b)) or incorrect segmentation caused by overlaps, (see (c)) or misplaced symbols, (see (d)). Recognition is still limited to a mathematical statement written in a single line as the input, (see (e)).

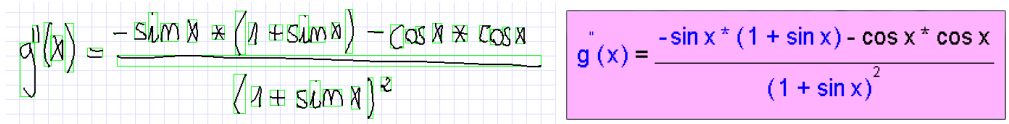
In terms of time complexity, the speed of the recognition depends on the number of input strokes, ranging from 0.5 seconds for easy formulae up to 6 seconds for complex ones, (see Figure 6.1(d) as an example of complex expression in number of strokes and see Figure 6.2(h) as an easy one). The time complexity of parsing is polynomial. This is demostrated in Figure 6.5 and in Figure 6.6. First one shows the times for testing formulas of the form '$a-a-a-a\ldots-a$' and the second one shows the times for different complex formulas. Compared to general exponencial time complexity, the expected time is lower because the algorithm does not process all possibilities, only evaluates some groups of strokes. An increase of the number of productions, i.e. an increase of strokes, (see Figure 6.8), leads to increase of the time of recognition, as shown in Figure 6.7. However there are exception. Some expressions with less productions take more time to be recognized beacouse of time spent on grouping symbols or determining relationships.

Comparing to the original pilot study, there is an increase in time, however, this increase is approximately proportional to the increase in the number of grammar productions. This is consistent with expectations. To achieve a really responsive system, it would require to make the recognition faster (about 10 times). We plan to rewrite our application in C++, this should result in such an improvement.

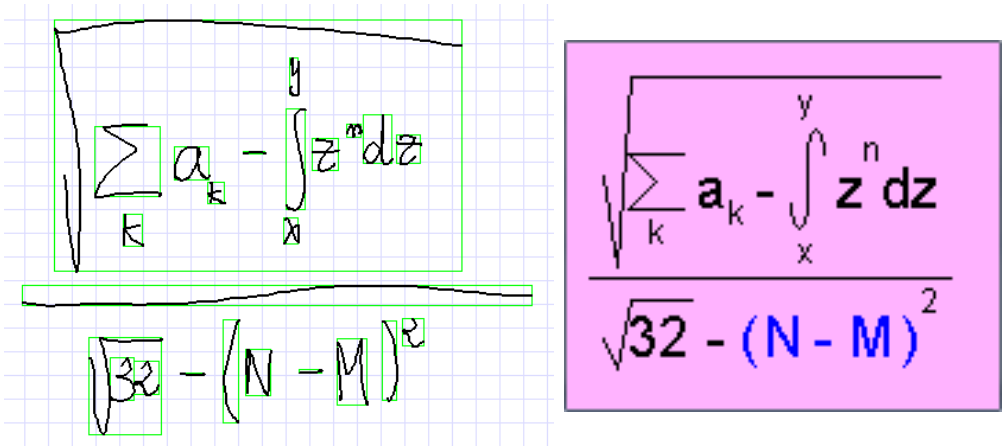
(a)



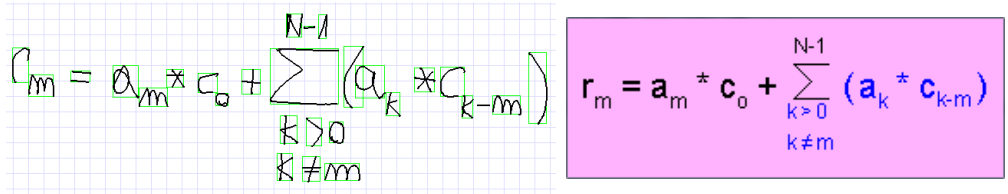
(b)



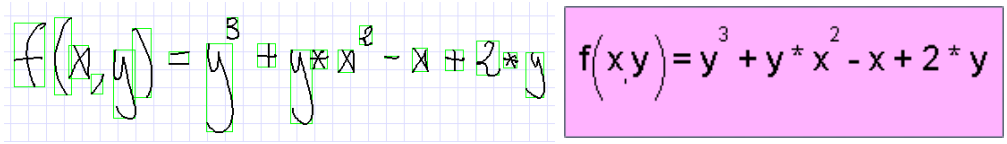
(c)



(d)



(e)



(f)

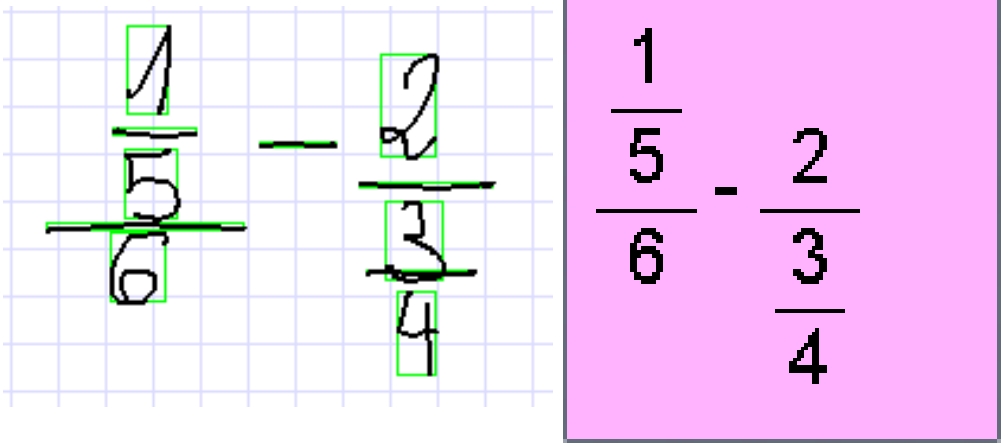
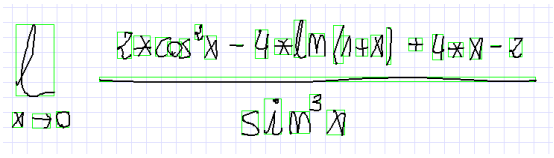


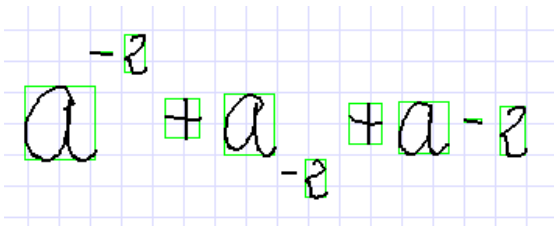
Figure 6.1 Several examples of formulae that are successfully recognized.

(g)



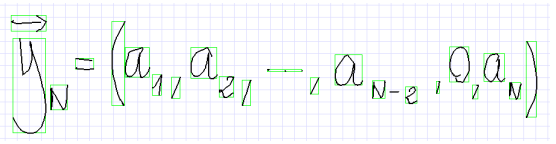
$$\lim_{x \to 0} \frac{2 * \cos^2 x - 4 * \ln(1+x) + 4 * x - 2}{\sin^3 x}$$

(h)



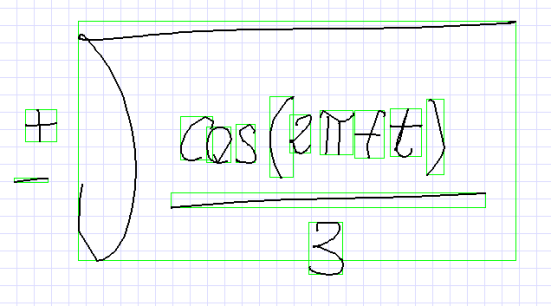
$$a^{-2} + a_{-2} + a - 2$$

(i)



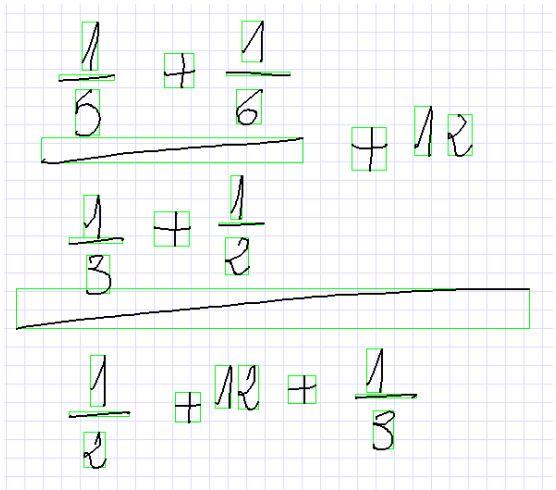
$$\overrightarrow{y_N} = (a_1, a_2, \ldots, a_{N-2}, 0, a_N)$$

(j)



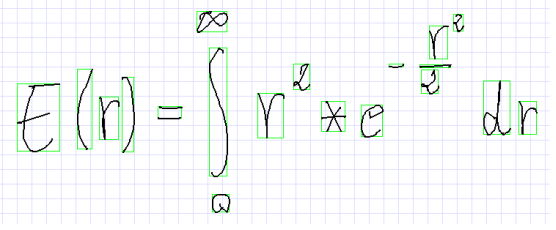
$$\pm \sqrt{\frac{\cos(2\pi f t)}{3}}$$

(k)



$$\frac{\frac{\frac{1}{5} + \frac{1}{6}}{\frac{1}{3} + \frac{1}{2}} + 12}{\frac{1}{2} + 12 + \frac{1}{3}}$$

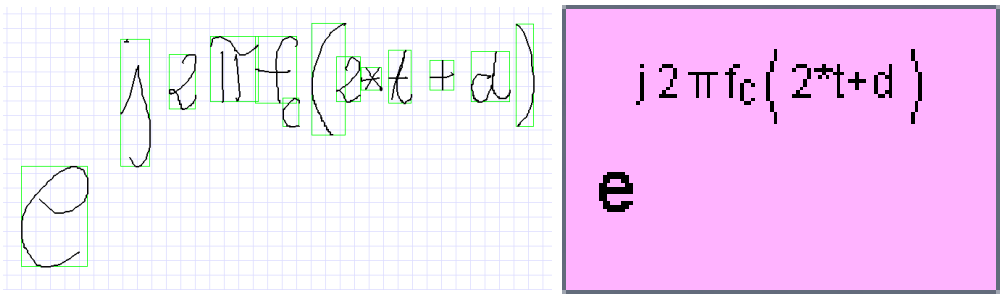
(l)



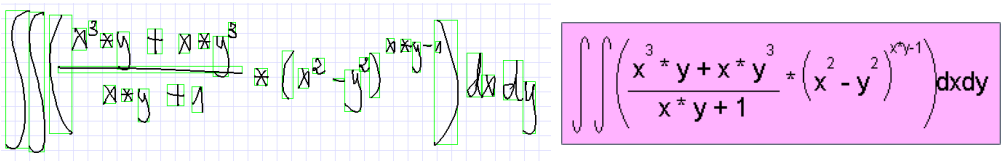
$$E(r) = \int_0^{\infty} r^2 * e^{-\frac{r^2}{2}} dr$$

Figure 6.2 Several examples of formulae that are successfully recognized.

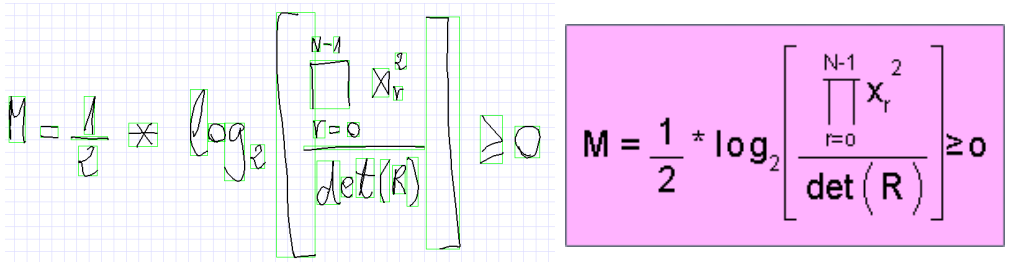
(m)



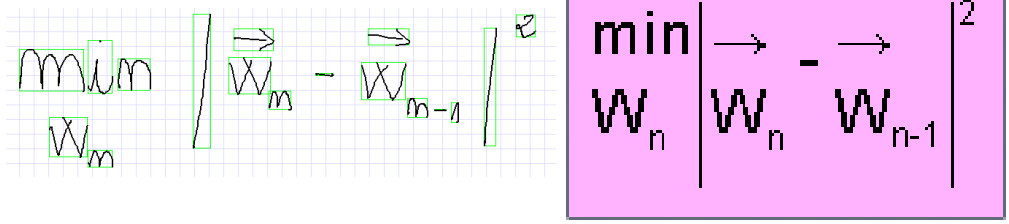
(n)



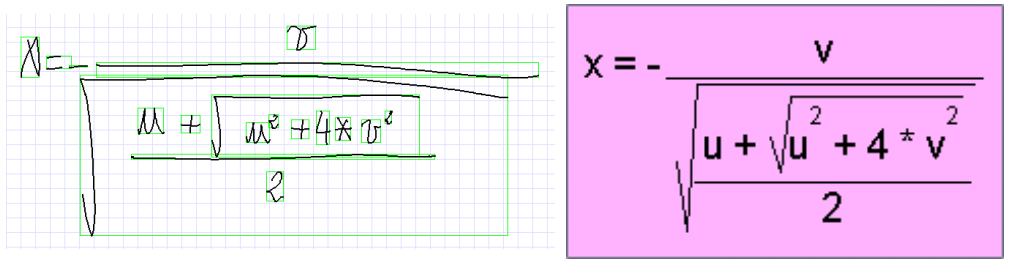
(o)



(p)



(q)



(r)

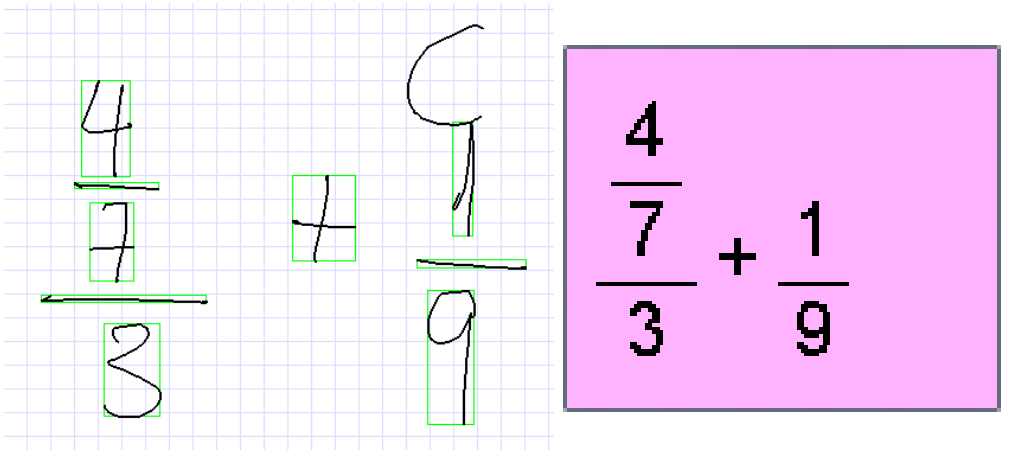


Figure 6.3 Several examples of formulae that are successfully recognized.

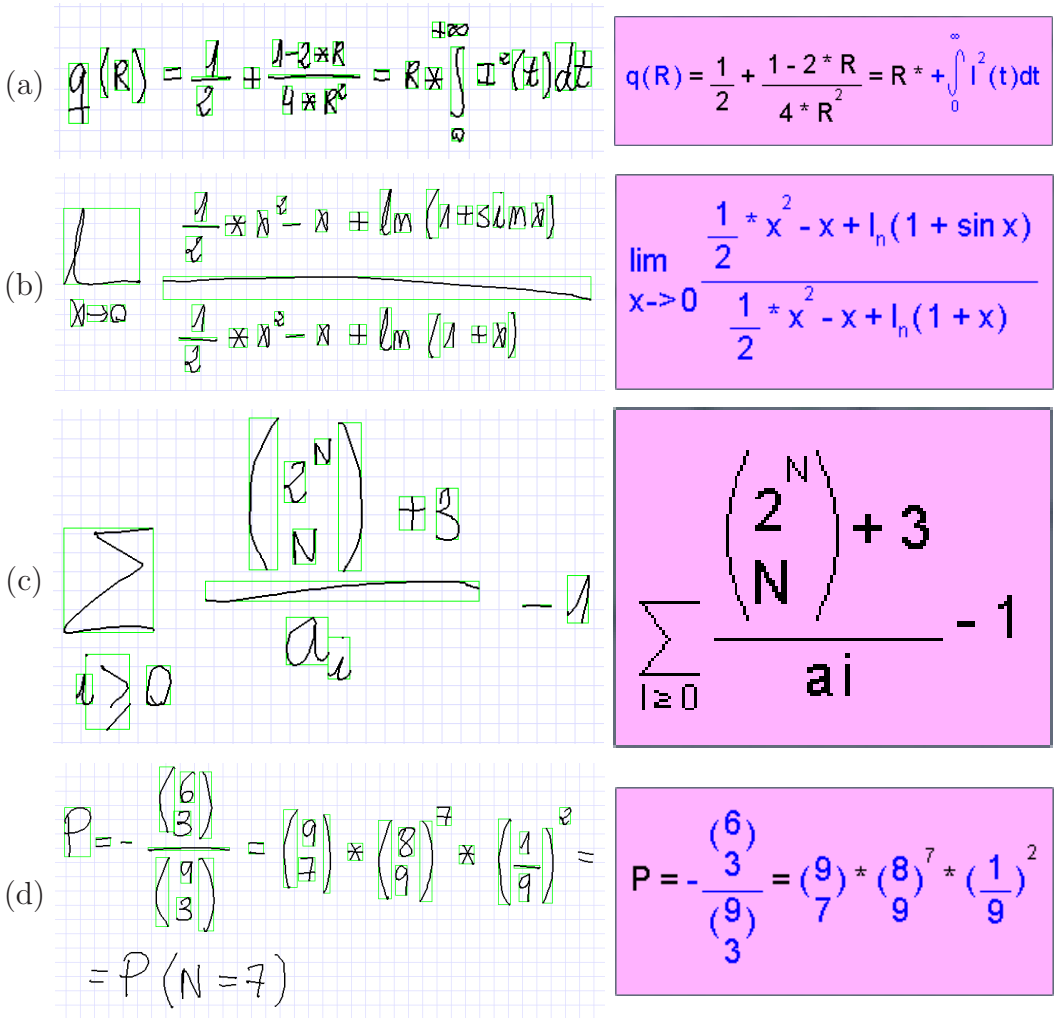


Figure 6.4 Examples of formulae which sometimes fail to be recognized.

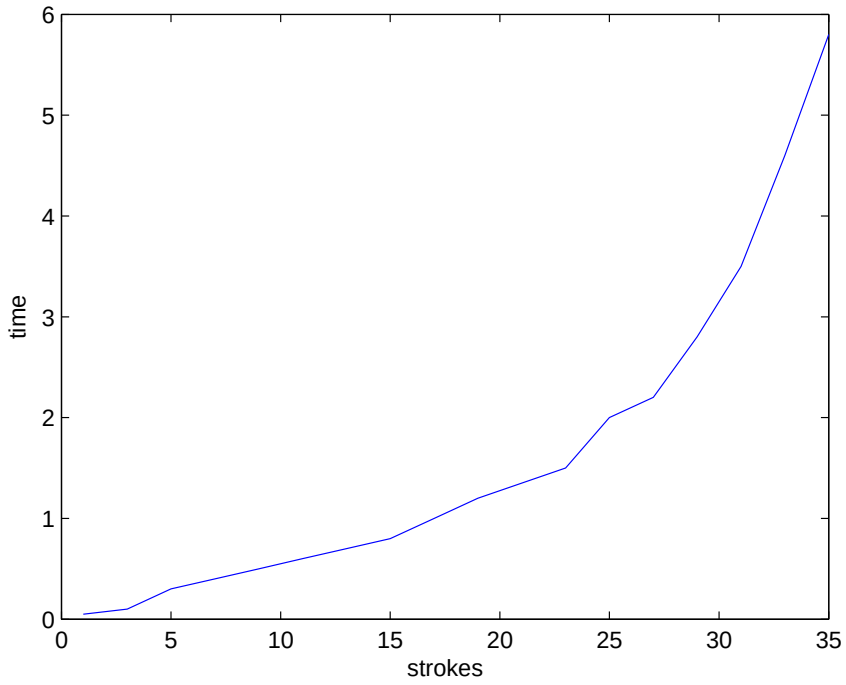


Figure 6.5 Strokes-Time complexity. Testing formulas of the form '$a-a-a-a\ldots-a$'.

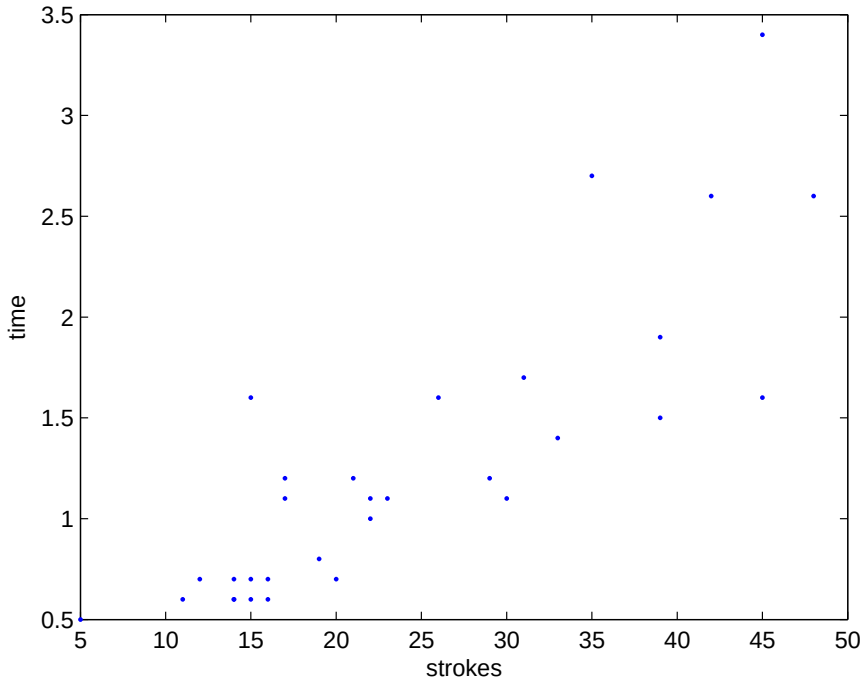


Figure 6.6 Strokes-Time complexity for different complex formulas.

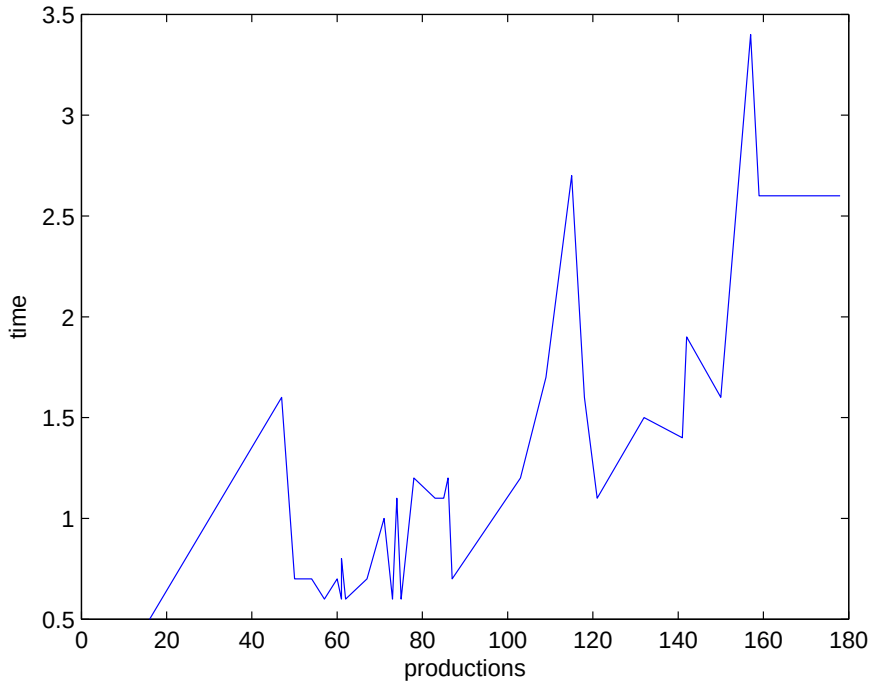


Figure 6.7 Productions-Time complexity for different complex formulas.

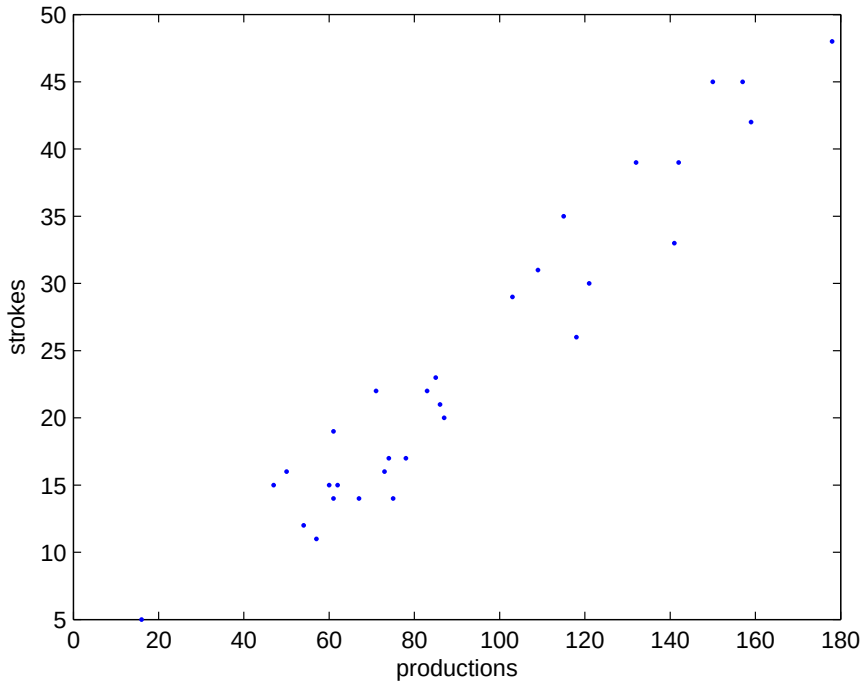


Figure 6.8 Productions-Strokes for different complex formulas.

7 Discussion

If we check our system for the common problems of mathematical notation commented in Section 5.1 we see that all those rules are reflected in the grammar.

To represent a product we require to draw a multiplicative sign, [$*$], e.g. $\cos x * \cos y$ like in Figure 6.1(b). It helps to group symbols to form function names (when possible) rather than a multiplicative product of particular variables. This representation also avoids confusion among groups of characters with different meanings like, for example, $\int x^2 dx$ and $d * x + p * y$ (see Figure 6.2(l)). On the other hand, we are able to recognize a concatenation of variables around π as an expression of a multiplicative product without the need to write $*$, which reduces the writing, e.g. $\cos(2\pi ft)$, (see Figures 6.2(j), 6.3(m)).

In addition, our requirements on the proposed method (to be able to deal touching symbols, symbols split into several components, ambiguities, and misplaced symbols) have been managed (see Figures 6.1(f), 6.3(r)).

However, from time to time, there are problems to recognize 'ln' as a function, since it is confused as subscript l_n, (see Figures 6.2(g), 6.4(b)). This problem can be solved by adjusting internal penalties for deriving the subscript variant. We are working on it.

With our set of productions, our current system is able to recognize common forms like fractions, radicals, binary operations, summations, integrals super and superscripts, functions, binomial coefficients, limits, vectors, complex numbers, i.e. a wide range of mathematical expressions, (see Chapter 6). Although the set of recognized symbols is still limited, most of the input formulae can be successfully recognized. Another limitation is that the system assumes the that expression to be parsed is a single line mathematical statement. So, if the user decides to draw two lines of mathematics, the system will attempt to find a subexpression in one of the lines best fits the input, however meaningless it may be.

8 Conclusions and Future Work

I have contributed to the problematic of the on-line mathematical formulae recognition by an extension of the application to be able to cope with more realistic formulae.

Parsing complex expressions is the area of interest of many researchers and there is a variety of systems that can interpret simple expressions. We proposed and demonstrated a system which incorporates the use of symbol baselines into the parsing algorithm allowing us to interpret even complex expressions.

In this thesis, I have showed that the method of structural construction can be applied for on-line mathematical formulae recognition. It helps to solve many segmentation problems that occur in handwritten formulae. The main contribution to the area of formulae recognition by the system are the following achievements:

- Elementary symbols detection of a sequence of strokes is done during structural analysis (then, no error corrections are needed). We take advantage of the rich formula structure which allows this approach. It is sufficient to evalutate groups of strokes.
- Structural analysis is robut. It is penalty oriented and searches for the formula structure that best matches the input sequence of strokes. It can deal with symbols touching, symbols split into several components and including ambiguities.
- The designed 2D co-ordinal grammar allows to recognize usual mathematical symbols and constructs. It is powerful enough to express the formulae structure. It can be also effectively parsed (thanks to constraints defined via rectangles and the usage of data strucutres for orthogonal range searching).

Moreover, the symbol recognition algorithm is naturally sensitive to variations in writing, our system achieves to solve it with a user adaptative system, adjusting symbol models to particular users.

The conclusions on time complexity are based on experimental results. There is not an exact formula since it depend on many factors, including the number of the symbols detected. Part of the computational time is due to parsing and depends more on the formula complexity than on the implementation. The expected time is based on a polynomial complexity parsing instead of general exponential time. It is lower because the algorithm does not process all the possibilities but only evaluates some groups of strokes. There is an increase in time, however, this increase is approximately proportional to the increase in the number of grammar productions.

Single line formula detection, which is the most immediate limitation of the current approach, could be extended to the detection of multiple lines in our fu-

ture work. The future plans are also to solve the incorrect formulae recognition due to erroneous grouping, or interpretation of ambiguously placed boxes, or incorrect segmentation caused by overlaps, improveing or replacing the OCR tool to achieve a higer OCR correctness rate, and find possibilties of how to learn and tune constraints assigned to productions automatically (so far they have been set manually). In addition, our future plans include improvements on the user interface to fit more as a comercial system and rewrite into C++ what would bring speed up.

The grammar itself could still be improved (extended) in order to recognize more mathematical formulae pushing the application towards its real use by end-users, e.g. mathematicians and physicists. However, I could conclude, that we have implemented a truly useful and efficient system recognizing mathematical handwriting formulae and have shown its performance on several examples including those, that could not be recognized by the previous version of the system.

(The application is available at http://cmp.felk.cvut.cz/ qqperez/GallardoPerez-TR-2009-03/OnLineExprs-Application.)

Bibliography

[1] Dorothea Blostein. The freehand formula entry system. http://research.cs.queensu.ca/drl//ffes/, 2004.

[2] Richard G. Casey and Eric Lecolinet. A survey of methods and strategies in character segmentation. *IEEE Transactions on Pattern Analysis and Machine Inteligence.*, 18(7):690–706, 1996.

[3] K.F. Chan. Pencalc. http://www.cse.ust.hk/pencalc/, 1999.

[4] Philip A. Chou and Gary E. Kopec. A stochastic attribute grammar model of document production and its use in document image decoding. In *First International Workshop on Principles of Document Processing*, pages 66–73, 1995.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, second edition, 2000.

[6] Christian Ensel. Pvmerlin, text and contact editor. http://www.nm.ifi.lmu.de/ ensel/privat/pvmerlin, 2004.

[7] Yuko Eto and Masakazu Suzuki. Mathematical formula recognition using virtual link network. *International Conference on Document Analysis and Recognition*, 0:0762, 2001.

[8] Kam fai Chan and Dit yan Yeung. Mathematical expression recognition: a survey. *International Journal on Document Analysis and Recognition*, 3:3–15, 2000.

[9] Vojtech Franc and Václav Hlavác. License plate character segmentation usint hidden markov chains. In Walter G. Kropatch, Robert Sablatnig, and Allan Handbury, editors, *DAGM 2005: Proceedings of the 27th DAGM Symposium*, volume 1 of *LNCS*, pages 385–392, Berlin, Germany, 2005. Springer-Verlag.

[10] Nicholas J. Higham. *Handbook of writing for the mathematical sciences.* Society for Industrial and Applied Mathematics, 1st edition, 1993.

[11] Donald E. Knuth. Mathematical typography. *j-BAMSN*, 1:337–372, 1979.

[12] Joseph LaViola. Mathpad2. http://www.cs.brown.edu/ jjl/mathpad/, 2003.

[13] Nicholas Matasakis. The natural log system. http://www.ai.mit.edu/projects/natural-log/overview, 2003.

[14] Nicholas Matsakis. Recognition of handwritten mathematical expressions. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1999.

[15] Shunji Mori, Ching Y. Suen, and Kazuhiko Yamamoto. Historical review of ocr research and development. *Document image analysis*, pages 244–273, 1995.

[16] R. Narasimhan. Labeling schemata and synctactic descriptions of pictures. *Information and Control*, 7(2):151–179, 1964.

[17] Loic Pottier. Mathematical formula recognition using graph grammar. In *In Proceedings of the SPIE*, pages 44–52, 1998.

[18] Daniel Průša and Václav Hlaváč. Mathematical formulae recognition using 2d grammars. In *Proceedings of the 9th International Conference on Document Analysis and Recognition*, volume II, pages 849–853, Curitiba, Brazil, 2007.

[19] Daniel Průša and Václav Hlaváč. Structural construction for on-line mathematical formulae recognition. In *CIARP*, pages 317–324, 2008.

[20] Michail I. Schlesinger and Václav Hlaváč. *Ten lectures on statistical and structural pattern recognition*, volume 24 of *Computational Imaging and Vision*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[21] Steve Smithies, Kevin Novins, and James Arvo. A handwriting-based equation editor. In *Graphics Interface*, pages 84–91, 1999.

[22] Masakazu Suzuki. Infty project. http://www.inftyproject.org/en/index.html, 2003.

[23] Masakazu Suzuki, Fumikazu Tamari, Ryoji Fukuda, Seiichi Uchida, and Toshihiro Kanahori. Infty: an integrated ocr system for mathematical documents. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 95–104, New York, NY, USA, 2003. ACM.

[24] Ernesto Tapia. Mathfor. http://mathfor.mi.fu-berlin.de/, 2002.

[25] C. C. Tappert, C. Y. Suen, and T. Wakahara. The state of the art in online handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(8):787–808, 1990.

[26] xThink Corporation. Interactive journal for math. http://www.xthink.com/mathjournal.html, 2003.

[27] Richard Zanibbi, Dorothea Blostein, and James R. Cordy. Recognizing mathematical expressions using tree transformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(11):1455–1467, 2002.

A Appendix: Grammar File

```
Formulae

Formulae->Expr
Formulae->Equality
Formulae->Inequality
Formulae->Nequality

Expr->Term
Expr->SimpleSeqFormulaes
Expr->SeqFormulaes
Expr->BinaryOperation
Expr->Brackets()
Expr->Brackets()sqare
Expr->Brackets{}
Expr->Brackets<>
Expr->Fract
Expr->Power
Expr->Sqrt
Expr->PowerSqrt
Expr->PowerInverse
Expr->PowerDerivative
Expr->Sum
Expr->Prod
Expr->Integral
Expr->Function
Expr->PowerFunction
Expr->BinomialCoeff
Expr->PowerBinomialCoeff
Expr->Factorial
Expr->Limit
Expr->PiExpr
Expr->NegativeExpr
Expr->PositiveExpr
Expr->ComplexExpr
Expr->ComplexExprSimple
Expr->ConjugateExpr
Expr->DoubleExpr
```

```
Expr->ScalarProd
Expr->VectorialProd
Expr->AbsoluteValue
Expr->AbsoluteSqare
Expr->Module
Expr->ModuleSqare
Expr->Optimize

Term->Number
Term->Variable
Term->Subscript
Term->Vector
Term->Etc
Term->Infinity

SimpleSeqFormulaes->SimpleSeq|Formulae
SimpleSeq->Formulae_CommaD

SeqFormulaes->Seq|Formulae
Seq->Formulae_CommaD
Seq->Seq|Seq

Argument->Term
Argument->Brackets
Argument->Fract
Argument->Power
Argument->Sqrt
Argument->PiExpr

BinaryOperation->Expr|OpExpr
OpExpr->BinaryOperator|Expr

Brackets->Brackets()
Brackets->Brackets()sqare
Brackets->Brackets{}
Brackets()->Left(|Br()Expr
Br()Expr->Formulae|Right)
Brackets()sqare->Left(sqare|Br()sqareExpr
Br()sqareExpr->Formulae|Right)sqare
Brackets{}->Left{|Br{}Expr
Br{}Expr->Formulae|Right}
Brackets<>->Left<|Br<>Expr
Br<>Expr->SimpleSeqFormulaes|Right>

Fract->Expr/LowerFract*
LowerFract->Line*/Expr
```

```
Sqrt->SqrtSign%Expr

Power->Term^Expr
Power->Brackets^Expr
PowerInverse->Term^InverseSign
PowerInverse->Brackets^InverseSign
PowerSqrt->Expr|Sqrt
PowerDerivative->Term^DerivativeSign
PowerDerivative->Brackets^DerivativeSign
PowerFunction->Power|Brackets
PowerFunction->PowerFunctionPart|Argument
PowerFunctionPart->NamedFunctionName^Expr
PowerFunction->PowerInverse|Brackets
PowerFunctionPart->NamedFunctionName^InverseSign
PowerFunction->PowerDerivative|Brackets
PowerFunctionPart->NamedFunctionName^DerivativeSign
PowerBinomialCoeff->BinomialCoeff^Expr

NegativeExpr->[-]|Expr
PositiveExpr->[+]|Expr

ConjugateExpr->ConjugateSign/Expr*

ComplexExpr->RealPart|ImaginaryPart
RealPart->Term|BinaryOperator
ImaginaryPart->ComplexExprSimple
ComplexExprSimple->ComplexSign|Expr
ComplexExprSimple->Expr|ComplexSign

DoubleExpr->BinaryOperatorDouble|Expr

BinomialCoeff->Left(|BrBiExpr
BrBiExpr->BiExpr|Right)
BiExpr->Expr/Expr

Factorial->Expr|[!]

Sum->SumExpr|Expr
Sum->SumSimple|Expr
Sum->SumSign|Expr
Sum->SeqSum
SumExpr->Expr/LowerSumPart*
LowerSumPart->SumSign*/Expr
LowerSumPart->SumSign*/Equality
LowerSumPart->SumSign*/Inequality
```

```
LowerSumPart->SumSign*/Equalities&Inequalities
Equalities&Inequalities->Equalities
Equalities&Inequalities->Inequalities
Equalities&Inequalities->Nequalities
Equalities&Inequalities->Equality*/Nequality
Equalities&Inequalities->Nequality*/Equality
Equalities&Inequalities->Nequality*/Inequality
Equalities&Inequalities->Inequality*/Nequality
Equalities&Inequalities->Inequality*/Equality
Equalities&Inequalities->Equality*/Inequality
Equalities&Inequalities->Equality*/Equalities&Inequalities
Equalities&Inequalities->Inequality*/Equalities&Inequalities
Equalities&Inequalities->Nequality*/Equalites&Inequalities
Equalities&Inequalities->Equalites&Inequalities*/Equality
Equalities&Inequalities->Equalities&Inequalities*/Inequality
Equalities&Inequalities->Equalites&Inequalities*/Nequality
Equalities->Equality*/Equality
Equalities->Equalities*/Equality
Nequalities->Nequality*/Nequality
Nequalities->Nequalities*/Nequality
Inequalities->Inequality*/Inequality
Inequalities->Inequalities*/Inequality
Equalities&Inequalities->Equalites&Inequalities*/Equalites&Inequalities
SumSimple->SumSign*/Expr
SumSimple->SumSign*/Inequality
SumSimple->SumSign*/Equalities&Inequalities
SeqSum->SeqSumExpr|Expr
SeqSum->SeqSumSimple|Expr
SeqSum->SeqSumSign|Expr
SeqSumExpr->SeqSumExpr|SumExpr
SeqSumExpr->SumExpr|SumExpr
SeqSumSimple->SeqSumSimple|SumSimple
SeqSumSimple->SumSimple|SumSimple
SeqSumSign->SeqSumSign|SumSign
SeqSumSign->SumSign|SumSign

Prod->ProdExpr|Expr
Prod->ProdSimple|Expr
ProdExpr->Expr/LowerProdPart*
LowerProdPart->ProductSign*/Equality
LowerProdPart->ProductSign*/Inequality
ProdSimple->ProductSign*/Variable
ProdSimple->ProductSign*/Inequality
ProdSimple->ProductSign*/Equalities&Inequalities

Integral->IntPart|IntExpr
```

```
Integral->IntSign|IntExpr
Integral->SeqIntegral
IntPart->Expr/LowerIntPart*
IntPart->IntSign*/Expr
LowerIntPart->IntSign*/Expr
IntExpr->Expr|IntSuffix
IntExpr->IntSuffix
SeqIntegral->SeqIntPart|SeqIntExpr
SeqIntegral->SeqIntSign|SeqIntExpr
SeqIntPart->SeqIntPart|IntPart
SeqIntPart->IntPart|IntPart
SeqIntPart->Expr/LowerSeqIntPart*
LowerSeqIntPart->SeqIntSign*/Expr
SeqIntPart->SeqIntSign*/Expr
SeqIntExpr->SeqIntExpr|IntExpr
SeqIntExpr->IntExpr|IntExpr
SeqIntExpr->Expr|SeqIntSuffix
SeqIntExpr->SeqIntSuffix
SeqIntSuffix->SeqIntSuffix|IntSuffix
SeqIntSuffix->IntSuffix|IntSuffix
SeqIntSign->SeqIntSign|IntSign
SeqIntSign->IntSign|IntSign
SeqIntSign->2IntSign
SeqIntSign->3IntSign
IntSuffix->[d]|Variable

Limit->LimExpr|Expr
LimExpr->LimSign*/TendTo
LimExpr->LimSign
TendTo->Expr|TendToPart
TendToPart->Arrow|Expr

Exp->[e]^Expr
Exp->[e]

Function->Variable|Brackets
Function->Subscript|Brackets
Function->NamedFunction

Equality->Expr|EqualsPart
EqualsPart->Equals|Formulae

Nequality->Expr|NequalsPart
NequalsPart->Nequals|Formulae

Inequality->Expr|InequalsPart
```

```
InequalsPart->Inequals|Formulae
Inequals->GreaterSign
Inequals->LowerSign
Inequals->GreaterOrEqualSign
Inequals->LowerOrEqualSign
Inequals->GreaterSign|GreaterSign
Inequals->LowerSign|LowerSign

Vector->Arrow/Variable*
Vector->Arrow/Subscript*
Vector->UnitaryVector
UnitaryVector->UnitaryVectorSign/Variable*
UnitaryVector->UnitaryVectorSign/Subscript*

ScalarProd->Vector|ScalarProdExpr
ScalarProdexpr->ScalarOperator|Vector
ScalarProd->Brackets<>
VectorialProd->Vector|VectorialProdExpr
VectorialProdExpr->VectorialOperator|Vector
ScalarOperator->[Sop]
VectorialOperator->[Vop]

Subscript->Variable_Number
Subscript->Variable_Variable
Subscript->Variable_DoubleTerm
Subscript->Variable_Subscript
Subscript->Variable_BinaryOperation
Subscript->Variable_Brackets
Subscript->Variable_NegativeSubscript
NegativeSubscript->[-]|Number
NegativeSubscript->[-]|Variable
NegativeSubscript->[-]|DoubleTerm
NegativeSubscript->[-]|Subscript
NegativeSubscript->[-]|Brackets()
DoubleTerm->Variable|Variable
DoubleTerm->Variable|Number
DoubleTerm->Number|Variable

AbsoluteValue->absSign|AbsoluteExpr
AbsoluteExpr->Expr|[pole]
AbsoluteSqare->AbsoluteValue^[2]
absSign->[pole]
Module->modSign|ModuleExpr
ModuleExpr->Expr|modSign
modSign->[pole]|[pole]
ModuleSqare->Module^[2]
```

```
Optimize->minExpr|Expr
minExpr->min*/Expr
minExpr->argmin*/Expr
minExpr->min
minExpr->argmin
Optimize->MaxExpr|Expr
MaxExpr->Max*/Expr
MaxExpr->argmax*/Expr
MaxExpr->Max
MaxExpr->argmax

LogBase->Log_Number
LogBase->Log

NamedFunction->NamedFunctionName|Argument
NamedFunction->Exp
NamedFunctionName->Cos
NamedFunctionName->Sin
NamedFunctionName->Tan
NamedFunctionName->arcos
NamedFunctionName->arsin
NamedFunctionName->artag
NamedFunctionName->Cosh
NamedFunctionName->Cot
NamedFunctionName->Coth
NamedFunctionName->Ln
NamedFunctionName->LogBase
NamedFunctionName->Err
NamedFunctionName->arg
NamedFunctionName->min
NamedFunctionName->Max
NamedFunctionName->argmin
NamedFunctionName->argmax
NamedFunctionName->mod
NamedFunctionName->abs
NamedFunctionName->det

Cos->[c]|Str_os
Str_os->[o]|[s]
Sin->[s]|Str_in
Str_in->[i]|[n]
Tan->[t]|Str_an
Str_an->[a]|[n]
arccos->Str_arc|Cos
arcsin->Str_arc|Sin
```

```
arctan->Str_arc|Tan
Str_arc->[a]|Str_rc
Str_rc->[r]|[c]
Ln->[l]|[n]
Log->[l]|Str_og
Str_og->[o]|[g]
Err->[e]|Str_rr
Str_rr->[r]|[r]
arg->[a]|Str_rg
Str_rg->[r]|[g]
min->[m]|Str_in
Max->[M]|Str_ax
Str_ax->[a]|[x]
argmin->arg|min
argmax->arg|Max
mod->[m]|Str_od
Str_od->[o]|[d]
abs->[a]|Str_bs
Str_rg->[b]|[s]
det->[d]|Str_et
Str_et->[e]|[t]
Lim->[l]|Str_im
Str_im->[i]|[m]

BinaryOperator->[+]
BinaryOperator->[-]
BinaryOperator->[*]
BinaryOperatorDouble->[+]/[-]
BinaryOperatorDouble->[-]/[+]

InverseSign->[-]|[1]
ComplexSign->[i]
ComplexSign->[j]
ConjugateSign->Line
DerivativeSign->CommaU
DerivativeSign->2CommasU
UnitaryVectorSign->[mod1]

Arrow->[->]

Line->[line]
Line->[-]

Equals->[=]
Nequals->[no=]
GreaterSign->[>]
```

```
GreaterOrEqualSign->[>=]
LowerSign->[<]
LowerOrEqualSign->[<=]
SumSign->[sum]
ProductSign->[prod]
IntSign->[int]
SqrtSign->[sqrt]
LimSign->[lim]
LimSign->Lim

Left(->[(]
Right)->[)]
Left(sqare->[(sqare]
Right)sqare->[)sqare]
Left{->[{]
Right}->[}]
Left<->[<]
Right>->[>]

PiExpr->Pi
PiExpr->Number|Pi
PiExpr->Variable|Pi
PiExpr->Pi|Variable
PiExpr->PiExpr|Variable
PiExpr->Pi|Subscript
PiExpr->PiExpr|Subscript
PiExpr->Pi|Brackets
PiExpr->PiExpr|Brackets
Pi->[pi]

Infinity->[infty]

Number->Digit
Number->Digit|Number
Number->DecimalNumber
DecimalNumber->DecimalExpr|Number
DecimalExpr->Number_Dot
DecimalExpr->Number^CommaU

CommaU->[']
2CommasU->["]
CommaD->[,]
Dot->[.]
Etc->Dots|Dot
Dots->Dot|Dot
```

```
Digit->[0]
Digit->[1]
Digit->[2]
Digit->[3]
Digit->[4]
Digit->[5]
Digit->[6]
Digit->[7]
Digit->[8]
Digit->[9]

Variable->[a]
Variable->[b]
Variable->[c]
Variable->[d]
Variable->[e]
Variable->[f]
Variable->[g]
Variable->[h]
Variable->[i]
Variable->[j]
Variable->[k]
Variable->[l]
Variable->[m]
Variable->[n]
Variable->[o]
Variable->[p]
Variable->[q]
Variable->[r]
Variable->[s]
Variable->[t]
Variable->[u]
Variable->[v]
Variable->[w]
Variable->[x]
Variable->[y]
Variable->[z]

Variable->[A]
Variable->[B]
Variable->[D]
Variable->[E]
Variable->[F]
Variable->[G]
Variable->[H]
Variable->[I]
```

```
Variable->[J]
Variable->[K]
Variable->[M]
Variable->[N]
Variable->[P]
Variable->[Q]
Variable->[R]
Variable->[T]
Variable->[U]
Variable->[V]
Variable->[W]
Variable->[X]
Variable->[Y]
```