Recognition of Mathematics Notation via Computer Using Baseline Structure

Richard Zanibbi
zanibbi@cs.queensu.ca

August 2000
External Technical Report
ISSN-0836-0227-
2000-439

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared August 3, 2000

Abstract

The spatial structure of mathematical expressions may be represented using the structure of baselines in an expression. This baseline structure may be represented as a tree, and then rewritten to represent different interpretations of spatial structure in an expression. This structure tree may also be translated into mathematical string languages such as LaTeX, Maple etc. A parsing algorithm which extracts baseline structure in mathematics expressions from a list of attributed symbols is presented, along with an implementation of the algorithm. The implementation of the parsing algorithm has been interfaced with an online mathematics entry system. The resulting integrated system allowed testing of the parser on hundreds of input expressions, with encouraging results. Future work in the extension of the current mathematics recognition technique and application of direction-based recognition to other notations such as music notation and circuit diagrams are also discussed.

Acknowledgments

Thanks to Dorothea Blostein who was a fun, knowledgeable, and supportive supervisor, Ed Lank for all his help with the design of DRACULAE and the ideas in this thesis, Nick Willan for his implementation of the user interface for DRACULAE and patiently listening to me work out many of the ideas in this thesis at the white board (for the record, Nick named the parsing application DRACULA and I simply added an "E"), Ken Whelan for his fantastic proofreading, Gary Anderson, who saved my life on many an occasion and was consistently an amazing source of information useful and entertaining, Hoda Fahmy, whose experience and insight have been very helpful (and who raised some of the semantics issues discussed at the end of this thesis), Jianping Wu, for his help with course work and things C++, Talib Hussein, Laurie Ricker, and all the other students and professors who helped me survive my M.Sc. in various ways.

Thanks to Genarro Costagliola, who provided what for me was a highly influential draft paper on recognizing a subset of mathematics notation using a positional grammar, and some valuable comments.

Thanks to Steve Smithies, Kevin Novins and Jim Arvo for letting me use FFES to develop DRACULAE. Without FFES, it would have been much harder to develop and test DRACULAE.

Thanks to Debby, Linda, Irene, Sandra, Tom Bradshaw and Gary Powley for all the help with the hard, thankless stuff that keeps things going and gets them done.

Finally, thanks to Katey for listening to my rambles about this stuff over and over again, and for being a fabulous proofreader, wife and friend.

Chapter 1

Introduction

In certain domains, the ability to use visual languages to communicate with computers allows simpler and more succinct expression of a user's intentions than is possible with a string language. For example, it is easier for a user to draw a complicated mathematical expression in mathematics notation (a visual language) than to write the same expression in a string language such as LaTeX or Maple. Due to the sometimes more succinct and intuitive expressions available in visual languages in comparison to string languages, disciplines such as music, architecture and engineering commonly use visual languages to convey information in the form of diagrams.

Informally, a visual language is comprised of sets of symbols laid out in two or three dimensions ("diagrams") which comprise the set of legal expressions under a syntactic and semantic definition of the visual language. The principal difference between a visual language and a string language is that the syntax of a visual language is of higher dimensionality. As a result, book-keeping is required for spatial relationships between the terminals and non-terminals of the language when parsing [1].

A visual language may be defined as in the following:

Graphical Primitive: A graphical primitive is a line or dot in an image.

Symbol: A symbol is defined by a non-empty set of spatial relations on graphical primitives which are intended to be perceived as a single unit.

Diagram: A diagram is a two or three dimensional collection (set) of symbols.

Visual Language: A visual language is a set of diagrams which are considered valid expressions in the defined language. Both the syntax and semantics of a visual language are determined through the spatial relationships between symbols in a diagram (based on description in[39]).

Currently there is ongoing research into the design and development of visual programming languages, including compiler-compilers for these languages [11, 19, 20]. The diagrams of a visual programming language may be translated to an executable form using a visual language compiler. Spreadsheets are the most common and perhaps also the most successful type of visual programming language to date[43].

Some visual languages such as math notation, music notation, and engineering drawings are not formally defined, and have dialects (i.e. notational practices which are not standard, but accepted). We call these visual languages *natural visual languages*, and define them as in the following:

Dialect: A dialect of a visual language is a variant of a visual language which employs symbolic and/or spatial conventions which are not standard in all other variants of the visual language.

Natural Visual Language: A natural visual language is a visual language for which no formal description exists and/or dialects of the visual language exist.

Mathematics notation is the natural visual language on which we focus in this thesis. For the remainder of this thesis, when "mathematics notation" is used, it is the Western standard, read left-to-right, which is being indicated (for an interesting comparison, see [25] for a discussion of recognizing Arabic mathematical notation, which is read right-to-left). Mathematics notation is not formally defined: we usually assume that "mathematics notation" includes notations for algebra, calculus, logic, and a number of other mathematical disciplines. Mathematical discourse also often involves the use of defined notation, and symbols often have different semantic interpretations (e.g. f(b) may be interpreted as function application or as implied multiplication, or may be defined notation), producing dialects.

Natural visual languages such as mathematics notation have both hard and soft conventions[6]. Hard conventions are those aspects of a visual language that are consistent across dialects. Soft conventions may or may not apply to a particular diagram depending on the dialect used when a diagram was created. An example of a hard convention in mathematics notation is the left-to-right direction of interpretation. Soft conventions in mathematics notation include layout of symbols.

The lack of formal definition for natural visual languages necessitates identification of hard and soft constraints before a language definition may be constructed. Without formal definitions, it is difficult and perhaps impossible to produce a completely accurate list of the two sets of conventions for any natural visual language, as these have to be obtained largely through observation and introspection. Once the conventions of the language are felt to be reasonably well understood and classified as hard or soft, one may set about defining a syntax specifying legal symbol placement in diagrams, and the semantic interpretation of this set of legal diagrams. Again, the lack of formal definition prevents this language definition from being complete in many cases.

In the case of mathematics notation, some general descriptions of the structure of the notation are available from typesetting procedures [13, 34, 33], a history of the evolution of mathematics notation [10], and a history of mathematics [9]. Cajori's history of notation unfortunately focuses on the evolution of symbols rather than on the use of space between symbols. In addition, the typesetting references describe notation from the perspective of generating mathematics notation, and do not explicitly state the hard and soft conventions of the notation[7].

Dialects complicate syntactic and semantic definition due to the resulting multiple interpretations that are possible. For example, in mathematics notation it is impossible to determine whether "cos" should be interpreted as a function, or implied multiplication of variables without a prior decision about how to interpret this letter grouping; i.e., choosing a soft convention. This is an example of how interpreting visual language dialects requires a priori information about the intended dialect of interpretation before the intended syntax and semantics can be obtained from a diagram. If this information is available, the appropriate soft conventions may be adopted to properly recognize a diagram.

1.1 The Visual Language Recognition Process

Figure 1.1 shows the process of recognizing visual languages by computer, and the four general types of diagram representation usable by a computer (labeled Types I-IV). Each of these diagrammatic representations is explained further below.

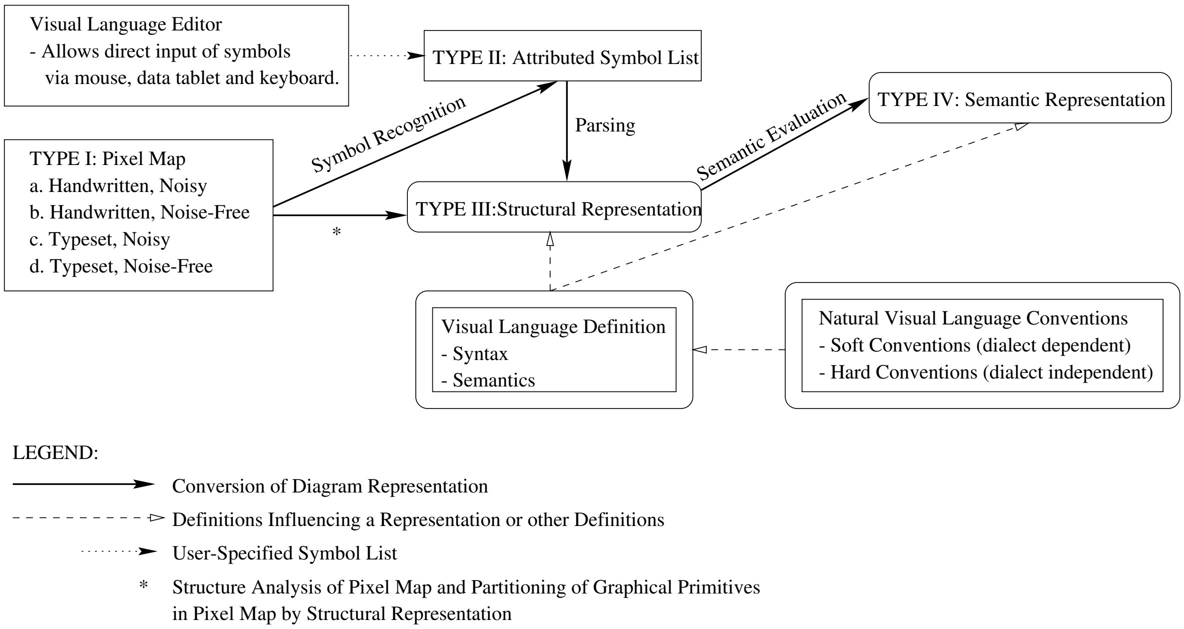


Figure 1.1: The Visual Language Recognition Process

1.1.1 Diagram Representation Type I: Pixel Map

Initial input is obtained either via a pixel map or an online visual language editor. An online editor has the advantage of eliminating symbol ambiguity, as a symbol recognition step is unnecessary. A user may either directly select symbols from a list, or a facility is provided for correction of symbol recognition errors by the user before passing the resulting attributed symbol list for parsing as done in [48].

Two issues concerning the use of pixel map representations are noise and manner of production. Noise in a pixel map is any additional pixel information which is not part of the diagram itself. For instance, pixels in a pixel map added through smudges and/or spurious dots present on scanned hard-copy or added by the scanning process are considered noise. Also, typeset input is more consistent in terms of both layout and symbol composition than handwritten input. Noise-free handwritten and typeset inputs are generally produced on-line: that is, a user creates the pixel map on the computer itself. Two examples of noise-free typeset pixel map are postscript images produced using LaTeX, and pixel maps created using a mouse or data tablet in a drawing application.

1.1.2 Diagram Representation Type II: Attributed Symbol List

An attributed symbol list gives for each symbol a name and set of additional attributes, which may contain coordinates indicating the relative position of the symbol in the diagram. Often this coordinate information is given in the form of bounding box coordinates for two-dimensional notations. A bounding box is the minimal rectangular region which contains all the pixels of a symbol in a digitized image of a diagram.

1.1.3 Conversion from Diagram Representation Type I to Type II

A pixel map may be converted to an attributed symbol list through the use of a symbol recognizer. Generally symbol recognition involves two stages. First, graphical primitives are located in the pixel map. Second, these graphical primitives are partitioned into sets based on their relative positions and then labeled using a process that maps these sets to symbol labels.

Both noise and the manner of production influence the certainty of symbol recognition. Pixel maps from typeset information are more consistent, and thus it is less difficult to recognize symbols than when handwritten input is given. Similarly, noise reduces the certainty of a symbol recognizer in both typeset and handwritten pixel maps. Generally, statistical methods are used to rank certainty of symbol identification and then produce the most statistically likely symbol list (according to the statistical model employed). In some instances, multiple lists of symbols ordered by statistical likeliness are maintained to allow analysis of different symbol sets.

In some cases symbol recognition is performed directly on the pixel map, while in others (such as projection profile cutting[45]) the pixel map is divided into regions before the application of a symbol recognizer, obtaining spatial structure concurrently (this is shown via the '*' labeled arrow). A symbol list may be produced iteratively using feedback, or after a complete partitioning of the pixel map.

1.1.4 Diagram Representation Type III: Structural Representation

A structural representation describes the spatial structure of symbols in a diagram. For example, a LaTeX string is an example of a structural representation of a mathematics expression. Another less explicit example is a parse tree produced by a visual language parser (from which a LaTeX string may have been translated).

An attributed symbol list may be converted to a structural representation through a visual language parser. As mentioned earlier, in some recognition methods a structural representation is obtained directly from a pixel map.

1.1.5 Diagram Representation Type IV: Semantic Representation

A semantic representation of a diagram contains the information content of a diagram. A Maple string is a semantic representation of a mathematical expression diagram. Semantic representations may in some cases be used for evaluation. In a visual programming language this is the execution of the program. In the case of a Maple string, this is the evaluation of the mathematical expression by Maple.

For a given visual language definition, a semantic representation may be obtained from a syntactically valid structural representation.

1.1.6 Definition of the Visual Language

The visual language syntax and semantics are needed for structural and semantic analysis. In the case of natural visual languages, significant work is required to identify and codify the syntax and semantics.

1.2 Thesis and Contributions

The research presented in this document investigates the following thesis.

> Through separating spatial structure from semantics in mathematics notation, a general and flexible recognition of mathematics expressions may be obtained.

By *general* we mean that dialects of mathematics notation may be conveniently handled. *Flexible* refers to the ability to handle a large range of symbol placements.

The following contributions are made in support of this thesis.

1. A model for the structure of mathematics notation

   A novel model, called the *baseline structure tree*, is introduced. *Baseline* in mathematics notation is used to refer to two different things: first, an imaginary line running through symbols which are horizontally adjacent, and secondly the set of horizontally adjacent symbols themselves. Baseline structure trees represent the spatial structure between sets of horizontally adjacent symbols in mathematics notation. This model of baseline structure is consistent with all common dialects of mathematics notation.

2. A visual language parsing algorithm

   This algorithm creates baseline structure trees from attributed symbol lists. The algorithm is more general than existing mathematics notation parsers because the spatial relationships used to obtain baseline structure may be redefined, and tree rewriting may be used to handle soft conventions (i.e. dialects) after an initial baseline structure tree has been obtained.

3. An implementation

   The visual language parsing algorithm was implemented and integrated into a complete recognition system for handwritten mathematics notation. The symbol recognition and user interface components of this system were created by Steve Smithies, Jim Arvo and Kevin Novins [48]. The system has been tested on hundreds of handwritten expressions with excellent results. The system was demonstrated at CASCON

'99 and was enthusiastically received. Public distribution of the system is planned.

Chapter 2

Review of Mathematics Notation Recognition

In this chapter the existing mathematics notation recognition literature is examined. A survey of mathematics recognition research may be found in [7], and a more general survey of diagram recognition in [6].

2.1 General Issues

Blostein notes in [7] that

> any recognition method, including procedurally-coded rules, implicitly or explicitly defines the syntax of recognizable expressions.

Each of the systems described in this chapter thus make some assumptions about the syntactic structure of mathematics notation. Unfortunately, in many cases the reasons for choosing a particular structural definition and/or syntax for mathematics notation are not described in detail.

Martin cites this type of analysis of mathematics notation syntax as a key step in the production of an effective recognition system [40]. More specifically Martin indicates that three things are necessary before a usable mathematics notation recognition system may be created. First, a study of the structure of mathematical notation. Second, the creation of recognition systems which are extendible. Third, a user-friendly means for extending a recognition system.

Towards analyzing the structure of mathematics notation, Martin provides some simple syntactic information on the notation (in the form of spatial relationships) and a number of examples of ambiguous expressions in [40]. He notes that

> Mathematical notation is designed to be unambiguous. However, if the expressions are not carefully written, or certain conventions are not observed, they may appear ambiguous[40].

By conventions Martin is referring to layout of symbols, and the choice of relative size of symbols.

Martin also states that a precedence parser may be built for recognizing mathematics notation if all symbols which indicate vertical displacement from the centre line are leftmost in their subexpressions. In this case, an algorithm may be constructed which scans an attributed symbol list left to right, producing an appropriate structural representation. The following expression with overlapping limits would not be recognized by such a parser as the symbol indicating vertical displacement from the centre line ($\sum$) is not leftmost.

$$\sum_{i=100}^{10000} i^2$$

2.2 Symbol Recognition in Mathematics Notation

The recognition of even typeset mathematical symbols has proven more difficult in the context of existing OCR systems that one might expect. Benjamin Berman and Richard Fateman [5] have done some research in this regard to develop character recognition systems better suited to the math notation recognition context.

Berman and Fateman observed that commercial optical character recognition systems with recognition rates of 99 % or higher were falling to 10% or less once tried on perfectly formed characters in mathematical equation contexts. They indicate that this is because the heuristics employed which work well on straight text, multi-column printing (such as in newspapers) and tables fails with math notation because of the following.

1. Variations in font size
2. Multiple baselines
3. Special characters
4. Different spelling digraph frequencies (i.e. statistical frequency of horizontally adjacent symbols)

2.2.1 Symbol Recognition Techniques Employed

The techniques that have been used to recognize symbols in mathematics notation are more or less standard in the pattern recognition literature. Template matching is used in [15, 5], nearest neighbor classification in [48], neural networks in [23], hidden Markov models in [35], and chain code recognition in [14]. Winkler and Lang [52] discuss a hidden Markov model approach which employs soft-decision making in order to generate multiple interpretations of an input expression. Miller and Viola [42] discuss the resolution of symbol ambiguity using higher level context. Recognition of Arabic mathematical symbols is addressed in [25].

The majority of symbol recognition research in mathematics notation focuses on handwritten input produced online via data tablet or scanned images of typeset equations. A smaller amount of research discusses recognizing typeset pixel maps produced online[42, 28].

2.3 Existing Mathematics Notation Parsers and Structural Analyzers

Existing mathematics recognition systems fall into two main categories. First, there are methods which obtain a structural representation directly from pixel maps, which we call *Structural Analyzers*. In the other category, analysis of structure is performed after symbol recognition has occurred, i.e. these methods obtain a structural representation from an attributed symbol list. We call these systems *Mathematics Notation Parsers*.

Below is a categorization of existing techniques.

1. Structural Analyzers

   (a) Projection Profile Cutting

2. Mathematics Notation Parsers
   (a) Attributed String Grammars
   (b) Structure Specification Schemes
   (c) Graph Grammars
   (d) Stochastic Grammars
   (e) Procedural Translation

We discuss the above methods in greater detail in the following sections.

2.4 Projection Profile Cutting

> Any mathematical expression can be considered as a collection of a number of components (single symbols or subexpressions) arranged horizontally, each of which may contain smaller components arranged vertically[45].

Okamoto et. al. in [45, 44, 49] outline a method of obtaining a structural representation of scanned images of mathematics notation using recursive projection profile cutting. A projection corresponds to projecting pixels onto the x and y axes of the image. The cutting process separates horizontally adjacent subexpressions using a vertical projection, followed by horizontal projection to separate baselines. This process is applied recursively.

Figure 2.1[44] shows an expression and the structural representation obtained after projection profile cutting has occurred. Horizontally adjacent nodes in the tree correspond to pixel regions obtained from a vertical projection, while vertically adjacent nodes in the tree correspond to pixel regions separated by a horizontal projection. Connected pixel regions which cannot be further separated by projection profile cuts are then recognized (shown as symbols in boxes at the leaves).

This approach is not able to detect superscripts, subscripts, matrices, limit expressions (e.g. summations) or expressions within square roots, each of which requires additional processing. These additional processes rewrite the structure tree created by the projection profile cutting process. A TeX string is then translated from the tree.

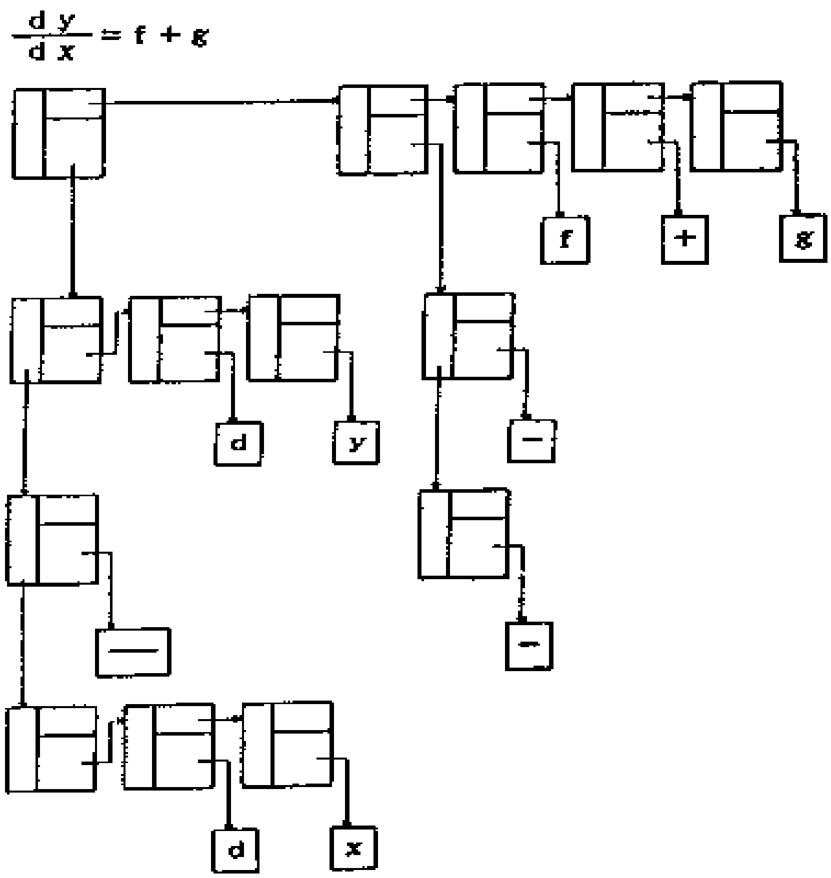


Figure 2.1: Example of Projection Profile Cutting

The general approach taken here is interesting. The following steps are employed in this system.

1. A structural representation is obtained using a structural feature of mathematics notation (i.e. the structure of horizontally and vertically adjacent symbols in a mathematics expression).

2. The structural representation is altered so that it is syntactically valid. In this case a valid syntactic representation is considered to be a tree which may be translated into valid TEX output.

3. The structural representation is mapped to a string (i.e. TEX).

In [49] projection profile cutting is augmented by bottom-up processing. Essentially, character recognition is performed first, and the problematic structures for projection profile cutting are located, analyzed and grouped before projection profile cutting is applied.

This research is closely related to that of Wang and Faure[50, 29]. In this research projections are used in conjunction with a more complicated analysis of spatial relationships between handwritten symbols created online. Separate methods are described for labeling superscripts and subscripts, and a routine for handling square root expressions is mentioned. As in the system by Okamoto et. al., an initial structure is obtained and rewritten to obtain a syntactically valid structure.

The simplicity and speed of projection profile cutting is appealing. However, in addition to difficulty with square roots and sub and superscripts, a strictly projection-based method may improperly segment characters with broken lines, and skew in an input image may alter the necessary horizontal and vertical relationships which are used for later analysis.

2.5 Mathematics Notation Parsers

2.5.1 Attributed String Grammars

Parsing mathematics notation using an attributed string grammar is perhaps the most common type of mathematics recognition system [2, 25, 40, 28, 23, 4, 46, 53, 12]. Generally these systems involve slightly modifying existing parsing methods to obtain a structure representation and/or semantic interpretation from attributed symbol lists[30].

The earliest reported mathematics recognition systems are those of Anderson[1], and Martin [40]. Both Anderson and Martin used systems which they called *coordinate grammars*. A coordinate grammar is essentially an attributed string grammar, with constraints on production application based on the relative positions of symbols. Unlike a string grammar, the order of symbols in the input is unimportant.

Anderson proposes representing symbols through both a bounding box, and a single coordinate intended to approximate the position of the centre line, or *baseline* through horizontally adjacent symbols. Anderson defines the "centre" of a symbol based on where the symbol lies relative to the baseline, as shown in Figure 2.2. In his system, all characters have a horizontal centre coordinate corresponding to the middle horizontal point of the symbol (shown as xcentre for “x” in Figure 2.2).

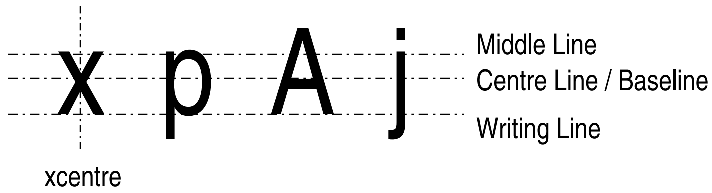


Figure 2.2: Typographic Centre of Symbols

Anderson’s system accepts an attributed symbol list obtained from a data tablet as input, and assumes that symbol recognition errors have been resolved before the input is received. The symbol list contains symbol identity and bounding box coordinates (which define the rectangular region containing all the pixels of a symbol) for each symbol. Consider the following rule from Anderson’s grammar for initially locating a division term. A nonterminal DIVTERM is matched if a horizontal line is found with symbols above and below which do not extend past the horizontal line on either side, or overlap the horizontal line vertically. The parse tree resulting from a match of this rule looks like Figure 2.3.

Anderson’s parser works top-down, attempting all possible partitions of symbols until a valid parse is obtained (some heuristics to improve performance are given in [2]). After the parse tree has been built, the semantics of the expression are obtained by propagating a “meaning” string attribute from the leaves to the root of the parse tree. For example, in the attribute synthesis step after initially building the parse tree, the nonterminal DIVTERM is assigned a ycentre coordinate equivalent to the ycentre coordinate of the horizontal line, and a meaning attribute based on the meaning attributes of S1 and S3.

Anderson’s grammar was possibly the most complete to date, including a separate grammar for handling simple matrices. However, the system was very computationally expensive. Fateman noted that

> Anderson’s parser...handled a very large domain of mathematical expressions. However, at the time of his work (1969), his rec-

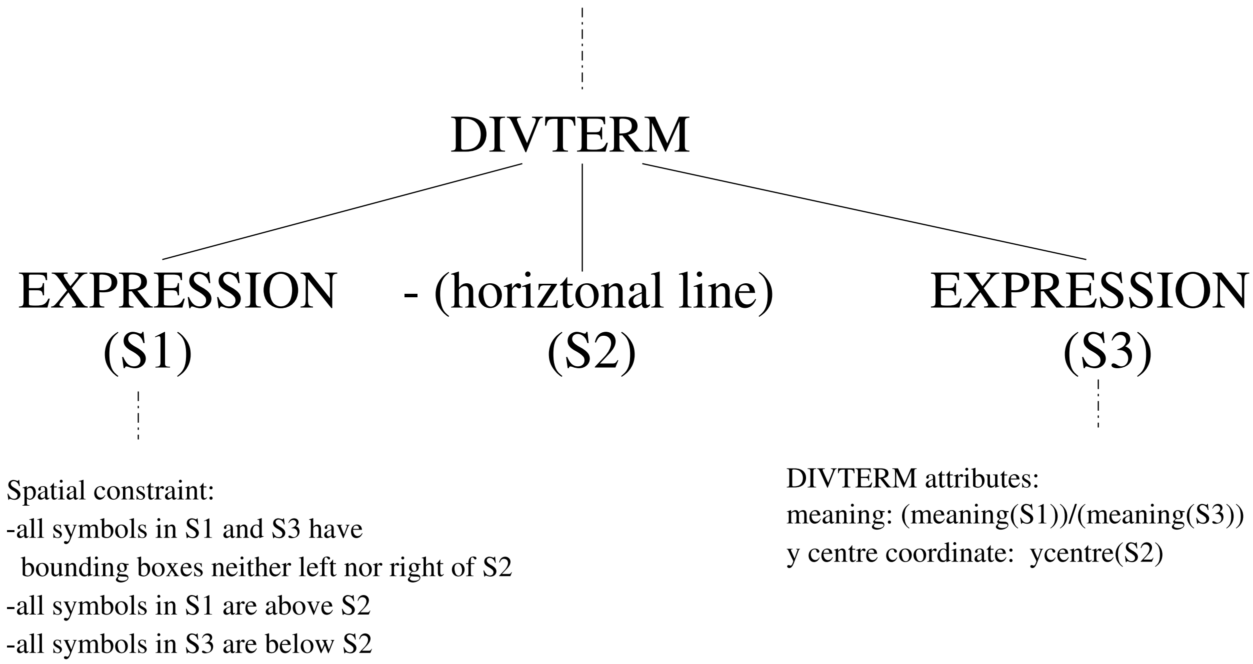


Figure 2.3: Partial Parse Tree for Anderson's Coordinate Grammar

ognizer appears to have been tested to its limits with equations having about eight symbols[28].

Later research in recognition of mathematics expression using attribute grammars tends to focus more on symbol recognition and ambiguity issues than producing a more expressive grammar.

2.5.2 Structure Specification Schemes

Chang in [12] presents another system for obtaining a tree structure representing an expression based on definitions of operator range, precedence, and dominance. Operator range refers to the syntactically valid spatial locations of an operator's arguments. Operator precedence defines the order of application of operators represented as symbols in an expression (e.g. * has greater precedence than +). Operator dominance is defined through a partial ordering of the operators in an input expression based on the operator precedence and range.

A *structure specification scheme* assigns to each operator a way in which an input pattern is partitioned into regions for the operator and its operands. Along with the ordering on operators given by operator dominance, a structure specification scheme may be used to obtain the structure of a mathematics expression.

Input to Chang's method is an attributed symbol list. The symbols in the original input are then recursively partitioned into operator and operand regions using the structure specification scheme, each time partitioning based on the least dominant operator.

The parsing algorithm provided by Chang is fast, being of $O(n^2)$ time complexity. The system described does not appear to handle subscripts or superscripts, as these operators are implicit, represented through relative position of symbols rather than symbols in the input.

2.5.3 Stochastic Grammars

In a stochastic grammar, probabilities are associated with productions of the grammar. Chou presents a stochastic grammar which is capable of recognizing expressions in the context of a significant amount of noise produced by flipping bits in a typeset pixel map [15]. Symbol recognition is performed using exhaustive template matching. A dynamic programming algorithm is then used to find the most likely parse for an input expression. The results are impressive for the given test examples, but the expressions themselves are quite small.

Miller and Viola in [42] discuss their work with a stochastic grammar based on Chou's, with some performance improvements. The inputs they examined were noise-free pixel maps produced online.

2.5.4 Procedural Translation

Lee and Lee[38] describe a set of procedural methods used to translate an attributed symbol list into a formatted string (e.g. in EQN format). Symbols which deviate from the typographical centre of an expression are grouped into units called *symbol groups*, and then the symbol groups and the remaining symbols in the input are ordered left-to-right based on the y-coordinate of their center points.

An output string is obtained by applying this group-order procedure recursively. In a later paper this algorithm is modified so that symbol groups are located recursively first, and a structural representation in the form of a tree is built[37]. Some additional discussion is given about correcting recognition errors through the use of semantic analysis.

A similar approach to Lee and Lee's is described by Winkler who uses a directed acyclic graph representation[51]. Winkler's process has the additional aspect of generating multiple interpretations using a probabilistic model of symbol layout.

The advantage of a procedural recognition method is speed. The major disadvantage of a procedural translation approach is the difficulty in understanding, maintaining and extending such a system (as it is represented solely in terms of procedural code).

2.5.5 Graph Grammars

Bunke in [8] demonstrated how attributed graph rewriting is a useful approach for recognizing schematic diagrams. Graph grammar approaches to diagram recognition have been used by Dori [24] for recognizing dimensions in machine drawings, Fahmy [26, 27] and Baumann [3] for music notation, and Grbavec [32] and Lavirotte and Pottier [36] for mathematics notation.

In graph-grammar based diagram recognition, a graph is initially built from input symbols, containing relation-labeled edges representing spatial structure. This graph is then constrained and/or collapsed using a graph grammar parser which propagates attributes between nodes during production application.

Individual rules in a graph grammar are relatively intuitive, due to the visual representation via graphs. Graph grammars are a very general formalism, but are computationally expensive to parse, as graph matching is NP-complete. Lavirotte and Pottier attempt to alleviate this problem through making their graph grammar deterministic, but this appears difficult to do without restricting the expressivity of the graph grammar. Smithies in [48] proposes using an A* searching algorithm to attempt to improve performance through heuristic means.

2.6 Summary

It is worth noting that in the systems described, trees are used to represent the structure of mathematics expressions, either explicitly, as in the case of projection profile cutting, or implicitly, in the parse trees produced by mathematical notation parsers (other than graph grammar parsers). This pattern of representation influenced the choice of structural representation to be explored in the next chapter.

The desirability of linearizing an attributed symbol list where possible is also a common theme. Where possible, linearization improves performance dramatically (as noted first by Anderson in [1]).

The problem of producing a system which is both efficient and sufficiently flexible to recognize complex spatial relationships remains an open problem. The area would probably benefit greatly if the kind of study of mathematics syntax proposed by Martin[40] were to be undertaken. The current lack of information characterizing the problem domain makes it necessary to expend a considerable amount of effort identifying conventions of mathematics notation before a recognition technique may even be designed.

Chapter 3

A Model for the Structure of Mathematics Notation

> A major objective [for mathematics notation recognition] is to define the syntax cleanly, to provide a unifying framework for handling the myriad details and exceptions that arise during mathematics recognition[7].

Mathematics notation is a natural visual language, and as a result there is no existing formal definition, and dialects exist. However, in order to recognize at least one or more of the dialects of mathematics notation, one needs to produce a language definition.

A study of the hard and soft conventions of the notation may be made through examining conventions observed in the literature, and through introspection and observation. A visual language definition for mathematics notation may then be specified.

Especially in light of the existence of defined notation, the possibility of creating a *complete* visual language definition for mathematics notation is unlikely. However, rather than attempt to produce an "as-complete-as-possible" language definition for a recognition system, it may be more advantageous to identify those notational conventions that seem to be present across dialects (i.e. hard conventions) and then produce a representation which can be easily manipulated for syntactic analysis and semantic interpretation under various language definitions.

In this chapter, using the hard conventions of direction of interpretation and operator dominance, a model of spatial structure in mathematics notation is provided. This is the baseline structure tree (BST), a tree representation which makes baselines and spatial relations between baselines explicit. An example is provided which demonstrates how a baseline structure tree may act as the starting point for a more detailed syntactic analysis under a visual language definition.

3.1 Hard Conventions of Mathematics Notation

As indicated by Okamoto[45], any mathematical expression may be considered as a collection of symbols and subexpressions arranged horizontally. This horizontal adjacency of symbols or subexpressions is referred to informally as a "baseline". Through observation, the direction of interpretation of symbols and/or subexpressions along a baseline is always left-to-right, corresponding to the reading direction in Germanic languages. We propose that this left-to-right direction of interpretation along baselines is a hard convention of mathematics notation.

Anderson observed the usefulness of the directedness of baselines in mathematics notation in 1968[1], indicating that proceeding left-to-right along a baseline may obtain the desired syntactic structure of linear expressions such as:

$$a + b + c/x$$

The linear structure allows the relationship between the above symbols to be represented in a string using only concatenation, unlike other spatial relationships which require a more explicit representation of two-dimensional spatial structure, as will be shown later.

Another hard convention of mathematics notation is the location of the symbol from which interpretation begins in unambiguous expressions. Fateman states that the subset of TEX used for computer algebra systems (e.g. Maple, Mathematica) is strictly left-to-right parseable if some heuristics are employed. Specifically, he states that in a given subexpression,

> the leftmost glyph governs the meaning of the expression. In the few exceptions to this rule, we have tried, by manipulating the expression glyphs to expand the key operator to the left to assert this truth. For example, we consider extending $\int$ by a "virtual" bar extending to its left[28].

In other words, expressions are generally interpreted beginning with their leftmost symbol. Common exceptions include the following:

1. Horizontal lines not being leftmost in their associated subexpression, as in

$$\frac{2}{4}$$

   where the four extends as far to the left as the horizontal line.

2. Limit symbols with overlapping limits. This occurs when symbols in one of the limits extends to the left of the limit symbol, as in

$$\sum_{i=1}^{10000}$$

In these cases, where the symbol from which interpretation begins is not leftmost, operator dominance as defined by Chang[12] may be used to locate the dominant operator in the leftmost subexpression, from where interpretation begins. For example, the dominant operator in the first example above is the horizontal line, and this horizontal line is the symbol from which to begin interpretation. In the case of symbols which are not in the scope of an operator and are leftmost, we simply begin interpretation from this symbol.

Martin in [40] provides a number of ambiguous cases where it is impossible to determine the starting symbol of an expression because either it is impossible to determine operator dominance (e.g. in the case where a fraction such as a/b/c is displayed vertically with equal length horizontal lines) or the range of operators is unclear (such as when a horizontal line representing a division overlaps a symbol slightly, making it unclear whether that symbol is an argument of the division or an adjacent term). Resolving such ambiguities requires a decision on the part of the reader concerning the range of operators and/or the dominant operator in the leftmost subexpression.

In the case then where an expression has an unambiguous operator dominance in the leftmost subexpression, interpretation begins from the dominant operator in that expression, or the leftmost symbol if that symbol alone constitutes the leftmost subexpression. This is a hard convention. The interpretation of ambiguous examples involves soft conventions, as these ambiguities may be resolved using a number of different approaches, none being necessarily correct.

This process of finding the starting symbol of an expression may be represented using the function START. START is specified in the following:

> **START:** START is a function which takes an attributed symbol list representing an unambiguous expression ($list_s$) and returns the starting symbol in $list_s$ or $\emptyset$ if $list_s$ is empty.

3.2 Symbol Layout as a Soft Convention

In mathematics notation, spatial relations are used to group symbols into units, and to specify operators. For instance, horizontal adjacency and the distance between horizontally adjacent symbols (whitespace) are used to group symbols into syntactic units (as in "cos x"). The binary operator for exponentiation is represented using the spatial relation between base and exponent (as in x^2).

Horizontal adjacency, subscripting and superscripting are hard conventions of mathematics notation spatial structure. When perceived as being clearly present by a reader, they are unambiguous spatial relationships between symbols. However, the set of layouts which define each spatial relation is a soft convention; for instance, consider Figure 3.1.

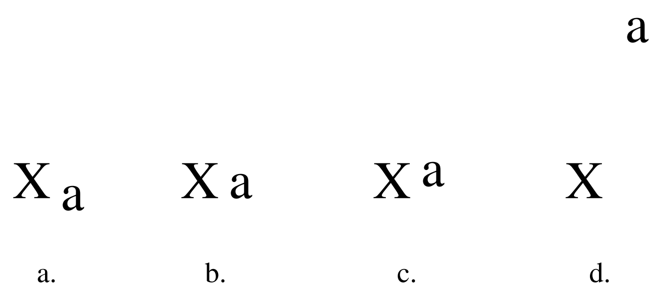


Figure 3.1: Example Spacing Between Adjacent Symbols

It is impossible to state with certainty exactly where the position of the "a" would stop being a subscript, as in Figure 3.1a, and start being horizontally adjacent as in Figure 3.1b. Likewise the same problem occurs when trying to decide where between the positions of "a" in Figure 3.1b and Figure 3.1c the "a" starts being in the superscripted position. Figure 3.1d is itself either syntactically invalid or an exponent, depending on the syntactic definition adopted.

From observation, superscripts, horizontal adjacency, above, below and containing (e.g. by a square root) spatial relationships are hard notational conventions. They are commonly used in the interpretation of mathematics notation. However, given the soft convention of symbol layout, any characterization of spatial relations in mathematics notation will require the adoption of a set of arbitrarily chosen thresholds to define regions for each relation. Thresholds may be defined for what are felt to be ambiguous regions, but this in itself will be a decision on the part of the visual language designer, as what constitutes a "spatially ambiguous" diagram is not formally defined.

3.2.1 Mathematics Notation as a Context-Sensitive Visual Language

Many of the spatial relations employed in mathematics notation may be described through a series of interacting spatial relations between symbols. Consider Figure 3.2.

$$X^{2} \sum_{i=1}^{100000} i$$

Figure 3.2: Exponent and Summation

The reader does not simply look directly above the limit to find the upper limit of the summation; this produces the wrong limit (00000). We can informally describe a process of locating the upper limit in Figure 3.2 in the following.

1. The 2 is closer to the x than the Σ, the 1 (in the upper limit) closer to the Σ

2. Though horizontally adjacent, the 2 is visibly separated from the 1 by whitespace; the 2 is thus superscripted from the x, and the 1 is part of the upper limit of the Σ

3. The zero which ends the upper limit is much closer to the Σ than the next symbol which is horizontally adjacent to the Σ (i), and so is part of the upper limit.

4. 0000 is directly above the Σ

5. Concatenating the 1 and 0 we have found to be part of the upper limit with the 0000 above the Σ, we obtain an upper limit of 100000

The upper limit of the summation in Figure 3.2 may be obtained through the process above or another similar process involving the examination of the relative positions of symbols in the expression.

The kind of complex spatial interaction present in Figure 3.2 is common in expressions with multiple baselines: obtaining the spatial relationship between two symbols may require knowledge of the relationship between those symbols to other symbols in their associated expression. This demonstrates how mathematics notation is a context-sensitive visual language.

3.2.2 Spatial Relations in Mathematics Notation

In this section a number of binary relations are informally defined in order to describe spatial structure in mathematics notation. These definitions are based in part on those discussed by Genarro Costagliola[22, 18, 17, 11, 19, 20]. The specifications are in Table 3.1. For each, S_1 is a single symbol.

We discuss the spatial relations in greater detail in the following.

- HOR indicates horizontal adjacency between two symbols on a baseline. For example, in Figure 3.1b, HOR(X,a) is a valid relation.

- ABOVE and BELOW correspond, intuitively, to baseline symbol sets which are above or below a symbol, but directly, i.e. only those symbols whose center horizontally overlaps the width of the domain symbol.

- CONTAINS indicates square root subexpressions. For example, $\sqrt{x}$ could be represented as CONTAINS($\sqrt{}$,x).

- BLEFT and TLEFT represent spatial relations between a limit symbol (such as $\int$) and symbols in limits which extend to the left of the limit symbol, when the limit symbol starts a baseline. For instance, in

$$\sum_{i=10000}^{1000000} i$$

Spatial Relation	Definition
HOR(S_1, S_2)	S_2 is the next symbol horizontally adjacent to S_1
SUPER(S_1, S_2)	S_2 is the set of symbols superscripted from S_1
SUBSC(S_1, S_2)	S_2 is the set of symbols subscripted from S_1
ABOVE(S_1, S_2)	S_2 is the set of symbols directly above S_1
BELOW(S_1, S_2)	S_2 is the set of symbols directly below S_1
CONTAINS(S_1, S_2)	S_2 is the set of symbols contained by the bounding box of S_1
TLEFT(S_1, S_2)	S_2 is the set of symbols to the top left of a limit symbol which starts a baseline (S_1)
BLEFT(S_1, S_2)	S_2 is the set of symbols to the bottom left of a limit symbol which starts a baseline (S_1)

Table 3.1: Spatial Relations

> The i and = of the lower limit and the 1 and 0 beginning the upper limit are in BLEFT and TLEFT relations with the limit symbol, i.e. TLEFT($\sum$,{1,0}) and BLEFT($\sum$,{i,=}).

In the next chapter each of these relations are defined in greater detail using coordinates of symbols. For now these general descriptions will suffice for explanation.

The previously defined spatial relations, along with the hard conventions of starting symbol and direction of interpretation along baselines may be used to create a structural representation for mathematics notation, the baseline structure tree.

We need first to define *baseline symbol set* and *main baseline symbol set*, as in the following:

Baseline Symbol Set A baseline symbol set is a set of attributed symbols for which the relation HOR holds between all s_i, s_{i+1} for $1 \leq i \leq n-1$ where n is the number of symbols in the set.

Main Baseline Symbol Set Given a list of symbols $list_s$, the main baseline symbol set is the baseline $bline_{main}$ associated with the symbol in $list_s$ from which interpretation begins (i.e. a non-empty set symbol returned by START($list_s$)).

Using the spatial relations discussed along with the idea of baseline symbol set, the structure of a large number of mathematics notation diagrams may be characterized using a tree and parenthesized strings translated from the tree.

Consider the expression

$$\sum_{i=1}^{100000} i^2 + 27i + 2$$

The spatial structure of the symbols in this expression can be represented via a baseline expression tree as given in Figure 3.3. For the purposes of the following examples, assume that the specified spatial relations are valid.

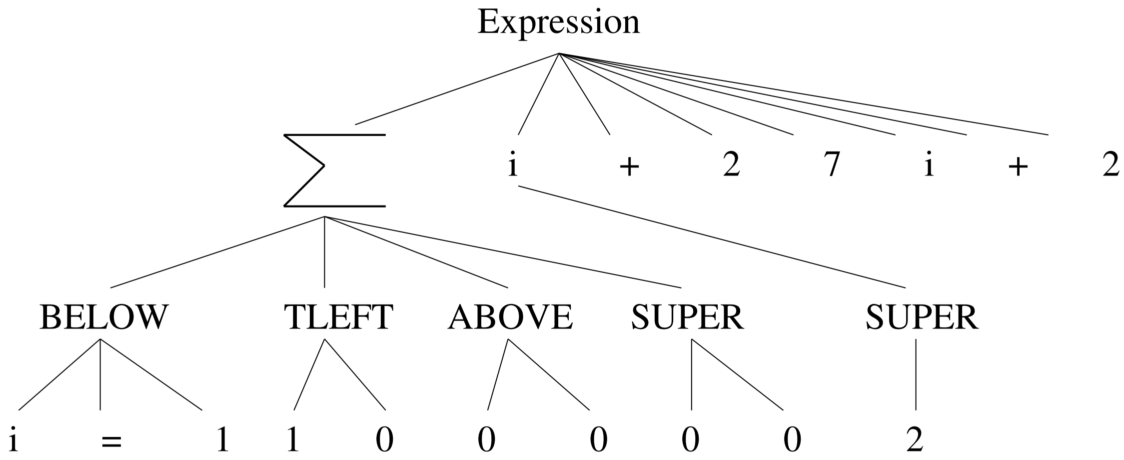


Figure 3.3: A Baseline Structure Tree

The root is labeled “Expression”, and along with the nodes with spatial relation labels, has children ordered left to right (i.e. a baseline symbol set is represented as children of the root and each relation). This eliminates the need in the tree to show HOR explicitly, as any two adjacent siblings below the root or a relation are in a HOR relationship. Symbols, such as the i in i^2 and $\sum$ may have spatial relation-labeled children indicating a set of symbols in a region relative to the symbol.

The main baseline of the expression is represented below the root, while the main baseline of each subexpression appears under an associated relation node.

3.2.3 Properties of Baseline Structure Trees

There are a number of advantages to structural representation via a baseline structure tree. These include:

1. The grouping and left to right order of baseline symbol sets is explicit.
2. All spatial relations are explicit.
3. A depth-first traversal of this tree will produce a linearized string representing the structure if bracketing is used. For instance, the linearization via depth first traversal of Figure 3.3 gives:

   Expression $\sum$ BELOW {i=1} TLEFT {10} ABOVE {00} SUPER {00} i SUPER{2}+27i+2

   If the root ("Expression") is eliminated, the above string possesses the type of syntactic structure used in LaTeX strings.
4. The tree may be restructured in order to represent more complicated structures via labeled relation nodes.

As a demonstration of restructuring a baseline structure tree, consider Figure 3.4. Here a tree-rewriting rule has been applied to Figure 3.3. This rule can be stated as: starting from the root, collect all TLEFT, ABOVE and SUPER nodes associated with a Sigma and replace them with a single relation node labeled "ULIMIT", placing all children of the TLEFT, ABOVE and SUPER nodes under ULIMIT. Then do the same for BLEFT, BELOW and SUBSC. The tree now has a structure which contains a syntactically valid representation of the limits of a summation. This is a very simple rule; in the next chapter a more complicated rewrite is described which allows almost arbitrary subexpressions as limits.

The original expression represented in Figures 3.3 and 3.4 is ambiguous; it is not clear what the scope of the summation is (i.e. does it end with i^2, with 27i or does it encompass all subexpressions adjacent to the $\sum$?).

This may be resolved through the use of a soft convention. For instance, specifying either a certain distance of white space indicating membership in the associated subexpression of the $\sum$, or restricting scope to the first subexpression in the summation's scope if bracketing is not used.

Once having chosen a soft convention, the convention may be represented in the tree using another tree rewrite. For instance, suppose a syntax definition where only the first unbracketed term is considered to be in the scope of

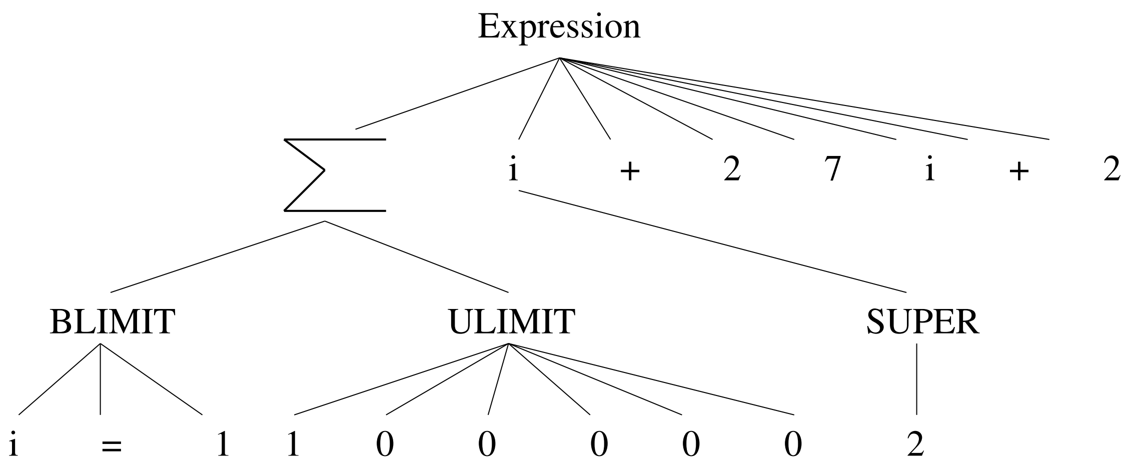


Figure 3.4: Baseline Structure Tree with Rewritten Limits

the summation is used. Another type of relation node may be added to the tree, which we will call SCOPE. The main baseline may then be scanned for symbols up to an operator, and then a rewrite may be applied to produce Figure 3.5.

In this thesis only a small number of rewrites are defined, and only for relatively "hard" notational conventions. A design for a more complete system, capable of handling many dialects is presented in the last chapter.

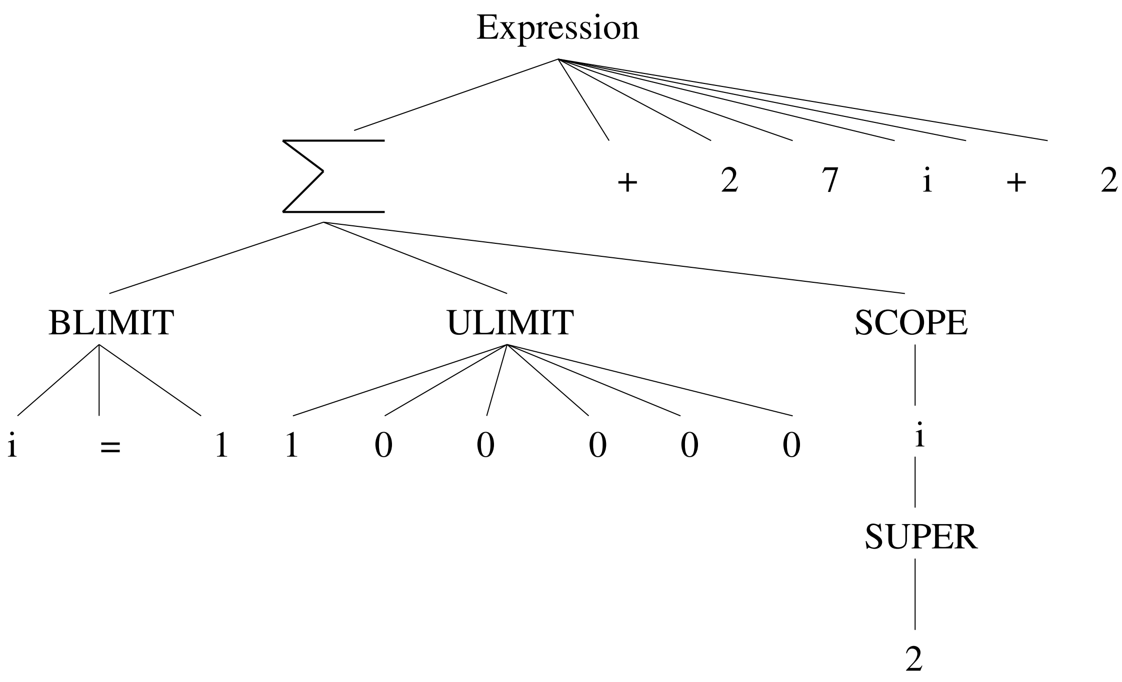


Figure 3.5: Baseline Structure Tree with Rewritten Summation Scope

Chapter 4

A Mathematics Notation Parser

As discussed in the last chapter, a baseline structure tree is a flexible structural representation of a mathematics notation diagram. In this chapter a parsing algorithm for obtaining baseline structure trees from an attributed symbol list is presented. Examples of syntactic analysis performed through tree rewriting, and context-sensitive translation to string languages are also discussed, along with relevant issues.

4.1 Preprocessing and Symbol Attributes

For the purposes of this algorithm, symbols in the input list must have identity and bounding box attributes. A bounding box is defined as a pair of coordinates used to specify the minimum and maximum (x,y) coordinates between which all the pixels of the symbol are located in the positive Cartesian plane. An example is shown in Figure 4.1.

As a preprocessing step, each symbol is given additional attributes, calculated using the identity and bounding box attributes. These additional attributes are type, centroid class, centroid coordinate, and "wall" attributes, which specify a partitioned region in the input.

The type attribute is used to group structurally similar symbols, for instance limit symbols ($\sum, \int, \Pi, \bigcup, \bigcap$), open brackets ({, (, [) and close brackets (},],)). Type attributes simplify processing based on structural function.

Each symbol is assigned a centroid attribute which is used to represent

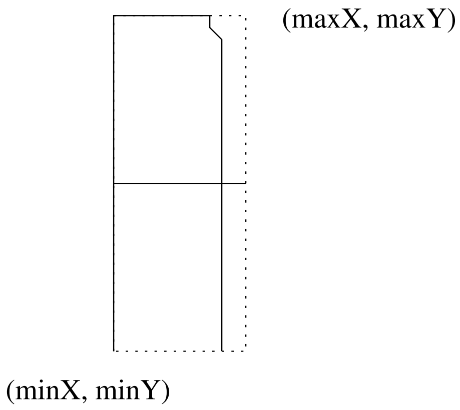


Figure 4.1: A Bounding Box

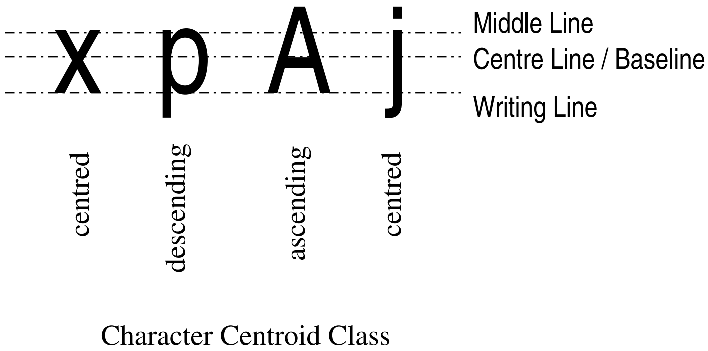


Figure 4.2: Centroid Class for Different Letters

the position of each symbol using a single coordinate. The value of this coordinate is calculated by examining the normal position of the bounding box of the symbol in relation to a baseline/centre line.

Figure 4.2 demonstrates different centroid classes for a number of letters. Ascending characters extend above the middle line, descending characters fall below the writing line, and what we term "centred" characters either fall between the writing line and middle line or extend past both writing line and middle line (such as the "j" in Figure 4.2).

Table 4.1 is based closely on Grabavec's list of centroid classes given in [31]. It shows the centroid classes for a number of characters used in mathematics notation. Symbols which do not appear in this list are in the centred centroid class.

Characters	Alignment
0..9	Ascending
A..Z, $\Gamma, \Delta, \Theta, \Lambda, \Xi, \Phi, \Omega$	Ascending
b,d,f,h,i,k,l,t	Ascending
g,p,q,y	Descending
a,c,e,j,m,n,o,r,s,u,v,w,x,z	Centred
δ, θ, λ	Ascending
$\gamma, \eta, \mu, \rho, \chi, \psi$	Descending
$\alpha, \beta, \epsilon, \zeta, \kappa, \nu, \xi, o, \pi\sigma, \tau, \upsilon, \phi, \omega$	Centred
Limit Symbols	Centred

Table 4.1: Centroid Classes for Symbols

With the exception of brackets, the x coordinate of the centroid is always the middle horizontal point, i.e. minX + ((maxX - minX)/2). In the case of open brackets, the minX coordinate is assigned to the x coordinate of the centroid, and the maxX coordinate is assigned to the x coordinate for close brackets.

The y coordinate of the centroid is then assigned using the centroid class. Ascending characters are assigned a y centroid coordinate at minY + (1/4)(maxY-minY). Descending characters are assigned a y centroid coordinate at minY + (3/4)(maxY - minY), and centred class characters are assigned their y-centre, i.e. minY + ((maxY-minY)/2).

These y-coordinate assignments are intended to reflect the approximate location of the baseline through typeset characters. This corresponds to a (rough) normalization of the input symbol. In particular, with handwritten input this may be a very crude approximation, essentially replacing the contents of the bounding box by a single point based on a model of typeset characters.

Finally, each symbol possesses four "wall" attributes (top, bottom, left and right) which specify a region in the input pattern (specified by (bottom,left) and (top,right)). Initially the bottom and left walls of all symbols are set to -1, and top and right to an arbitrarily large positive integer (e.g. ∞). These wall attributes will be used in the parsing algorithm to partition the input when examining spatial relationships between symbols.

4.2 Syntax Directed Scanning

In this section we discuss syntax directed scanning, which is a key component of the parsing algorithm described in the following sections.

In [17] Genarro Costagliola and Shi-Kuo Chang describe how *syntax directed scanning of the input* can allow conventional LR parser generators such as YACC to be used for building parsers for a class of visual languages (including a simple mathematic language) which may be described using what they term *positional grammars*. The ability to build a *positional LR parser* requires the use of syntax directed scanning of the input.

In this technique, functions corresponding to the spatial relations between symbols are used to describe the syntax of a visual language. These functions are then used to drive the scanning of the input pattern during a parse. Normally in an LR parser linear retrieval is employed, where the next unexamined symbol in the input array is returned. The spatial functions order the input pattern, resulting in a very efficient (i.e. $O(n^2)$) parser for a restricted, but useful, class of visual languages.

The input to a positional LR parser is assumed to be a list of attributed symbols (i.e. containing centroid coordinates) on which the spatial functions may operate. Each spatial function takes the index of a symbol in the input array and returns the index of a token matching the associated relation, or indicates that no symbol matching the spatial relation was found. A symbol matched by a spatial function during a parse is marked in the input, allowing that symbol to be removed from consideration during later analysis.

In Appendix A a positional grammar is given which corresponds to the syntax of baseline structure trees. However, the creation of a generalized LR parser as described by Costagliola[22, 18, 17, 11, 19, 21, 20] is not possible (see Appendix A for further discussion).

4.3 Spatial Functions

In this section the spatial relations introduced in Chapter 3 are defined as a set of functions. For the remainder of this discussion the term *input list* will refer to a pre-processed list of attributed symbols (i.e. with type, centroid

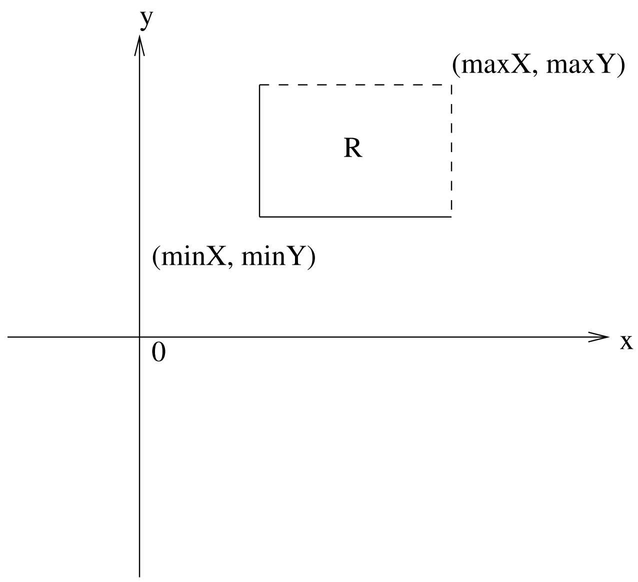


Figure 4.3: An Example Region

class, centroid and wall attributes). An input list is assumed to have been sorted by left-most bounding box x-coordinate, in ascending order.

A region R is defined as an axis parallel rectangular area in the positive integer Cartesian plane defined by a pair of (x,y) coordinates. A coordinate C is said to be in a region R if it lies in the region defined by R, *and* C does not lie on the maximum Y or maximum X coordinate boundaries. Figure 4.3 demonstrates this. The maximum X and Y boundaries are shown with dotted lines to indicate that a point C is not considered within region R on those boundaries.

Horizontal overlap is defined as in the following:

Horizontal Overlap A symbol S_1 is *horizontally overlapped* by a symbol S_2 when the minX bounding box coordinate of S_2 is less than or equal to the centroid x coordinate of S_1.

4.3.1 START, OVERLAP and SP Functions

START is a function which returns the starting symbol for a partitioned region of an input list. The algorithm for START provided in Appendix

B contains a simplified analysis of operator dominance (i.e. it returns the leftmost limit symbol if one is present, rather than analyzing the range of limit symbols), but for the purposes of this research was found to be adequate for a large number of test cases.

START calls a function OVERLAP before returning a symbol. OVERLAP is a function which determines whether a symbol is overlapped by a horizontal line (this is part of the operator dominance analysis). OVERLAP takes an integer, a pair of y-coordinates and an input list. The function then returns either the passed integer if the symbol at that index is not overlapped by a horizontal line, or the index of the largest overlapping line (see Appendix B for the OVERLAP algorithm).

START scans the input to locate the leftmost symbol in R. If no symbol is found, -1 is returned. If a symbol is found, the input is scanned further, looking for the leftmost limit symbol. If no limit symbol is found or is the same as the leftmost symbol, the leftmost symbol is checked for overlap with horizontal lines and then the result is returned. If a limit symbol is found, the scan reverses, checking all symbols to see if they are horizontally adjacent to the limit symbol. If no such symbol is found, the leftmost symbol is part of a limit and the limit symbol is checked for overlap, and the result of overlap check is returned. Otherwise the leftmost symbol is not part of a limit, and is checked for overlap. The result of the overlap examination is then returned.

Another function SP may be defined, corresponding to the type of start symbol function for visual languages described in [22]. SP gives the start symbol of the entire expression, using R defined as (0,0), (∞,∞). Assuming $list_{in}$ is non-empty, SP($list_{in}$) returns the first symbol of the main baseline symbol set of the entire expression (i.e. in the region defined above).

4.3.2 HOR Function

The extension of baselines can be summarized using the following spatial function definition of HOR. HOR takes a symbol and an input list $list_{in}$, and returns either a symbol of $list_{in}$ or $\emptyset$.

Given an input list $list_{in}$ and a symbol s, HOR($list_{in}$,s) $= a$ where

1. a is $\emptyset$ if no unmarked symbol is horizontally adjacent to s in the region defined by s's wall attributes.

2. a is an unmarked symbol in the region defined by s's wall attributes

which is the next horizontally adjacent symbol on the baseline symbol set of which s is a member.

3. If a horizontally overlaps a horizontal line *hline*, let a be the longest horizontal line which overlaps *hline*, or *hline* if no horizontal line overlaps *hline*

Horizontal adjacency is defined using the wall attributes of s (i.e. a region) and the centroid coordinates of a. HOR produces a defined result for HOR(s) for each of the cases in Table 4.2.

The algorithm for HOR is very simple. If s is a horizontal line or bracket, START is called on the region defined by the wall attributes of s and the maxX bounding box coordinate of s (i.e. replacing leftWall). The remaining cases simply require a scan of the input list, performing tests on coordinate positions of s and the identity and coordinates of each symbol encountered. The scan stops when a symbol meeting the criteria for HOR is met. A call is then made to OVERLAP (again, to check with overlap with horizontal lines).

As established in the last section, START is $O(n)$ in the worst case. For the other HOR cases, a scan of the input with $O(1)$ comparisons for each element ($O(n)$) is followed by a call to OVERLAP ($O(n)$). Thus HOR is of $O(n)$ time complexity in the worst case. Due to the input list being sorted, in many cases only a single set of comparisons is made (i.e. $O(1)$ best case).

Figure 4.4 shows the HOR region searched in the general case. In this research the convention of assigning the Upper Threshold to 5/6 of the bounding box height, and the Lower Threshold to 1/6 the height of the bounding box was used, as in [45] and others. TLEFT and BLEFT are bracketed as they are only searched if the "A" was a limit symbol. TLEFT and BLEFT regions are only examined when a limit symbol starts a baseline, or follows a bracket or horizontal line on a baseline.

The special cases for binary operator and calling START reflect the type of deviations from the centre line that occur and which are still perceived as adjacent. In the example in Table 4.2, if the addition sign moves up or down in the binary operator case, the baseline structure is still clear.

In the general case, the leftmost symbol on the right of s in the HOR region is returned as a if it is present and not overlapped by a horizontal line. Note that in this definition of region, symbols may have the same minX coordinate and be processed as being horizontally adjacent, in the order in $list_{in}$.

s	a	EXAMPLE
Binary Operator (not Horizontal Line)	Horizontal Line to right of binary operator in proper region	$+\frac{1}{2}$
Any Symbol	Next leftmost symbol in region which has a bounding box that extends both above and below that of the domain symbol	$x\int$
Horizontal Line Open Bracket	START((wall attributes, let leftWall = maxX of s),$list_{in}$)	$(\sum_{-\infty}^{+\infty}$
General Case	Leftmost symbol with center in HOR region	$\int$ x

Table 4.2: Horizontal Adjacency of Symbols Under HOR

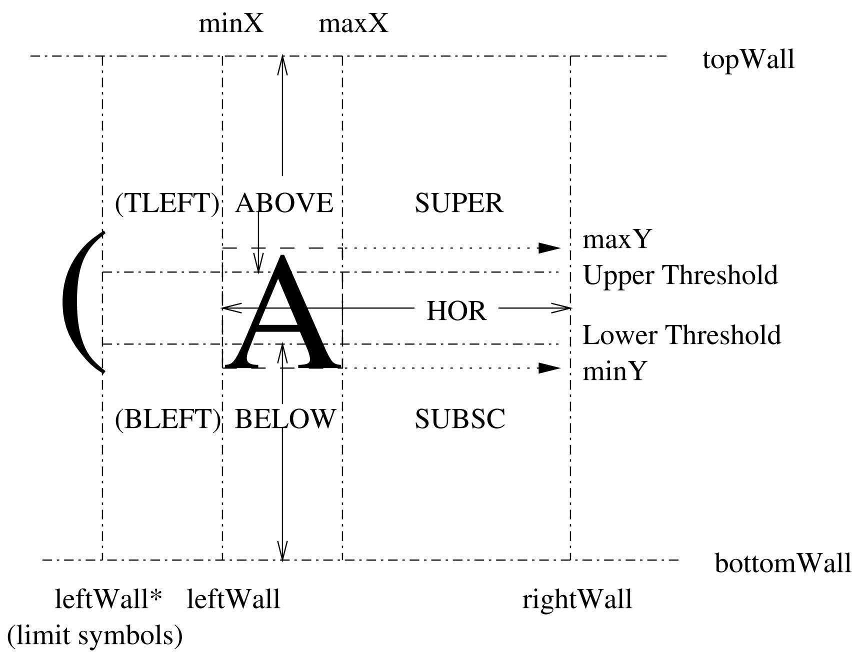


Figure 4.4: General Regions for Spatial Functions

For horizontal lines and open brackets, START is applied to the region defined by the maxX bounding box coordinate of the symbol, and the remaining three wall attributes.

These definitions of HOR, START and the remaining spatial regions clearly adopt some soft conventions of spacing (e.g. a definition of regions, overlap and a series of thresholds). The thresholds adopted are designed to be as flexible as possible, allowing for the widely separated symbols such as those in Figure 3.1d to be recognized as spatially related (in this particular case, as a superscript).

4.3.3 Secondary Baseline Symbol Sets

A given baseline symbol set can be divided in several ways, i.e. by the occurance of other baselines in the regions shown in Figure 4.4. We call these *secondary* baseline symbol sets, as their position is obtained relative to the main baseline symbol set in a given region. The regions pertinent to secondary baseline symbol sets may be examined using the following method.

- Let B be a baseline symbol set in region R ((RminX,RminY),(RmaxX,RmaxY))
- For i=1..n-1, where n is the number of symbols in B ($s_1 \dots s_n$ are the symbols in B)
  - set the rightWall attribute of s_i to the minX coordinate of s_{i+1}
  - set the leftWall attribute of s_i to the symbols' minX coordinate unless this is a limit symbol. If the limit symbol follows an open bracket or horizontal line, assign maxX of s_{i-1} to leftWall. If the limit symbol starts a baseline, assign RminX to leftWall. Otherwise set leftWall to the minX coordinate of the limit symbol.
  - set the topWall attribute of s_i to RmaxY
  - set the bottomWall attribute of s_i to RminY
- In any order, examine the following two disjoint regions for each of the symbols in B (see Figure 4.4)
  1. ABOVE: {(minX,Upper Threshold),(maxX,topWall)}
  2. BELOW: {(minX,bottomWall),(maxX,Lower Threshold)}

Additionally, the following (also disjoint) regions may be examined:

1. SUPER: {(maxX,Upper Threshold),(rightWall,topWall)}
2. SUBSC: {(maxX,Lower Threshold),(rightWall,bottomWall)}
3. TLEFT: {(leftWall,Upper Threshold),(minX,topWall)}
4. BLEFT: {(leftWall,bottomWall),(minX,Lower Threshold)}
5. CONTAINS: {(minX,minY),(maxX,maxY)}

For each region R above START(R,$list_{in}$) is applied, which returns the starting symbol of the main baseline symbol set in that region.

Symbol identity of the symbols in B determines which regions are examined:

- TLEFT and BLEFT are examined only for limit symbols which either start a baseline (first symbol in B), or directly follows a horizontal line or an open bracket (both of which indicate the end of a subexpression).
- CONTAINS is used only for square roots.
- SUPER is not used for horizontal lines or open brackets.
- SUBSC is not used for horizontal lines or open brackets.

4.4 A Mathematics Notation Parsing Algorithm

In Appendix B an O(n^2) parsing algorithm is given which extracts a baseline structure tree from an input list. Essentially the algorithm recursively locates symbols which start a baseline, extracts the associated baselines, and then records the observed baseline structure in a baseline structure tree. In this section we discuss the algorithm in only a very general way. This discussion is simplified; symbols are described as being pushed on and off the stack when it is really the index to the symbol and an associated tree node that is pushed onto either data structure, and a number of other details are ignored. Please consult the appendix for the more detailed description.

While locating baselines (essentially using START in different regions), symbols which start a baseline are pushed on a queue. In the extraction stage, symbols are removed from the queue one at a time. After removing a symbol from the queue, it is pushed on a stack, and the function HOR is then used in a loop to locate the associated baseline symbols, which are then also pushed on the stack. When HOR returns $\emptyset$ (e.g. the end of a baseline is found) "EOBL" is pushed on the stack to act as a separator. The next symbol in the queue is then removed and the same process repeated until the queue is empty.

When the queue is empty, the algorithm reverts back to locating baselines, by using START on all the appropriate regions for secondary baselines relative to the symbols on the stack. Any symbols which start a secondary baseline are placed in the queue. The algorithm stops when either all symbols have been added to the baseline structure tree, or no new baselines are found (in which case an error is indicated). The baseline structure tree is then returned.

Some alterations may be made to the algorithm. For instance, the algorithm has been constructed so that the tree is built one level at a time, though this is not necessary. An algorithm using a single stack which builds the baseline structure tree depth-first may also be created. The paired data structures of a stack and a queue are remnants of earlier research, before START was used recursively.

4.5 Tree Rewriting

Graph rewriting is a powerful and flexible formalism. As discussed in Chapter 2, it has been used for parsing many different types of visual languages. Tree rewriting, a subset of graph rewriting, has the advantage of being tractable. Tree rewriting has proven a very powerful technique for translating programming languages to different versions/dialects[16].

The context present in a baseline structure tree may be used to manipulate the tree in order to represent higher order-spatial relations directly, and/or subtrees may be regrouped and re-parsed. In this way, the initial baseline structure tree produced by the algorithm above may be used for structural analysis using different visual language syntax definitions, i.e. dialects. This is why the baseline structure tree represents a meta-syntax; the original structure is a subset of the syntax of existing dialects (of which we are aware).

As an example, recall Figure 3.3. In this instance the baseline structure tree, while valid in terms of representing the defined spatial relations, was not descriptive enough to represent the common syntactic structure present in the context of a summation.

The simple rule provided in Chapter 3 for rewriting Figure 3.3 as Figure 3.4 is adequate for limits comprised of a single baseline. This rule represents a higher-order spatial relation (i.e. the new "ULIMIT" relation is defined in terms of binary spatial relations in the tree).

However, it may be possible for fractions and/or other limits to be present. To address this new possibility, one can simply remove and then concatenate the three regions SUPER, ABOVE and TLEFT found as children of a limit symbol. Place the concatenated regions in a separate input to the Baseline Structure Tree parsing algorithm. Then place the resulting baseline structure tree and place the result as a child of the limit symbol with the relation label "ULIMIT".

As a last example, consider Figure 4.5. The initial baseline structure tree (Figure 4.5b)r shows the "1" from "10000" as part of the superscript region of the "x". This is because the $\sum$ does not start the main baseline. However, the superscript region off of the "x" may be divided into two regions partitioned by the rightmost symbol which has a centroid that overlaps the $\sum$ horizontally; in this case the "j". All symbols to the left of and including the partition symbol remain in the SUPER region; the remaining symbols are placed in a new region labeled ULIMIT. The symbols in the SUPER and ABOVE regions relative to the $\sum$ are also placed in ULIMIT, and the subtrees rooted at SUPER and ABOVE removed. The symbols in the ULIMIT and SUPER regions (SUPER relative to the "x") are then re-parsed, producing the tree in Figure 4.5c.

It is worth pointing out that using whitespace analysis may have simplified this problem greatly. The algorithm described in this thesis uses neither white space or point-size information. The expression in Figure 3.2 would require this type of analysis, as the last tree rewrite provided would not alter the baseline structure tree, leaving part of the limit (the 1) in the SUPER region of the x.

The visual language syntax defined by the parsing algorithm and the above rewrite rules is very simple. However, it is descriptive enough to obtain the structure of a large number of expressions, as demonstrated in the next chapter. Future work stemming from more complex syntax definitions is discussed in the last chapter.

4.6 Translation to Output Strings

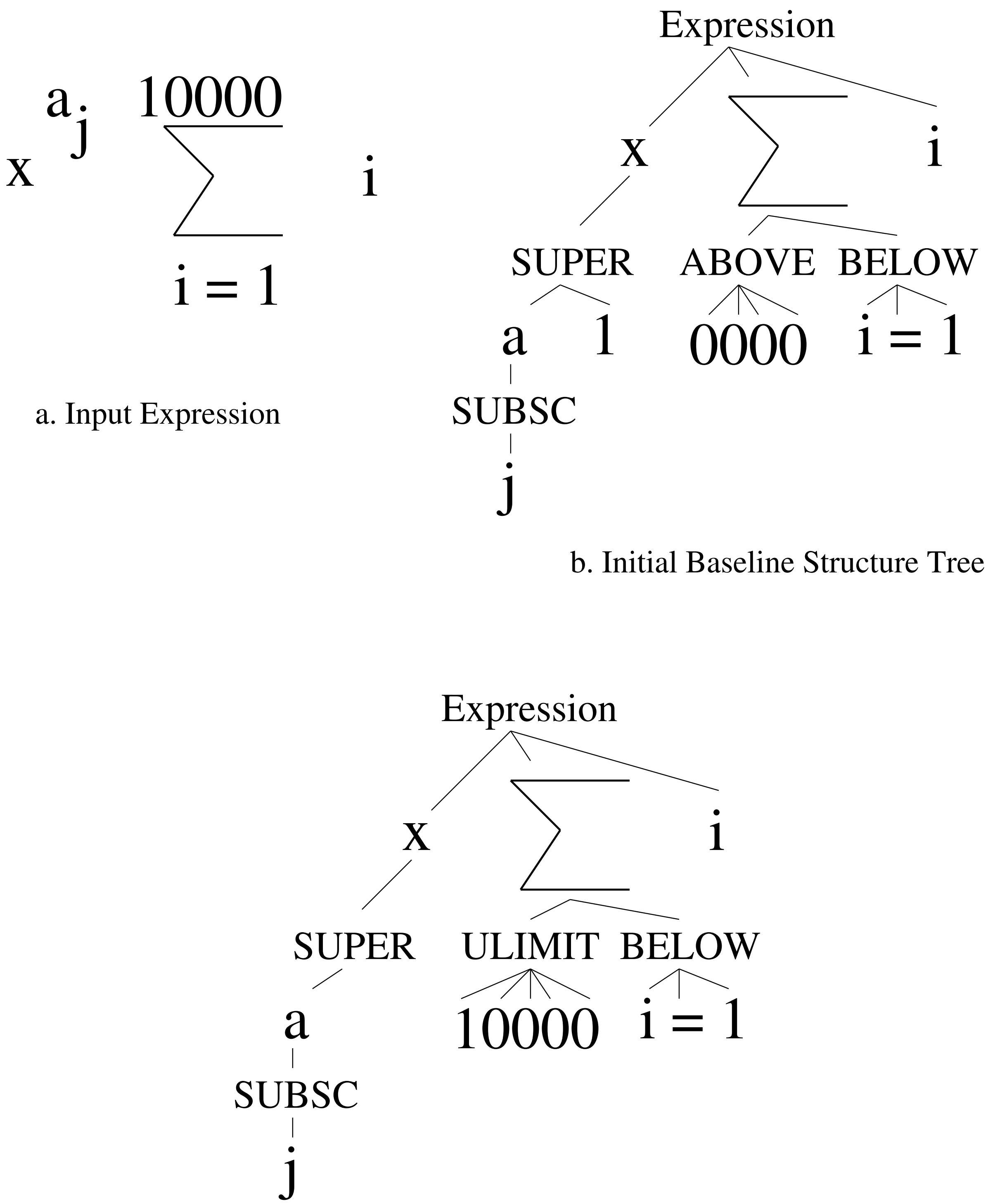
A linear representation of a baseline structure tree, called a *positional sentence*[22], may now be obtained by performing a depth-first traversal of the tree. The structure of an expression in a positional sentence translated from a baseline structure tree is a recursive structure of the form:

$$s_0 \text{ [ABOVE}\{SB_1\}] \text{ [BELOW}\{SB_2\}] \text{ [SUBSC}\{SB_3\}] \text{ [SUPER}\{SB_4\}]$$
$$\text{[ULIMIT}\{SB_5\}] \text{ [BLIMIT}\{SB_6\}] \text{ } [s_1]$$

All items in square brackets are optional, and each nested baseline (SB_*) posesses the same structure as above.

A LaTeX string can be obtained in a similar fashion, mapping the appropriate symbol names and/or contexts in the tree to the corresponding LaTeX symbol. Alternately the baseline structure tree may be rewritten into a LaTeX compatible form (i.e. replacing symbols and contexts) and then directly translated to a string using a depth-first traversal. In either case the root ("Expression") needs to be removed in order to create a legal LaTeX string.

In theory this type of translation is also possible for Maple and Mathematica or other computer algebra system languages, but this involves additional semantic issues discussed in Chapter 7, and is beyond the scope of this thesis.



a. Input Expression

b. Initial Baseline Structure Tree

c. Rewritten Baseline Structure Tree

Figure 4.5: Tree Rewriting

Chapter 5

Implementation and Test Results

Based on the general parser outlined in Chapter 4 a parsing application was built, the Diagram Recognition Application for Computer Understanding of Large Algebraic Expressions (DRACULAE). DRACULAE was developed using an existing mathematics entry system called FFES, developed by Steve Smithies at the University of Otago, New Zealand. FFES had a parser of its own which was based on existing graph grammar techniques[47] but which was very restricted in terms of number of symbols and layout that the system could handle. DRACULAE was developed with the intention to augment and/or replace the graph grammar-based parser. The FFES graphical user interface is shown in Figure 5.1.

DRACULAE was written in Java, and FFES was implemented in Tcl/Tk and C++. DRACULAE has been interfaced with FFES through alteration of the FFES Tcl/Tk code. Both DRACULAE and FFES have been built on Linux platforms, and DRACULAE has also been succesfully built under Solaris.

At present, only LaTeX and positional sentences are available as output, i.e. structural representations. In the last chapter we discuss issues in obtaining mappings to semantic representations such as Maple and Mathematica strings.

In the following a general overview of FFES and DRACULAE are provided, followed by a discussion of test results for DRACULAE.

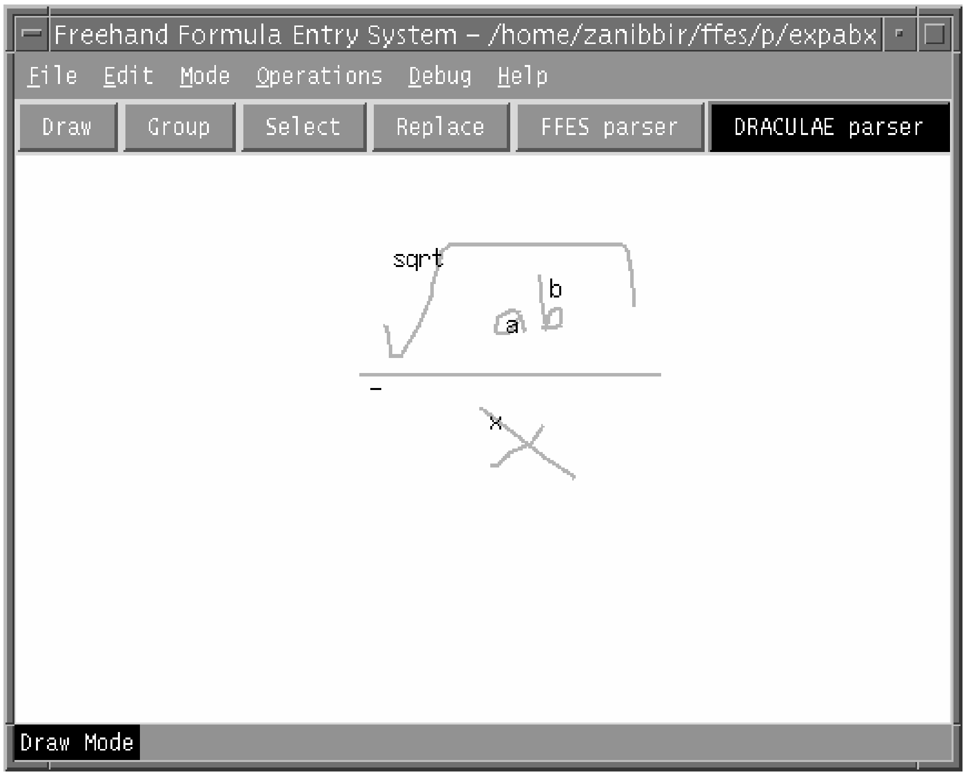


Figure 5.1: The Freehand Formula Entry System

5.1 The Freehand Formula Entry System

In this section we briefly outline the FFES system. For greater detail, please see [47, 48].

The Freehand Formula Entry System is designed to allow an individual to input math expressions to a computer using a mouse or data tablet. The system is comprised of a graphic user interface, a graph-grammar parser and a nearest-neighbour symbol recognizer (created by Jim Arvo at Caltech).

The user draws a number strokes, which FFES tracks and groups for recognition in the background. After a specifiable time delay or after four strokes have been made (the maximum number of strokes needed for the set of symbols recognized), the symbol recognizer is called on all possible partitions of the strokes sent to the recognizer. Note that Jim Arvo has indicated that a newer version sends only $O(n^2)$ sets of stroke partitions to the recognizer.

The system allows easy correction of any stroke grouping or symbol identity recognition errors. The "Group" button allows the user to select strokes to be joined by drawing a line through the strokes to be grouped. The program then performs recognition on the selected group and any strokes separated by the grouping.

If a symbol label is incorrect, the "Replace" button allows the user to click the mouse on the misrecognized symbol, and then select from a pull-down list of labels (in order of confidence from the symbol recognizer) or type a labelling string in from the keyboard.

In addition, the spatial position of symbols may be moved using the "Select" button. The user can select individual symbols or groups of symbols and move them in the image. This is useful for interaction between the parser output and the user; inputs which have their structure misparsed can be easily altered.

The original graph-grammar based parser may be called on the recognized symbols using the "FFES parser" button, while the "DRACULAE parser" calls the implementation of the baseline structure-tree based parsing algorithm described in the last chapter. The FFES parser was kept in the application for comparison only, and in the future will be removed.

The error-correction facilities of FFES proved invaluable during development of DRACULAE, because this allowed our research to assume the presence of a perfect symbol recognizer. Because of the clear labelling in FFES, any mis-labelled symbols in the image input can be quickly located and corrected. While the user may occassionally miss mis-classified symbols, these errors are easily detected. Likewise, if the layout of symbols confuses the parser, this can be easily corrected through directly manipulating the location of the symbols. Currently DRACULAE parses in under two seconds on average on a Pentium-III 450MhZ Linux system, with a two to ten second delay to create the graphical user interface window. This is due to the implementation language (Java) rather than the complexity of the parser.

FFES was tested with human users, and was found to be a promising way to input mathematical expressions to a computer[47]. The largest drawback in the system, time and accuracy wise was the existing parser.

5.2 DRACULAE implementation

DRACULAE is comprised of a parser and a graphic user interface. DRACULAE takes a list of symbols as input (from any source providing bounding box and symbol identity information), and passes it to the parser. Then the baseline structure tree output by the parser is passed to the graphic user interface, which displays the tree and calls translators to display the LaTeX and positional sentence translations, as shown in Figure 5.2. Tabs are used to switch between the "Image Viewer", which displays an image of the LaTeX processed string shown in the Output String line (e.g. $\frac{\sqrt{ab}}{x}$), and the "Tree Viewer", which shows the baseline structure tree and symbol attributes.

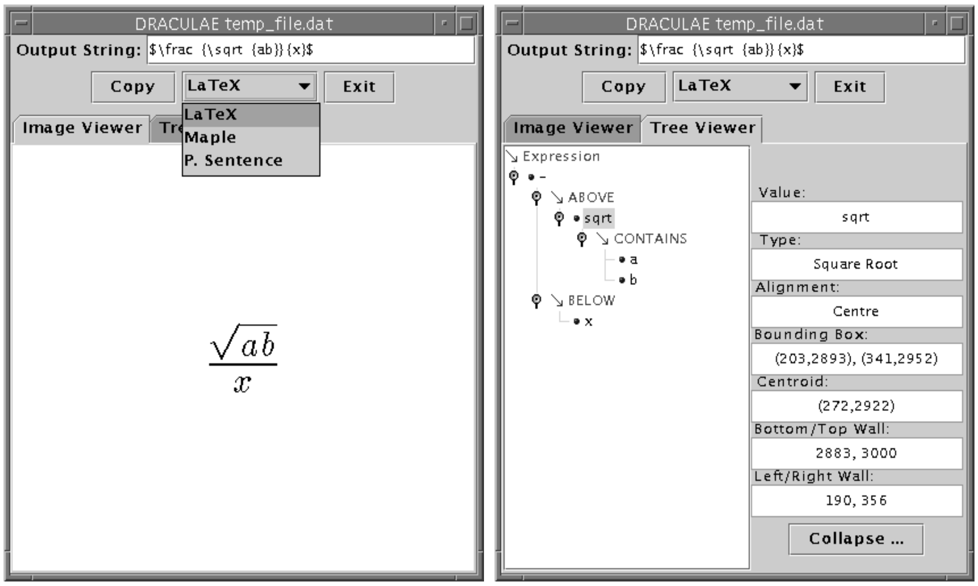


Figure 5.2: DRACULAE Graphic User Interface (Image and Tree Views Shown)

Java was an ideal prototyping language for the following reasons. First, it creates a (theoretically) platform independent prototype. Second, the javadoc facility was appealing from a program understanding and analysis point of view. Third, the "Swing" library allowed automated visualization of the tree structure, which was crucial to the program design.

DRACULAE is relatively fast, parsing in under a second for many large expressions. In Linux the largest delay seems to result because the version of Java which was available did not posess a just-in-time compiler, resulting in purely interpreted execution. Window creation and display is particularly slow (taking up to 10 seconds at times).

In addition to being relatively fast, DRACULAE is robust. If symbols are missed in the input, a message is sent to the terminal along with the symbols missed. The part of the expression that was parsed is passed along. If no input is passed, a single node labelled "Expression" (the root of the tree) is returned.

5.3 Test Results

Through FFES, DRACULAE has been tested on hundreds of mathematical expressions. There are approximately one hundred test cases which act as a test suite. Examples of inputs which have been properly recognized are presented in Figures 5.3 to 5.7. Figures 5.3 and 5.5 are taken from a Calculus textbook[41], while Figure 5.4 is presented as an example of limit handling in DRACULAE. Figure 5.6 shows an expression with accents properly handled by DRACULAE, and finally Figure 5.7 demonstrates a large, complex expression recognized by DRACULAE.

5.3.1 General Results

The following are some general results obtained during testing.

- On average the time from requesting a parse from DRACULAE to viewing the graphical user interface is under ten seconds.
- We have tested on expressions with more than 40 symbols in complex layouts, and noticed little or no performance discrepancy compared with smaller expressions.
- DRACULAE is robust. If a misparse occurs, the generated positional sentence and LaTeX strings are returned, and a LaTeX image created. In this case the structure of the baseline structure tree may be observed if the image does not offer enough information. The LaTeX string will often have the spatial relations explicitly in the output image in

this case (e.g. ABOVE, SUPER etc. appear directly in the generated image). Empty input is handled by returning a baseline structure tree with a single root and the string "Expression" as LaTeX output.

- As DRACULAE performs no semantic analysis, syntactically invalid inputs (e.g. with unmatched parentheses) are handled without difficulty, provided the baseline structure is clear to DRACULAE.

- The thresholds used for regions work reasonably well if there is little skew in the input expression. However, slanted expressions result in improper subscripting or superscripting.

- Horizontal lines are mapped to division lines, subtraction symbols, overline (e.g. boolean negation) or underline depending on the context in the baseline structure tree. For instance, if a horiztonal line has symbols above and below, LaTeX is instructed to create a fraction. If the line has an argument above or below, underline or overline strings are created. Finally, if no symbols are above or below the line, the line is translated as a subtraction symbol. Horizontal lines are thus allowed to interact in complex ways with little ambiguity.

- When horizontal lines overlap less than the required thresholded amount for overlap detection, unusual outputs are created.

- Limit symbols (e.g. $\int, \sum$) may have overlapping limits (see Figure 5.3), and these limits may be of almost arbitrary complexity, as shown in Figure 5.4. However, due to the current definition of START, the dominant limit symbol must be leftmost in the case of an expression such as that given in Figure 5.4. This could be resolved through incorporating size information into START's definition.

- There is a rewrite rule to collect separated $=$ signs using context in the tree. The particular version that has been implemented does not work if the $=$ is in a limit, but otherwise performs well. This could be resolved by a constraint on the length that a line must be to be considered "overlapping" a limit symbol.

- Many expressions which are not translated to LaTeX appropriately are nontheless recognized accurately. For instance, choice notation created using single integers and/or variables have the relative position of the

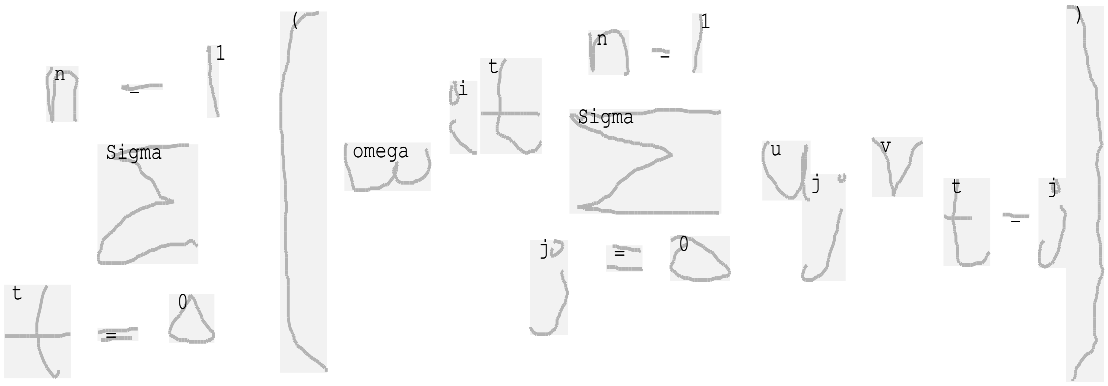


Figure 5.3: Test Input Example 1

symbols correctly represented in the baseline structure tree. A tree rewrite to group the expression into an appropriate LaTeX string simply hasn't been created yet.

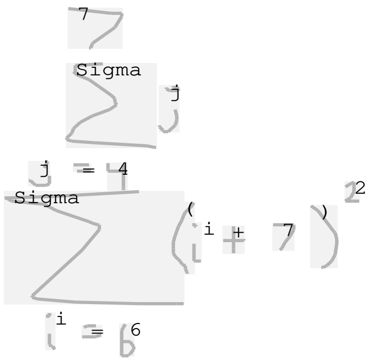


Figure 5.4: Test Input Example 2

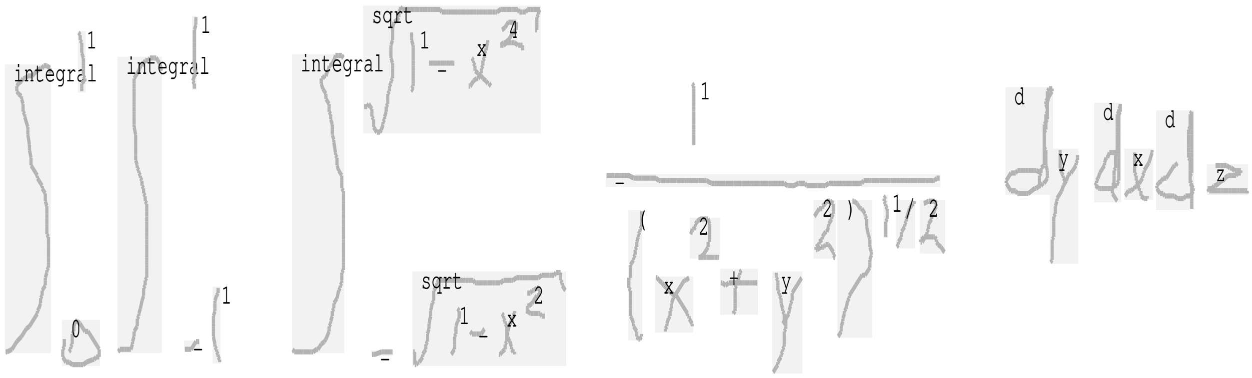


Figure 5.5: Test Input Example 3

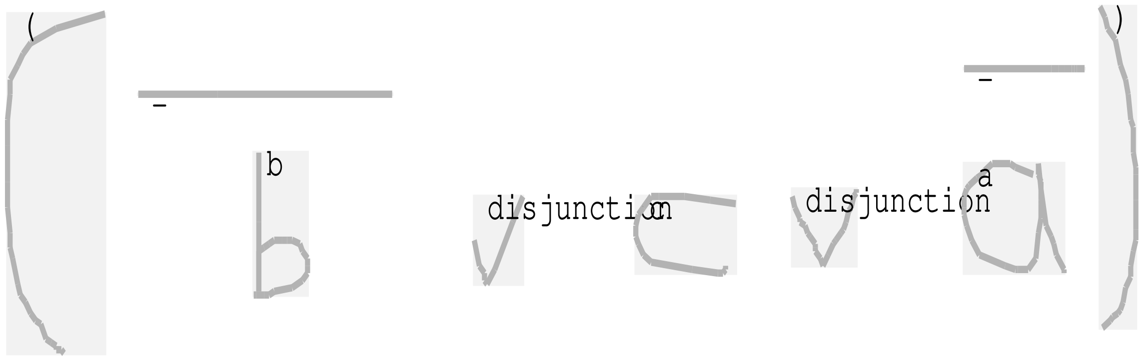


Figure 5.6: Test Input Example 4

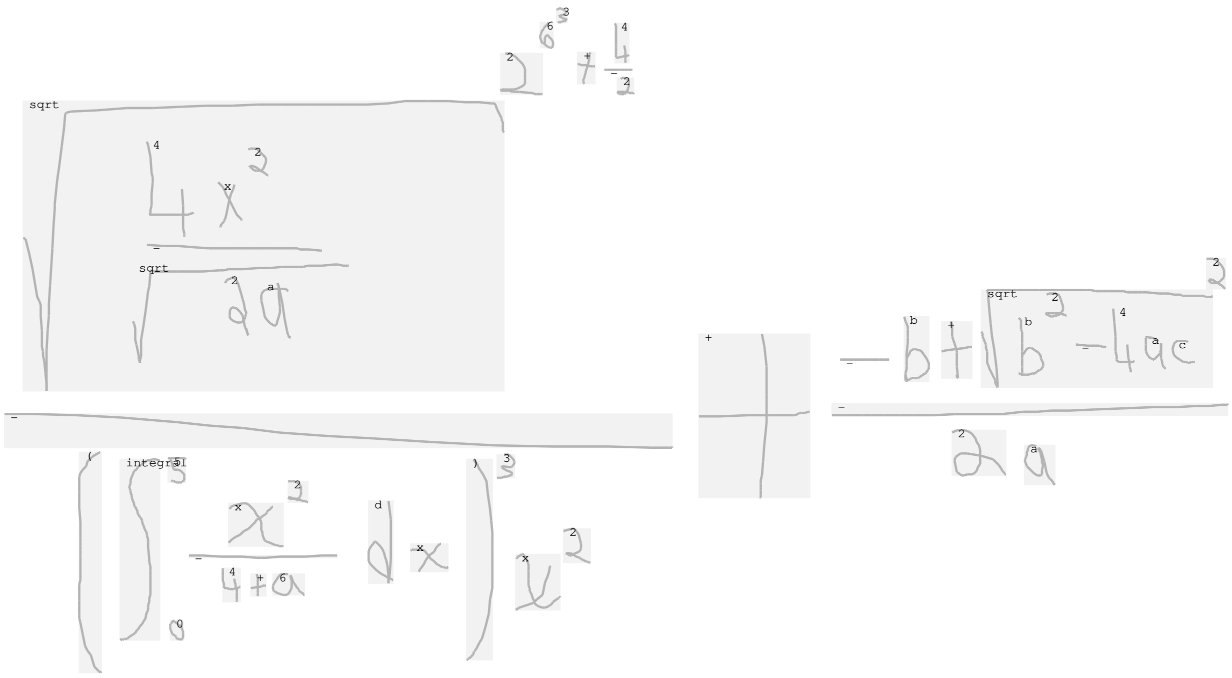


Figure 5.7: Test Input Example 5

Chapter 6

Future Work and Conclusion

In this chapter future work arising from this research is discussed. Improvements which may to be made to DRACULAE are outlined, along with more general issues such as reimplementation of DRACULAE in TXL, semantic issues which arise in mapping from baseline structure trees to semantic interpretations, and diagrammatic notations other than mathematics notation which may benefit from a recognition method similar to baseline extraction.

6.1 Limitations of DRACULAE

6.1.1 Starting Symbol Definition

As noted in the last chapter, Figure 5.4 would not be successfully parsed if the $\sum$ in the upper limit were moved to the left of the lower $\sum$. This is because the operator dominance analysis currently performed by START does not take symbol size into account. It also was not originally clear that this was an analysis of operator dominance rather than simply a spatial analysis. With this new information, DRACULAE would benefit greatly from a more generally defined START function with a better defined operator dominance analysis.

6.1.2 Whitespace

The current implementation of DRACULAE performs no tokenization of input, and does not provide rewrite rules to group integers, separate function names, etc. This would not be difficult to add to the current system, however analysis of whitespace is necessary for creating a more mature system.

Futher, neither matrices or multiple line expressions are recognized properly using DRACULAE. It is unclear whether simple rewrites could be created to handle these cases, though it seems more likely that segmenting the input and sending single-line expressions to DRACULAE would result in better performance due to the potential complexity of analysis.

6.1.3 Sensitivity to Symbol Size

Very large symbols have very large regions, and small symbols vice-versa. It is unclear whether representation of symbols by single points in handwritten expressions is reasonable, or whether some additional bounding box and/or pixel information would result in better performace, particularly in situations where skew is present.

6.1.4 Pixel Level Information

DRACULAE cannot, nor will it be able to handle structures which require pixel-level information. As an example, consider nth-roots. Without knowing where the line dividing the expression into inner and outer parts of the square root is, it is impossible to differentiate a value in the scope of a root from a value indicating the degree of a root.

An arbitrary threshold could be used, but this would restrict the types of expressions that could be accepted by DRACULAE.

In order to locate the degree of a root without uncertainty, DRACULAE would need access to the position of the dividing line. Kerned symbols are another example where pixel level information would result in improved performance, for instance in the case of T_n, which DRACULAE would currently recognize as T BELOW {n} if the T bounding box overlaps the centroid of the n.

6.2 TXL Implementation of DRACULAE

Java was a convenient language for prototyping DRACULAE, abstracting a great deal of low level details. However, it is not easy to express tree rewriting rules in Java, requiring a fairly large amount of detailed code for even a single rewrite. Additionally, while the Java implementation of DRACULAE is fast, it remains to be seen how rapidly a version in a compiled language would execute.

TXL is a programming language which performs a process identical to that used for DRACULAE: a tree is built using a grammar, the tree is rewritten using a set of productions, and then the tree is translated into an output string[16]. TXL allows attributed nodes in its trees, making the translation from Java to TXL relatively easy. TXL represents tree productions using a compact and simple syntax; this makes extension and maintenance of a system like DRACULAE easier than if the system is programmed in programming language such as C, C++ or Java. To extend DRACULAE as it currently is to a real system capable of handling dialects, Figure 6.1 is proposed.

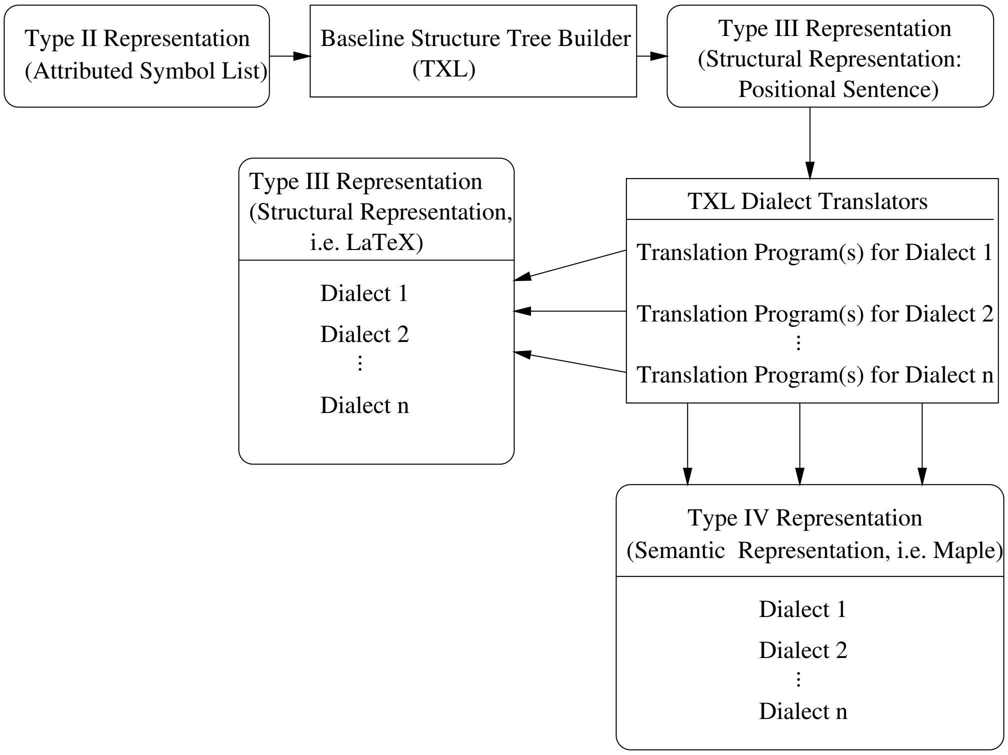


Figure 6.1: Proposed TXL Reimplementation of DRACULAE

6.3 Issues in Semantics of Mathematics Notation

As stated earlier, mathematics notation is a natural visual language and as a result does not possess a fixed semantic interpretation. Therefore, it would be worth making a study to determine which parts of mathematical semantics, if any, are fixed (i.e. are hard conventions) and which fluctuate by dialect (i.e. are soft conventions).

In order for a semantic interpretation to be made, the baseline structure obtained using a system such as DRACULAE must be broken/rewritten into tokens: integer values must be distinguished from decimal values, exponential numbers, function names, variable names, and so on. The current system makes no such analysis, though this is clearly necessary for both displaying more complicated expressions (i.e. to indicate to LaTeX where whitespace is to appear) and for semantic interpretation and/or evaluation of mathematical expressions that have had their baseline structure extracted. This would not be difficult to add directly to DRACULAE in the form of tree rewrite rules.

In the case of simple arithmetic, a semantic interpretation may obtained through mapping from the baseline structure tree to the operator tree for the expression[40], as shown in Figure 6.2.

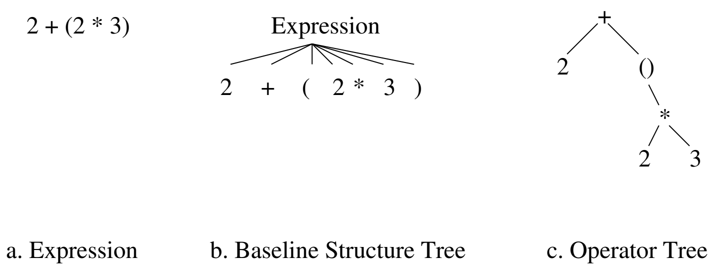


Figure 6.2: Simple Arithmetic Expression and Representations

The operator tree may be obtained from the baseline structure tree using operator dominance[12]. The operator tree indicates the order of application of the operators bottom-up.

For more complex mathematical expressions, a computer algebra system language such as Maple or Mathematica may be used for representation. For this type of representation to be constructed, a priori semantic information is required. For instance, to produce appropriate output for a Maple program that uses a function p, there must be a facility to indicate that p(a) is in fact function applciation and not implied multiplication of terms.

It would be worth studying what other types of a priori semantic information are needed to produce valid semantic interpretations for different dialects of mathematics notation. With such information, DRACULAE or a similar system could be extended to produce strings in a computer algebra system format (e.g. Maple) for different dialects of mathematics notation, as shown in Figure 6.1.

6.4 Use of Directed Recognition Algorithms in Other Notations

The algorithm used in this thesis exploits the *direction* inherent in baselines. Given the ability to locate the beginning of a baseline symbol set, an algorithm may be defined in order to progress left-to-right to find the other symbols on a given baseline. In a divide and conquer fashion, baseline symbol sets are located recursively using binary spatial relations.

Other notations besides mathematics notation have a clear element of direction, and it may be possible to recognize the structure of these notations in a similar fashion. Music notation for instance has direction inherent in the progression of notes, left-to-right, across a staff. The analogy between baselines in mathematics notation and a voice in music notation is strong; in fact, a single musical voice can be viewed as a type of symbol set similar to a baseline symbol set. A complicating factor in music notation is that unlike baselines, voices in music notation can overlap, and as a result it is not trivial to determine which voice a note belongs to. It is worth examining whether a partially or completely dialect-neutral representation of the structure of music notation may be obtained. There are many dialects of music notation, so it is unclear whether this is possible.

Circuit diagrams also have an element of direction: the direction of current flow. If power sources can be located in a diagram, it may be possible to employ the direction of circuit flow directly to aid recognition.

6.5 Conclusion

The research presented in this document has established the following thesis.

> Through separating spatial structure from semantics in mathematics notation, a general and flexible recognition of mathematics expressions may be obtained.

As a reminder, *general* is used to indicate that dialects of mathematics notation may be conveniently handled. *Flexible* refers to the ability to handle a large range of symbol placements.

The following contributions which support this thesis have been discussed.

1. A model for the structure of mathematics notation

   A novel model, called the *baseline structure tree*, was introduced in Chapter 3. This model represents spatial structure between baselines in mathematics expressions. It is demonstrated in Chapter 3 how a baseline structure tree may be rewritten in order to perform syntactic analysis for different dialects of mathematics notation.

2. A visual language parsing algorithm

   In Chapter 4 an algorithm which creates baseline structure trees from attributed symbol lists is described. The parsing algorithm presented is more flexible than exisiting mathematics notation recognition systems because the spatial relationships used to obtain baseline structure may be redefined, and tree rewriting may be used to handle soft conventions (i.e. dialects).

3. An implementation

   An implementation of the visual language parsing algorithm was presented in Chapter 5. The implementation is named the Diagram Recognition Application for Computer Understanding of Large Algebraic Expressions (DRACULAE). Chapter 5 outlines how DRACULAE has been integrated into a complete recognition system for handwritten mathematics notation, the Freehand Formula Entry System (FFES). The symbol recognition and user interface components of FFES were created by Steve Smithies, Jim Arvo and Kevin Novins [48]. The system has been tested on hundreds of handwritten expressions with excellent results. FFES and DRACULAE were demonstrated at CASCON '99 and were enthusiastically received. Public distribution of the system is planned.

Bibliography

[1] R.H. Anderson. *Syntax-Directed Recognition of Hand-Printed Two-Dimensional Equations.* PhD thesis, Harvard University, Cambridge, MA, January 1968.

[2] R.H. Anderson. Two-dimensional mathematical notation. In K.S. Fu, editor, *Syntactic Pattern Recognition*. Springer-Verlag, New York, 1977.

[3] Stephan Baumann. A simplified attributed graph grammar for high-level music recognition. In *Proc. Third Intl. Conf. on Document Analysis and Recognition*, Montreal, Canada, August 1995.

[4] Abdelwaheb Belaid and Jean-Paul Haton. A Syntactic Approach for Handwritten Mathematical Formula Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(1):105–111, January 1984.

[5] Benjamin P. Berman and Richard J. Fateman. Optical character recognition for typeset mathematics. In *Proceedings of the 1994 International Syposium on Symbolic and Algebraic Computation*, pages 348–353, July 1994.

[6] Dorothea Blostein. General diagram-recognition methodologies. In *Lecture Notes in Computer Science*, volume 1072, pages 106–122. Springer-Verlag, New York, 1995.

[7] Dorothea Blostein and Ann Grbavec. Recognition of mathematical notation. In *Handbook of Character Recognition and Document Image Analysis*, pages 557–582. World Scientific Publishing Company, 1997.

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

[9] Florian Cajori. *A History of Mathematics*. Chelsea Publishing Company, New York, 1919.

[10] Florian Cajori. *A History of Mathematical Notations*. The Open Court Publishing Company, Chicago, Illinois, 1929. 2 vols.

[11] G. Castagliola, A. De Lucia, S. Orefice, and G. Tortora. A framework of syntactic models for the implementation of visual languages. In *Proc. 1997 Symposium on Visual Languages*, pages 58–65, 1997.

[12] Shi-Kuo Chang. A method for the structural analysis of two-dimensional mathematical expressions. *Information Sciences*, 2:253–272, 1970.

[13] T.W. Chaundy, P.R. Barrett, and Charles Batey. *The Printing of Mathematics*. Oxford University Press, London, 1957.

[14] Ling-Hwei Chen and Peng-Yeng Yin. A system for on-line recognition of handwritten mathematical expressions. *Computer Processing of Chinese and Oriental Languages*, 6(1):19–39, June 1992.

[15] P. A. Chou. Recognition of equations using a two-dimensional stochastic context-free grammar. In W. A. Pearlman, editor, *Visual Communications and Image Processing IV*, volume 1199 of *SPIE Proceedings Series*, pages 852–863, 1989.

[16] J.R. Cordy, C.D. Halpern, and E. Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, Jan 1991.

[17] Genarro Costagliola and Shi-Kuo Chang. Parsing linear pictorial languages by syntax-directed scanning. *Languages of Design*, 2:223–242, 1994.

[18] Genarro Costagliola, Andrea De Lucia, and Sergio Orefice. Towards efficient parsing of diagrammatic languages. In *Proceedings of Advanced Visual Interfaces*, pages 162–171. ACM Press, 1994.

[19] Genarro Costagliola, Andrea De Lucia, and Sergio Orefice. A parsing methodology for the implementation of visual systems. *IEEE Transactions on Software Engineering*, 23(12), December 1997.

[20] Genarro Costagliola, Andrea De Lucia, Sergio Orefice, and Genny Tortora. Positional grammars: A formalism for LR-like parsing of visual languages. In *Visual Language Theory*, pages 171–191. Springer-Verlag, New York, 1998.

[21] Gennaro Costagliola and Shi-Kuo Chang. Using linear positional grammars for the LR parsing of 2-d symbolic languages. draft paper; for copy contact gencos@dia.unisa.it, 1998.

[22] Gennaro Costagliola, Andrea De Lucia, and Sergio Orefice. Towards efficient parsing of diagrammatic languages. In *Proceedings of Advanced Visual Interfaces 1994*, pages 162–171. ACM Press, 1994.

[23] Yannis A. Dimitriadis and Juan López Coronado. Towards an art based mathematical editor, that uses on-line handwritten symbol recognition. *Pattern Recognition*, 28(6):807–822, 1995.

[24] Dov Dori and Amir Pneuli. The grammar of dimensions in machine drawings. *Computer Vision, Graphics and Image Processing*, 42:1–18, 1988.

[25] Talaat Salem El-Sheikh. Recognition of handwritten arabic mathematical formulas. In *United Kingdom Information Technology Conference*, March 1990.

[26] Hoda Fahmy and Dorothea Blostein. A graph grammar programming style for recognition of music notation. *Machine Vision and Applications*, 6:83–89, 1993.

[27] Hoda Fahmy and Dorothea Blostein. A graph-rewriting paradigm for discrete relaxation: Application to sheet music recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, August 1997.

[28] Richard J. Fateman and Taku Tokuyasu. Progress in recognizing typeset mathematics. In *Proceedings of the International Society for Optical Engineering*, volume 2660, 1996.

[29] Claudie Faure and Zi Xiong Wang. Automatic perception of the structure of handwritten mathematical expressions. In R. Plamondon and C. G. Leedham, editors, *Computer Processing of Handwriting*, pages 337–361. World Scientific Publishing Co., 1990.

[30] K.S. Fu. *Syntactic Pattern Recognition.* Springer-Verlag, New York, 1977.

[31] Ann Grbavec. Recognition of mathematics notation using graph rewriting. Master's thesis, Queen's University, Kingston, Ontario, Canada, January 1995.

[32] Ann Grbavec and Dorothea Blostein. Mathematics recognition using graph rewriting. In *Third Intl. Conf. on Document Analysis and Recognition*, Montreal, August 1995.

[33] Nicholas J. Higham. *Handbook of Writing for the Mathematical Sciences.* Society for Industrial and Applied Mathematics, Philadelphia, 1993.

[34] Donald E. Knuth. *TeX and METAFONT - New Directions in Typesetting.* Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1979.

[35] Andreas Kosmala and Gerhard Rigoll. On-line handwritten formula recognition using statistical methods. In *Proceedings of the Fourteenth International Conference on Pattern Recognition*, pages 1306–1308, August 1998.

[36] Stéphane Lavirotte and Loïc Pottier. Optical Formula Recognition. In *Proc. 4th International Conference on Document Analysis and Recognition*, volume 1, pages 357–361, Ulm, Germany, 1997.

[37] Hsi-Jian Lee and Jiumn-Shine Wang. Design of a mathematical expression recognition system. In *Proceedings of the third International Conference on Document Analusis and Recognition*, pages 1084–1087, 1995.

[38] Hsi-Juan Lee and Min-Chou Lee. Understanding Mathematical Expressions Using Procedure-Oriented Transformation. *Pattern Recognition*, 27(3):447–457, 1994.

[39] Kim Marriott, Bernd Meyer, and Kent D. Wittenburg. A survey of visual language specification and recognition. In *Visual Language Theory*, pages 5–85. Springer-Verlag, New York, 1998.

[40] William A. Martin. Computer Input/Output of Mathematical Expressions. In *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, pages 78–89, March 1971.

[41] William G. McCallum, Deborah Hughes-Hallett, Andrew M. Gleason, and et al. *Multivariable Calculus: Draft Version*. John Wiley and Sons, Inc., 1994.

[42] Eric G. Miller and Paul A. Viola. Ambiguity and constraint in mathematical expression recognition. In *Proceedings of the 15th National Conference of Artificial Intelligence*, Madison, Wisconsin, July 1998. American Association of Artificial Intelligence.

[43] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[44] Masayuki Okamoto and Akira Miyazawa. An experimental implementation of a document recognition system for papers containing mathematical expressions. In H.S. Baird H. Bunke and K. Yamamoto, editors, *Structured Document Image Analysis*, pages 36–53. Springer-Verlag, New York, 1992.

[45] Nasayuki Okamoto and Bin Miao. Recognition of Mathematical Expressions by Using the Layout Structures of Symbols. In *Proceedings of the First International Conference on Document Analysis and Recognition*, volume 1, pages 242–250, Saint-Malo, France, 1991.

[46] Giulia M. Pagallo. Constrained attribute grammars for recognition of multi-dimensional objects. In *Advances in Pattern Recognition*, pages 359–365. Springer-Verlag, 1998.

[47] Steve Smithies. Freehand formula entry system. Master's thesis, University of Otago, Dunedin, New Zealand, May 1999.

[48] Steve Smithies, Kevin Novins, and James Arvo. A Handwriting-Based Equation Editor. In *Proc. Graphics Interface*, Kingston, Ontario, Canada, June 1999.

[49] Hashim M. Twaakyondo and Masayuki Okamoto. Structure Analysis and Recognition of Mathematical Expressions. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, volume 1, Montréal, Canada, 1995.

[50] Zi-Xiong Wang and Claudie Faure. Structural Analysis of Handwritten Mathematical Expressions. In *Proceedings of the Ninth International Conference on Pattern Recognition*, pages 32–34, 1988.

[51] H.-J. Winkler, H. Fahrner, and M. Lang. A Soft Decision Approach for Structural Analysis of Handwritten Mathematical Expressions. In *International Conference on Acoustics, Speech and Signal Processing*, pages 2459–2462. IEEE, 1995.

[52] Hans-Jürgen Winkler and Manfred Lang. Symbol segmentation and recognition for understanding handwritten mathematical expressions. In *Progress in Handwriting Recognition*. World Scientific, Singapore, 1997.

[53] Yanjie Zhao, Tetsuya Sakurai, Hiroshi Sugiura, and Tatsuo Torii. A methodology of parsing mathematical notation for mathematical computation. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, July 1996.

Appendix A

A Positional Grammar for Baseline Structure

In this appendix a *positional grammar* is presented which represents the structure of baselines in mathematics notation (i.e. a syntax of baseline structure trees). The strings derived from the grammar are positional sentences which correspond to baseline structure trees.

A positional grammar is an attributed context-free grammar augmented by a set of positional relations which are explicit in the derived parse string. The output of a *positional parser* for a positional grammar is a string called a positional sentence, which alternates terminals of the grammar with positional relations (see [22, 18, 17, 11, 19, 21, 20] for more details). There is a class of positional grammars for which fast (i.e. $O(n^2)$) LR parsers with actions may be constructed using an automatic parser generator, such as YACC.

In a positional grammar multiple positional relations on one symbol are presented using enumeration of relations in the positional sentence. For example, in

$$aHOR_0\{c\}SUPER_1\{2\}SUBSC_2\{i\}$$

the numeral n after each relation indicate that it applies to the symbol on the immediate right, and the n-th last symbol in the string. The above example represents a_i^2c.

Figure A.1 defines a positional grammar for the structure of baselines in a mathematics expression. Note that SYMBOL is a terminal of the grammar,

1. E $\rightarrow$ **SP** BLSS
2. BLSS $\rightarrow$ BLS **HOR** BLSS | ϵ *
3. BLS$\rightarrow$ SYMBOL B C D E F G H
4. B $\rightarrow$ **TLEFT** {BLSS} | ϵ
5. C $\rightarrow$ **BLEFT** {BLSS} | ϵ
6. D $\rightarrow$ **ABOVE** {BLSS} | ϵ
7. E $\rightarrow$ **BELOW** {BLSS} | ϵ
8. F $\rightarrow$ **SUPER** {BLSS} | ϵ
9. G $\rightarrow$ **SUBSC** {BLSS} | ϵ
10. H $\rightarrow$ **CONTAINS** {BLSS} | ϵ

Figure A.1: Positional Grammar of Baseline Structure

representing an attributed symbol. Attributes are propagated from the left hand side to the right hand side. This is done in the following way:

1. Rule 1 will assign the wall attributes of BLSS to the pair ((-1,-1),(∞,∞))

2. In Rule 2 the wall attributes of the left-hand BLSS (baseline symbol set) are passed to both of the nonterminals on the right hand side. The rightWall attribute of BLS (baseline symbol) is then set to the minimum bounding box coordinate for the symbol represented by the BLSS nonterminal on the right side of the rule (this provides the necessary partition for SUBSC and SUPER regions).

3. In Rule 3 the wall attributes of BLS are passed to SYMBOL and all nonterminals (B to H).

4. In Rules 4-10 if a symbol is located the BLSS on the right hand side inherits wall attributes equivalent to the region within which the symbol was found.

A problem with the grammar above is that the grammar presents context through bracketing, which is not present in the original positional grammar formalism. As a result, the manner in which to cleanly construct the bracketing in a parser for the grammar above has not presented itself. The algorithm in Chapter 4 is essentially equivalent to a top-down parser based on the grammar above. For that parsing algorithm, the bracketing issue is resolved by constructing a baseline structure tree during the parse, and then bracketing based on context while translating the tree into a string.

Attributes need to be synthesized top-down for the grammar above, and as a result an LR parser could not be constructed for the grammar even if the bracketing issue were resolved. More research is needed to see whether the techniques used by Costagliola et. al. to create LR parsers with actions may be generalized to create LL parsers with actions. The advantage of such a generalization is that it would allow the use of automatic parser generators to build LL as well as LR parsers for visual languages describable by positional grammars.

Appendix B

Algorithms

B.1 START and OVERLAP

In the following we define spatial functions START(R,$list_{in}$), where R is a region, and $list_{in}$ an input list. START(R,$list_{in}$) returns the starting symbol s_{start} of input list $list_{in}$ in region R. We assume that $list_{in}$ is a list sorted by leftmost coordinate, indexed from 1 to the number of elements in the list.

START Let $list_{in}$ be the passed input list

Let ($leftWall$,$bottomWall$),($rightWall$,$topWall$) be the passed values defining a region R

Let $leftmostIndex$:= -1

Let $limitIndex$:= -1

Let $listIndex$:= 1

Let $overlapIndex$:= -1

Let n be the number of items in $list_{in}$

While $leftmostIndex$ = -1 and $listIndex \leq n$

- If $list_{in}(listIndex)$ is an unmarked symbol with its centroid in region R, let $leftmostIndex$:= $listIndex$
- else let $listIndex$:= $listIndex$ + 1

If $leftmostIndex$ = -1 then return $leftmostIndex$

else

- While $listIndex \leq n$ and $limitIndex$ = -1

  - If $list_{in}(listIndex)$ is an unmarked limit symbol in region R then let $limitIndex := listIndex$
  - else $listIndex = listIndex + 1$
- If $limitIndex$ = -1 or $limitIndex = leftmostIndex$ return OVERLAP($leftmostIndex$,$topWall$,$bottomWall$,$list_{in}$)
- else
  - Let $upperThreshold$ be the maximum y bounding box coordinate value at $list_{in}(limitIndex)$
  - Let $lowerThreshold$ be the minimum y coordinate bounding box value at $list_{in}(limitIndex)$
  - While $listIndex > leftmostIndex$
    * $listIndex := listIndex$ - 1
    * If the centroid y coordinate of the symbol at $list_{in}(listIndex)$ $<$ $upperThreshold$ and the same y centroid coordinate $\geq$ $lowerThreshold$ then let $overlapIndex := listIndex$
  - If $overlapIndex < limitIndex$ then return OVERLAP($leftmostIndex$,$topWall$,$bottomWall$,$list_{in}$)
  - else return OVERLAP($limitIndex$,$topWall$,$bottomWall$,$list_{in}$)

END OF ALGORITHM

Next we describe OVERLAP(symbolIndex,topWall,bottomWall, $list_{in}$).

OVERLAP Let $symbolIndex$ be the passed index to $list_{in}$

Let $topWall$ and $bottomWall$ be passed y-coordinates

Let $list_{in}$ be the passed input list

Let $listIndex := symbolIndex$

Let $stop$:= false

Let n be the number of items in $list_{in}$

If $list_{in}(symbolIndex)$ is a line, then let $maxLength$ be maxX - minX of that symbol.

else let $maxLength$:= -1

Let $mainLine$:= -1

While $listIndex > 1$ and $stop$ = false

- If $list_{in}(listIndex$ - 1) contains a symbol which has a minX bounding box coordinate less than that of the symbol at $list_{in}(symbolIndex)$ then $stop$:= true

- else *listIndex* := *listIndex* - 1

While *listIndex* $\leq$ n and minX bounding box coordinate of the symbol at $list_{in}(listIndex)$ is less than maxX of $list_{in}(symbolIndex)$

- If $list_{in}(listIndex)$ is
  - An unmarked horizontal line with y centroid coordinate < topWall and $\geq$ bottomWall and
  - Has a minX bounding box coordinate $\leq$ minX of $list_{in}(symbolIndex)$ and
  - Is longer than *maxLength*

  then let *maxLength* be the length of this line (maxX-minX) and let *mainLine* := *listIndex*

If *mainLine* = -1, return *symbolIndex*

else return *mainLine*

END OF ALGORITHM

The worst-case time complexity for START and OVERLAP are $O(n)$. In the worst case, OVERLAP scans the entire list first forward, and then backward, performing $O(1)$ operations for each element, giving $O(2n)$. START in the worst case will first scan the entire input forward and backward, in the case of a limit symbol which is rightmost in the input with the start of one of the limits being leftmost (again, $O(2n)$). Assuming that all symbols horizontally overlap, the subsequent call to OVERLAP then requires an additional $O(2n)$ steps. 2n + 2n ϵ $O(n)$. Therefore both algorithms are of linear time complexity in the worst case.

B.2 Main Parsing Algorithm

In this section we present the full algorithm described in chapter 4.

Let $list_{in}$ be a sorted list of preprocessed symbols

Let T be a tree with a single node at the root, T_{root}

Let S be a stack

Let Q be a queue

Let ParentNode, SymbolNode and RelationNode be tree nodes

Let Temp1 and Temp2 and s_{start} be integers

Let region R = $\{(0,0),(\infty,\infty)\}$

Obtain the index of the start symbol s_{start} = SP(R)

If $s_{start} \neq -1$, set the wall attributes of s_{start} to R, enqueue (s_{start},T_{root}) in Q, and mark the symbol at $list_{in}(s_{start})$

While Q is not empty

- (EXTRACT BASELINES)
  While Q is not empty
  - (Temp1,ParentNode) := dequeue(Q)
  - Assign to SymbolNode a new tree node with all the attributes of the symbol at Temp1 in $list_{in}$
  - Push (Temp1,SymbolNode) on S
  - R := wallAttributes(Temp1)
  - Temp2 := HOR($list_{in}$,Temp1)
  - While Temp2 $\neq$ -1
    * Mark the symbol at $list_{in}$(Temp2)
    * wallAttributes(Temp2) := wallAttributes(Temp1)
    * Let SymbolNode be a new tree node containing all attributes associated with $list_{in}$(Temp2)
    * Add SymbolNode as the last child of ParentNode
    * Push (Temp2,SymbolNode)
    * rightWall(Temp1) := minX(Temp2)
    * If Temp2 is a limit symbol, and Temp 1 is a horizontal line or open bracket, assign leftWall(Temp2) := maxX(Temp1)
    * Temp1 := Temp2
    * Temp2 := HOR($list_{in}$,Temp1)
  - Push "EOBL" on the stack
- (LOCATE SECONDARY BASELINES)
  While S is not empty
  - If top(Stack) = "EOBL" then Pop(S)
  - (Temp1,SymbolNode) := Pop(s)

  - Check all of the relevant secondary baseline regions for the symbol at index Temp1 in $list_{in}$ (TLEFT, BLEFT, ABOVE, BELOW, SUBSC, SUPER, CONTAINS - each calls START(R,$list_{in}$) for R defined for the region being examined (see Figure 4.4)). For each region examined which returns an result (i.e. Temp2 := START(R,$list_{in}$(Temp1)) and Temp2 $\neq$ -1), do the following:
    * Mark the symbol at $list_{in}$(Temp2)
    * wallAttributes(Temp2) := R
    * Let RelationNode be a new tree node labeled with the name of the matching relation.
    * Add RelationNode as a child of SymbolNode
    * Enqueue (Temp2, RelationNode) in Q

Scan the input. If any unmarked tokens remain, output an error indicating symbols in the input were not added to the baseline structure tree (this corresponds to Genarro Costagliola's "ANY" function[11]).

Return T

END OF ALGORITHM

The algorithm is of time complexity $O(n^2)$. At most n symbols are considered for each of the extract and locate processes (indicated in upper case letters); i.e. the inner loops execute $O(n)$ times each. HOR and START are both $O(n)$ as established earlier. This efficiency is due to the determinism of the parser.

An adjacency list representation may be used for the tree T. With an adjacency list, creating nodes is an $O(n)$ operation in the worst case. We never need to examine the tree during the process of the algorithm.

The maximum size of the tree is 2n, where n is the number of symbols in the input list. The worst case occurs when no baseline symbol set has more than one symbol as an element.

Vita

Name	Richard Zanibbi
Place and year of birth	Sudbury, Ontario, 1974
Education	Queen's University, 1993–1999
Experience	Teaching assistant, Department of Computing and Information Science, Queen's University, 1998 Research assistant, Department of Computing and Information Science, Queen's University, 1998-1999 Software Developer, Legasys Corporation, Kingston, Ontario, Canada, 1999
Awards	Wilfred Laurier University Scholarship(declined), 1993 Queen's Graduate Award, 1998-1999 University of Otago Optical Music Recognition Research Scholarship(declined), 1999